RUHR-UNIVERSITY BOCHUM

Horst Görtz Institute for IT Security

Technical Report HGI-TR-2008-001

Property-Based TPM Virtualization

Ahmad-Reza Sadeghi, Christian Stüble, Marcel Winandy

Chair for System Security
and
Sirrix AG security technologies

Ruhr-University Bochum
Horst Görtz Institute for IT Security
D-44780 Bochum, Germany

HGI-TR-2008-001
March 18, 2008

Property-Based TPM Virtualization

Ahmad-Reza Sadeghi
Ruhr-University Bochum
ahmad.sadeghi@trust.rub.de

Christian Stüble
Sirrix AG, Germany
stueble@sirrix.com

Marcel Winandy
Ruhr-University Bochum
marcel.winandy@trust.rub.de

Abstract

Today, virtualization technologies and hypervisors celebrate their rediscovery. Especially migration of virtual machines (VMs) between hardware platforms provides a useful and cost-effective means to manage complex IT infrastructures. A challenge in this context is the virtualization of hardware security modules like the Trusted Platform Module (TPM) since the intended purpose of TPMs is to securely link software and the underlying hardware. Existing solutions for TPM virtualization, however, have various shortcomings that hinder the deployment to a wide range of useful scenarios. In this paper, we address these shortcomings by presenting a flexible and privacy-preserving design of a virtual TPM that in contrast to existing solutions supports different approaches for measuring the platform's state and for key generation, and uses property-based attestation mechanisms to support software updates and VM migration. Our solution improves the maintainability and applicability of hypervisors supporting hardware security modules like the TPM.

Keywords: Trusted Computing, Trusted Platform Module, virtualization, migration

1 Introduction

Corporate computing today is characterized by enterprises managing their own IT infrastructure. In his article, "The end of corporate computing" [7], Nicholas G. Carr predicts a shift from holding corporate assets to purchasing services from third parties. Similar to electricity suppliers, there would be enterprises offering IT functionality to other companies. *Virtualization* technology would be one of the key drivers of the changing IT paradigm.

Indeed, virtualization enables the deployment of standardized operating environments on various hardware platforms, features the execution of several virtual machines (VMs) on a single platform, and allows to suspend a VM and resume it at a later time. An important feature of virtualization is that one can migrate a VM between hardware platforms, which allows an easy transfer of working environments, e.g., in case of hardware replacements or switching to another computer. Moreover, Virtual Machine Monitors (VMM), or *hypervisors*, are also known to be an efficient way to increase the security of computer systems [14, 15], since they provide isolation between VMs by mediating access to hardware resources and controlling a rather simple interface of resources compared to a full operating system. Thus, different environments can be protected against harm from other environments or violations of user privacy. For instance, an employee can simply separate home and office usage in VMs.

Trusted Computing is considered to be another promising concept to improve trustworthiness and security of computer systems. The Trusted Computing Group (TCG) [29], an industrial initiative towards the realization of Trusted Computing, has specified security extensions for commodity computing platforms. The core TCG specification is the *Trusted Platform Module* (TPM) [30, 31], currently implemented as cost-effective, tamper-evident hardware security module embedded in computer mainboards. The TPM provides a unique identity, cryptographic functions (e.g., key generation, hash function SHA-1, asymmetric encryption and signature), protected storage for small data (e.g., keys), and monotonic counters (storing values that can never decrease). Moreover, it provides the facility to securely record and report the platform state (so-called integrity measurements) to a remote party. The platform state typically consists of the hardware configuration and the software stack running on the platform, which is measured (using cryptographic hashing) and stored in the TPM. Several operating system extensions [20, 26] already support the TPM as underlying security module.

In this context, the combination of virtualization and trusted computing provides us with new security guarantees such as assurance about the booted VMM, but it also faces us with new challenges: On the one hand, VMs should be flexible to support migration. On the other hand, security modules like the TPM act as the *root of trust* attached to the hardware, and must be shared by various VMs. Hence, different approaches for TPM virtualization have already been proposed [5, 9, 22]. Being able to migrate a VM together with its associated virtual TPM (vTPM) is of special importance to guarantee the availability of protected data and cryptographic keys after migration. However, the existing solutions have some shortcomings which strongly limit their deployment: After migrating a VM and its vTPM to another platform with different integrity measurements than the source platform, or after performing an authorized update of software, the VM cannot access cryptographic keys, and thus the data protected by those keys anymore. This hinders the flexibility of migrating the VM to a platform providing the same security properties but probably different integrity measurements as the source platform. Moreover, differentiated strategies for key generation and usage are missing. Some IT environments demand for cryptographic keys generated and protected by the hardware TPM while some VMs would benefit from the performance of software keys. In addition, some VMs can be migratable while others must not be.

Contribution. In this paper we address these problems through the following contributions:

- We propose a vTPM architecture that supports various functions to measure the state of the platform, various usage strategies for cryptographic keys, and both based on a user-defined policy of the hypervisor system (Section 5).
- We show how to realize property-based attestation and sealing based on the new measurement functions of the vTPM. Our design can protect user privacy by filtering properties to be attested in order to not disclose the particular system configuration (Section 6).
- We allow a transparent migration of vTPM instances to platforms with a different binary implementation and show that is possible without losing the strong association of security properties (Section 7).

Moreover, our design does not require to modify the software of a VM (except for the driver in the guest OS that interfaces to the vTPM instead of the hardware TPM). Existing TPM-enabled applications directly profit from the flexibility of the underlying vTPM. We expect furthermore that our vTPM design can be realized based on other secure coprocessors [27, 32] because of its flexibility and high-level abstraction of functionality.

Outline. We first consider some typical use cases that need flexible vTPMs in Section 2 and define the corresponding requirements in Section 3. Section 4 considers some background of the TPM and discusses the related work. We present our contribution in Sections 5, 6, and 7, whereas we address how the requirements are achieved in Section 8.

2 Use Case Scenario: Corporate Computing

We consider the use case in a corporate setting as our running example to make various essential requirements on VMs and vTPMs more clear. Nevertheless, these requirements also hold for many other applications such as e-government, grid computing, and data centers.

Suppose an enterprise employee uses a laptop for both corporate and private tasks which run in isolated VMs (Figure 1).

Private working environment: This environment may use the TPM, e.g., to protect the key of a hard-disk encryption program or the reference values of an integrity checker. Using existing

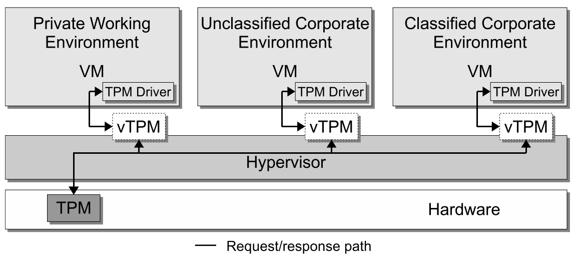
Private Working Environment
VM
TPM Driver
Unclassified Corporate Environment
VM
TPM Driver
Classified Corporate Environment
VM
TPM Driver
vTPM
vTPM
vTPM
Hypervisor
TPM
Hardware
Request/response path

Figure 1: Private and corporate working environments with virtual TPMs.

vTPM approaches, the protected data would become unavailable if the user updates a software component within the VM.

Unclassified corporate environment is for processing unclassified data of the company. Users should be able to migrate this VM to their computer at home to continue working. After migration, access to protected data and reporting integrity measurements of the VM should still be possible as long as the underlying platform conforms to the company's security policy.

Classified corporate environment: This environment is for processing classified data. Hence it has stronger security requirements regarding the usage of encryption keys. To access a corporate VPN, the company's security policy may require this environment to be bound to this specific hardware and that the cryptographic keys are protected by a physical TPM.

3 Requirements on TPM Virtualization

The scenarios described above show the need for a flexible vTPM architecture that supports all required functionalities. We consider below the main requirements of such an architecture, where we add new requirements *R5-R8* to those (*R1-R4*) already identified by [5].

R1 Confidentiality and integrity of vTPM state: All internal data of a vTPM (keys, measurement values, counters, etc.) have to be protected against unauthorized access.

R2 Secure link to chain of trust: There must be an unforgeable linkage between the hardware TPM and each vTPM as well as between the VM and its associated vTPM. This includes trust establishment by managing certificate chains from the hardware TPM to vTPMs.

R3 Distinguishability: Remote parties should be able to distinguish between a real TPM and a vTPM since a virtual TPM may have different security properties than a physical one.

R4 Uncloneability and secure migration: The state of a vTPM shall be protected from cloning, and it can be securely (integrity, confidentiality, authenticity) transferred to another platform if the destination platform is conform to the desired security policy.

R5 Freshness: The vTPM state shall not be vulnerable to replay attacks (e.g., an adversary shall not be able to reset the monotonic counters of a vTPM).

R6 Data availability: Data sealed by a vTPM should be accessible if the platform provides the desired security properties. This must also hold after migration or software updates.

R7 Privacy: Users should be able to decide which information about the platform state (configuration of hardware and hypervisor) is revealed to a VM or to a remote party.

R8 Flexible key types: Different protection levels and implementations of cryptographic keys should be supported (as described in the use case scenarios).

As we will discuss later, the existing vTPM solutions do not fulfill all requirements of the typical use cases as described in Section 2.

4 Background and Related Work

4.1 The Trusted Platform Module

The TPM has two main key (pairs): the *Endorsement Key* (EK) representing the TPM's identity and the *Storage Root Key* (SRK), used to encrypt other keys generated by the TPM (which are stored outside the TPM). The TPM supports *trusted boot* by allowing to record measurements of the hardware configuration and software stack during the boot process. These measurements (typically, SHA-1 hash of binaries) are stored in specific TPM registers called *Platform Configuration Registers* (PCRs). Adding a hash m to a PCR is called extension and requires to use the function TPM_Extend(i, m), which concatenates m to the current value of the i-th PCR by computing a cumulative hash.

Based on these PCR values, the TPM provides the *sealing* functionality, i.e., binding encrypted data to the recorded configuration, and *attestation*, i.e., reporting the system state to a (remote) party. The latter uses the function TPM_Quote, which presents the recorded PCR values signed by an *Attestation Identity Key* (AIK) of the TPM. The AIK plays the role of a pseudonym of the TPM's identity EK for privacy reasons, but to be authentic the AIK must be certified by a trusted third party called Privacy-CA.

4.2 Integrity Measurement

AEGIS [2] performs an integrity check during the boot process of the operating system and builds a chain of trust based on root reference values protected by special hardware. Enforcer [19] is a Linux kernel security module operating as integrity checker for file systems. It uses a TPM to verify the integrity of the boot process and to protect the secret key of an encrypted file system. IMA [26] inserts measurement hooks in functions relevant for loading executable code in Linux in order to extend the measurement chain to the application level.

4.3 Property-Based Attestation

TCG binary attestation has some important drawbacks: (i) disclosure of platform configuration information could be abused for digital fingerprinting, platform tracking (*security and privacy*) and discriminating against specific system configurations, (ii) lack of flexibility; data bound to a particular configuration is rendered inaccessible after system migration, update or misconfiguration (*data availability*), and (iii) less scalability due to necessary management of every trusted platform configuration. To tackle these problems, property-based approaches were proposed in the literature (see below): Instead of attesting hash values of binaries, they attest abstract properties describing the behavior of a program or system, e.g., that the hypervisor is certified according to a certain Common Criteria protection profile. The advantage is that properties can remain the same even if the binaries change (e.g., due to updates).

Haldar et al. [11] present a language-based approach where they exploit security properties of programming languages, e.g., type-safety. Their approach also allows to provide a mechanism for runtime attestation. However, it requires a trusted language-specific execution environment and is limited to applications written in that language.

Jiang et al. [13] have shown that it is also possible to have certificates stating that the key-holder of a certain public key has a desired property, e.g., to be an application running inside an untampered secure coprocessor with a certain configuration.

A pragmatic approach for property-based attestation uses property certificates [8, 21, 23]. A trusted third party (TTP) issues certificates $cert(pk_{TTP}, prop, m)$, signed by the TTP's public key pk_{TTP}, and stating that a binary with hash m has the property $prop$. When a PCR of the TPM is going to be extended with a measurement value, a *translation* function looks for a matching certificate. If the function can find and verify a matching certificate, it extends the PCR with the public key pk_{TTP} or, as proposed by [18], with a bit string representation of $prop$. If no certificate is found or if the verification fails, the PCR is extended with zero.

We apply translation functions to our vTPM (Section 5.1). We use the simple version of property certificates, e.g., issued by a corporate CA, certifying “approved by IT department”.

4.4 Trusted Channel

A *trusted channel* is a secure channel with the additional feature that it is bound to the configuration of the endpoint(s). The idea is to embed an attestation (binary or property-based) of the involved endpoint(s) in the establishment of the secure channel [10, 28]. Hence, each endpoint can get an assurance whether the counterpart complies with trust requirements before the secure channel is settled. Asokan et al. [3] describe a protocol that creates a secret encryption key that is not only bound to the TPM of the destination platform but also bound to the configuration of the trusted computing base (TCB). Binding a key to the configuration of the underlying TCB has been used with TPM [19] and secure coprocessors [13, 27].

4.5 TPM Virtualization

Berger et al. [5] propose an architecture where all vTPM instances are executed in one special VM. This VM also provides a management service to create vTPM instances and to multiplex the requests. To protect the vTPM state when it is stored on persistent memory, the state is encrypted using the sealing function of the physical TPM. Optionally, the vTPM instances may be realized in a secure coprocessor card. In order to extend the chain of trust, they link the vTPM to its underlying trusted computing base by mapping the lower PCRs of the real TPM to the lower PCRs of a vTPM. This is supposed to enable the vTPM to include the configuration of the underlying hypervisor platform during an attestation procedure.

However, this approach has the restriction that after migrating a VM and its vTPM to a different hypervisor platform, the VM cannot access data that was sealed by the vTPM on the source platform. In our approach, we show how property-based measurement can be realized in the vTPM while the interface to the VM remains the same as for binary attestation. This removes the restriction that migration is only possible between binary identical platforms.

The vTPM in [5] has a different certificate for its vEK compared to a real TPM, e.g., including a statement that it is virtual. Thus, a verifying party will be able to distinguish between a vTPM and a TPM. The authors discuss different strategies for trust establishment, i.e., the way new certificates are issued for a vTPM: (a) The vEK could be signed by the AIK of the physical TPM and the vTPM requests certificates for its vAIKs at a privacy CA. (b) The TPM could directly sign the vAIK with its AIK. (c) A local CA issues a certificate for the vEK of the vTPM. For our example scenario, we can choose the certificate strategy (c) since the employee’s company could serve as a local CA to issue these certificates.

GVTPM [22] is an architectural framework that supports various TPM models and different security profiles for each VM under the Xen hypervisor [4]. Moreover, GVTPM is not limited to TPM functionality and may be generalized to any security coprocessor. This is similar to our approach since we use a high-level abstraction of TPM functionality and support user-defined policies for vTPMs. However, in addition to GVTPM we consider different measurement functions and different key types inside each vTPM.

Anderson et al. [1] realize the implementation of vTPM instances as isolated domains instead of running all vTPMs in one privileged VM. Our architecture can also execute vTPM instances in isolated domains since our approach does not depend on a specific implementation.

In contrast to the previous approaches which realize virtual TPMs in software, Goldman and Berger [9] have specified additional commands that would be needed to enhance a physical TPM to directly support VMs. However, there is no such TPM chip model available at present.

5 Flexible vTPM Architecture

This section describes the general design of our vTPM architecture. For each VM that needs a vTPM, there is a separate vTPM instance. We assume the underlying hypervisor to protect the internal state and operations of each vTPM from any unauthorized access. This can be achieved by using a secure hypervisor as proposed in [24, 25], which enforces access control to resources and controls communication between virtual machines. A VM can only access its associated vTPM via the vTPMInterface.

Figure 2 shows the logical design of our vTPM. The main building blocks are the following: PropertyManagement represents the virtual PCRs and manages different mechanisms to store and read measurement values (Section 5.1); KeyManagement is responsible for creating and loading keys (Section 5.2); vTPMPolicy holds the user-defined policy of the vTPM instance (Section 5.3); CryptographicFunctions provide monotonic counters, random number generation, hashing, etc.; MigrationController is responsible for migrating the vTPM to another platform.

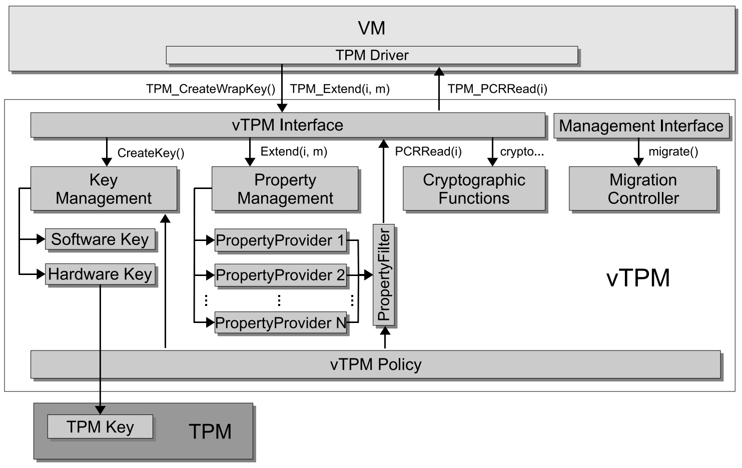

Figure 2: Logical architecture of the vTPM

5.1 Property Management and Property Providers

To improve flexible migration and to preserve the availability of sealed data after migration or software updates, an essential step is to support other measurement strategies. Applying property-based measurement and attestation [13, 18, 21, 23] to a vTPM allows for much more flexibility in the choice of the underlying hypervisor system and for easier updates of applications – while a VM can still use sealed data or run an attestation procedure if the properties of the programs remain the same (see Section 4.3).

We define the process of recording measurements into the TPM in a more general way. Therefore, we redefine the extension function of the TPM in the following way:

$$\mathsf{Extend}(i,\ m):\ PCR_i \leftarrow \mathsf{translate}(PCR_i, m).$$

In case of the TCG specification, translate is $SHA1(PCR_i||m)$.

Our vTPM design is based on a plug-in-like architecture for various vPCR extension strategies. Each extension strategy is realized by a PropertyProvider module implementing another translate() function. To add measurement values to the PCRs of the vTPM (vPCRs), the guest OS in a VM simply uses the standard TPM_Extend() function, specifying the PCR number i and the hash data m to be stored. The PropertyManagement calls each property provider to extend the corresponding vPCR with the measured data value. Each PropertyProvider applies its translation function on the data and stores the resulting value in the corresponding vPCR field. The general form of the PCR extension is as follows:

$$\text{PropertyProvider}_j.\text{Extend}(i,\ m)\text{: } vPCR_{i,j} \leftarrow \text{translate}_j(vPCR_{i,j}, m)$$

Note that each PropertyProvider has its own vector of virtual PCRs. Thus there is a matrix of vPCR values for each vTPM, as depicted in Figure 3. The way how to store the vPCR values is up to the implementation of each property provider. One could cumulatively hash all input values, as the TCG version of Extend. An alternative would be to simply concatenate the inputs on each invocation of Extend.

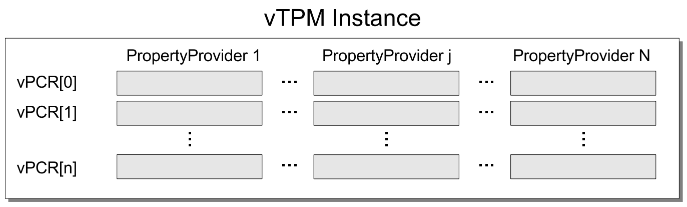

Figure 3: Matrix of vPCRs for a vTPM instance.

To give an example of different property providers, consider the virtual machine VM_k wants to extend PCR_i with a hash value m of a binary, e.g., when the guest OS within VM_k loads and measures a software component. The vTPM instance $vTPM_k$ is associated with VM_k. Suppose there are two PCR extension strategies, a HashProvider and a CertificateProvider. The HashProvider extends PCR_i with the hash m as provided by the VM. The CertificateProvider, however, looks for a property certificate (see Section 4.3).

In this example, the vTPM actually has two PCRs for PCR_i, i.e., $vPCR_{i.hash}$ and $vPCR_{i.cert}$. However, when VM_k requests to read the current PCR value, e.g., by invoking the function TPM_PCRRead(i), the VM is only aware of an abstract PCR_i and the returned data must be of fixed-length for compliance to the TCG specification. This is achieved by the PropertyFilter that defines, based on vTPMPolicy, which property provider has to be used when reading this particular vPCR. The responsible provider then returns the requested value.

5.2 Flexible Key Generation and Usage

To achieve a flexible key usage, the KeyManagement hides details of different strategies to create cryptographic keys when a VM requests a new key pair. The keys can be generated as software keys in the vTPM and as a result they are protected as part of the vTPM's state. Alternatively, the vTPM can delegate the key generation to a physical security module, e.g., a TPM or a smartcard. In this case, the keys are protected by the hardware.

For example, in our "classified" corporate VM scenario, it is required to have an encryption key protected by the physical TPM. When the VM requests to create the key at the vTPM, the KeyManagement delegates the request directly to the hardware TPM. Note that the VM cannot decide which key type to be used; instead, this is decided by the vTPM policy.

Although the vTPMPolicy can specify which type of key is to be used, not all combinations are possible. A vTPM cannot use a hardware AIK to sign the vPCRs because the vTPM does not possess the private key part of the AIK. However, the realization of KeyManagement is not limited to software and physical TPMs. Instead, the underlying flexibility allows the realization based on different hardware security modules while providing VMs compatibility to the TCG specification.

5.3 User-Defined vTPM Policy

The user of the hypervisor system can specify a vTPMPolicy per vTPM instance when the instance is created. The policy specifies what information about the system state is actually visible to the VM and, hence, to other systems the VM is allowed to communicate with. This is possible due to the selection of property providers, which define possible translations of measurement values. For all vTPM operations, the policy defines what property provider has to be used. For example, a policy can define to always use the CertificateProvider for sealing operations requested by the VM in order to enable flexible migration to a certified platform.

For each vTPM instance, the vTPMPolicy specifies the key strategy to be used. In this way, we can source out privacy issues the VM would have to handle otherwise. For instance, the policy decides when to use a particular vAIK and how often it can be used until the KeyManagement has to generate a new one.

5.4 Initialization of the vTPM

On its instantiation, the vTPM creates a new Endorsement Key (vEK) and a new Storage Root Key (vSRK). Certificates for the vEK and for vAIKs can be issued, e.g., by a local CA.

Existing vTPM solutions [5] propose to directly map the lower PCRs of the physical TPM to the lower vPCRs of the vTPM. These PCRs contain measurements of the BIOS, the bootloader, and the hypervisor. While this provides a linkage to the underlying platform, it is based on the hash values of binary code only, which hinders migration as discussed earlier.

In our solution, we map these PCR values by applying our property providers and build up a vPCR matrix, holding a vector of vPCRs for each property provider. How the mapping is actually done is up to the implementation of the property providers. After initialization of the platform by means of trusted boot, the physical TPM contains the measurements of the platform configuration. When a new vTPM instance is created by the hypervisor, the PropertyManagement of this vTPM requests the physical TPM to read out all PCRs, i.e., from PCR_0 to PCR_n. Then each property provider is invoked with the following function:

$$\mathsf{PropertyProvider}_j.\mathsf{initVirtualPCRs}\ (PCR_0,...,PCR_n)$$

To give some examples, PropertyProvider$_A$ could map the values of PCR_0,...,PCR_7 one to one to $vPCR_{0.A}$,...,$vPCR_{7.A}$, whereas PropertyProvider$_B$ could accumulate somehow all physical measurements into one single vPCR. Finally, PropertyProvider$_C$ could translate the PCR values into properties using property certificates. Thus, this approach allows to support different mapping strategies simultaneously.

By defining the vTPM policy accordingly, we can control which mapping will actually be used later. For instance, to support availability of sealed data after migration, we can define to use the certificate-based property provider when the VM wants to seal data to $vPCR_0$,...,$vPCR_7$. If flexible migration should not be allowed, we would define to use PropertyProvider$_A$, resulting in sealing data to binary measurements of the underlying platform.

6 Realization of Property-Based Functionality with Our vTPM

In this section we describe how we can use the feature of property providers to realize property-based attestation and property-based sealing in the vTPM.

6.1 Property-Based Attestation

The CertificateProvider is one example of a property provider that uses property certificates issued by a TTP. As mentioned in Section 5.1, CertificateProvider applies its translation function to extend $vPCR_{i.cert}$ with the TTP's public key pk_{TTP}. The attestation protocol works as follows: A verifier requests attestation to (PCR_i,...,PCR_j) of VM_k; the VM requests its vTPM to *quote* the corresponding vPCRs with the key identified by $vAIK_{ID}$:

$$(\mathit{pcrData},\ \mathit{sig}) = vTPM_k.\mathsf{Quote}(vAIK_{ID},\ \mathit{nonce},\ [i,...,j])$$

where *pcrData* denotes the quoted vPCR values, *sig* denotes the vTPM's signature on *pcrData* and *nonce*. Internally, the PropertyManagement of the vTPM decides according to the vTPMPolicy which property provider is to be used for attestation. If the CertificateProvider is chosen, then $vTPM_k$ will use its vAIK as identified by $vAIK_{ID}$ to sign the values of $vPCR_{[i,...,j].cert}$.

The verifier verifies the signature *sig* and whether *pcrData* represent the desired properties. Hence, we can use vTPMPolicy to restrict attestation to certain property providers, depending on the use case. This allows to control which information about the VM and the user's system is going to be revealed to a remote party and as a result fulfills our privacy requirement.

6.2 Property-Based Sealing

The sealing procedure of our vTPM works as follows. A virtual machine VM_k chooses a handle *vBindkeyID* of a binding key that was previously created in the virtual TPM instance $vTPM_k$, and then issues the sealing command to seal *data* under the set of virtual PCRs (PCR_i,...,PCR_j). The vTPM realizes the sealing function as follows:

```
$vTPM_k$.Seal($vBindkeyID$, [$i$,...,$j$], $data$):
    provider := vTPMPolicy.askForProvider([$i$,...,$j$]);
    FOR $l$ := $i$ TO $j$ DO $prop_l$ := provider.PCRRead($l$);
    $pk$ := KeyManagement.getPublicKey($vBindkeyID$);
    $ed$ := encrypt[$pk$]($i||prop_i||...||j||prop_j||data$);
    return $ed$.
```

The vTPM asks its vTPMPolicy which property provider to use, which can depend on the combination of vPCRs for the sealing operation. It requests the KeyManagement to load the corresponding binding key, retrieves the vPCR values of the specified PropertyProvider, and encrypts *data*, and the vPCR values with corresponding vPCR number. When the VM wants to unseal the data again, the vTPM proceeds as follows:

```
$vTPM_k$.UnSeal($vBindkeyID$, $ed$):
    ($sk$, $pk$) := KeyManagement.getKeyPair($vBindkeyID$);
    ($i||prop_i||...||j||prop_j||data$) := decrypt[$sk$]($ed$);
    provider := vTPMPolicy.askForProvider([$i$,...,$j$]);
    FOR $l$ := $i$ TO $j$ DO BEGIN
        $prop'_l$ := provider.PCRRead($l$);
        if ($prop'_l \neq prop_l$) return $\emptyset$;
    END
    return $data$.
```

The vTPM first loads the binding key pair identified by *vBindkeyID* and decrypts the sealed data *ed*. The vTPMPolicy decides again which PropertyProvider to use. The current vPCR values are compared to the values stored in the sealed data. Only if all matching pairs are equal, the plain text *data* is returned to VM_k.

Of course, a property provider like CertificateProvider is needed as one possible way to realize property-based sealing. This is especially interesting if the sealing is related to software components in the VM. Depending on the realization of the property provider, unsealing will be possible if the measured applications of the VM have been changed but still provide the same properties, i.e., the corresponding property certificate is available and valid.

Moreover, property-based sealing enables the availability of sealed data after migration of a VM and its corresponding vTPM to a platform with a different binary implementation. This can be achieved, e.g., by using a CertificateProvider for the $vPCR_{[0,...,7].cert}$, representing the properties of the underlying hypervisor platform. This measurement does not change after migration to a target platform having a certificate stating the same properties.

7 Migration of vTPM

Our vTPM migration protocol is based on the vTPM migration protocol in [5]. However, in contrast to [5] we do not use a migratable[1] TPM key to protect the session key but rather we

[1]There are various attributes for TPM keys. Migratable keys are allowed to be migrated to another TPM.

propose to embed the migration procedure in a *trusted channel.* As described in Section 4.4, the trusted channel allows to create a secret encryption key that is not only bound to the TPM of the destination platform but also bound to the configuration of its TCB. In our case the TCB comprises the vTPM and the hypervisor. The advantage of using such a trusted channel is that, once it has been established, it can be re-used for migration of several vTPM instances between the same physical platforms. Moreover, a transfer can even securely occur after the target machine has rebooted.

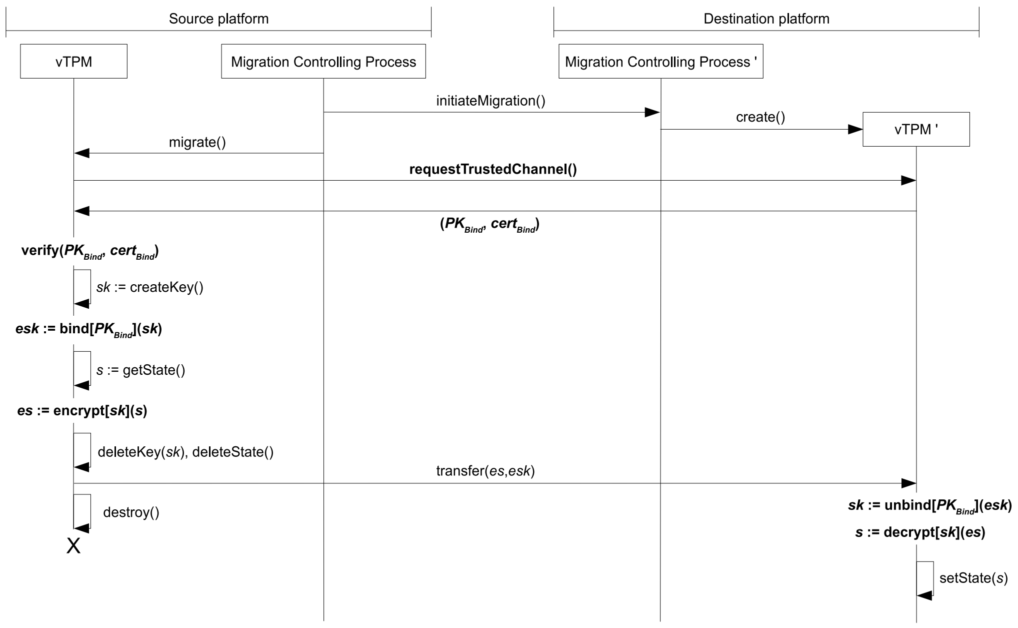

Figure 4: A vTPM migration based on a trusted channel.

Figure 4 shows our migration procedure, based on the trusted channel protocol of [3][2]. The process (of the hypervisor) responsible for migrating the VM also initiates the migration of the associated vTPM. After creating a new vTPM instance on the target system, the source vTPM requests to establish a trusted channel to the destination vTPM. When the trusted channel is successfully established, the source vTPM encrypts its state and transfers it to the destination. The source vTPM destroys itself subsequently, i.e., the vTPM deletes its own state from memory. On the target platform, the vTPM decrypts the state and activates it.

Additionally, there is another issue if the hypervisor supports to suspend a vTPM, i.e., if the vTPM state was stored on persistent memory before. If the suspended vTPM state is sealed to the hardware TPM (see Section 4.5), a migration of the suspended vTPM state ("offline migration") is not possible. However, we can resume a suspended vTPM (i.e., unseal the vTPM state) on the source platform, migrate the vTPM state to the target, and suspend and seal the vTPM state on the target platform to its hardware TPM, respectively. To ensure that the vTPM state is unique and cannot be reactivated at the source platform, the hypervisor has also to delete the key that is used to seal the vTPM state on persistent storage.

In order to prevent data loss from transmission failures during migration, the encrypted vTPM state can be stored persistently before transmission so that the state can be transmitted again to the target platform (if the migration is still pending and the keys of the trusted channel are still valid). Based on the ideas of [32], the encrypted state could be deleted on the source after the source receives an acknowledgment from the target.

[2]See also Appendix B.

8 Requirements Revisited

We briefly address the requirements of Section 3. Our architecture supports *flexible key types* by means of KeyManagement (Section 5.2). We have addressed *data availability* with PropertyManagement (Section 5.1) and property-based sealing (Section 6). To protect *privacy*, we make use property-based attestation and PropertyFilter, which controls the disclosure of properties according to the vTPM policy. The *inclusion in the chain of trust* is realized by mapping the PCRs of the physical TPM to the vTPM (Section 5.4). The requirement of *distinguishability* was already addressed by prior work (see Section 4.5). To protect the *confidentiality and integrity of vTPM state* and to maintain *uncloneability*, we can also resort to existing approaches, which we briefly discuss below (the details are out of scope of this paper).

Runtime protection of the vTPM state is assumed to be provided by the hypervisor through isolation. But to enable a VM and its vTPM to suspend and resume, all data belonging to the state of vTPM instance need to be protected against copying clones to other platforms or replaying old states on the local platform. In case the vTPM state has to be stored on persistent memory, prior work [5] encrypts the vTPM state using a key that is sealed to the state of PCRs in the hardware TPM, i.e., binding it to the configuration of the TCB.

To prevent a local replay of an old vTPM state, the sealed state has to be stored on storage providing freshness. For instance, [3] proposes a solution based on monotonic counters of the TPM. To prevent a replay of migration, the target platform needs to be able to detect the freshness of the transferred vTPM state. In [3] and [5], the source encrypts the data to be transferred together with a unique nonce that was defined by the target platform.

9 Conclusion and Future Work

We have presented a flexible and privacy-preserving design for virtual TPMs that supports different approaches for measuring the platform's state and for key generation. We have demonstrated that our design allows to implement property-based sealing and attestation in a vTPM. This enables the availability of protected data and cryptographic keys of the vTPM after migrating to another platform that provides the same security properties but may have a different binary implementation. TPM-enabled applications executed in a VM can directly profit from this flexibility without the need for modification.

The vTPM design is part of a security architecture that we currently implement. We are going to decompose the vTPM functionality into several services that can be used as required. Future work also includes the evaluation of performance and scalability. Moreover, flexible offline migration of vTPM states is an open issue which we will work on.

References

[1] Anderson, M. J., Moffie, M., and Dalton, C. I. Towards trustworthy virtualisation environments: Xen library os security service infrastructure. Tech. Rep. HPL-2007-69, Hewlett-Packard Laboratories, April 2007.

[2] Arbaugh, W. A., Farber, D. J., and Smith, J. M. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1997).

[3] Asokan, N., Ekberg, J.-E., Sadeghi, A.-R., Stüble, C., and Wolf, M. Enabling fairer digital rights management with trusted computing. In *Proceedings of the 10th Information Security Conference (ISC)* (2007), vol. 4779 of *Lecture Notes in Computer Science*, Springer.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Bolton Landing, NY, USA, Oct. 2003), ACM.

[5] Berger, S., Caceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th USENIX Security Symposium* (Aug. 2006), USENIX, pp. 305–320.

[6] Brickell, E., Camenisch, J., and Chen, L. Direct anonymous attestation. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington, DC, USA, Oct. 2004), ACM Press.

[7] Carr, N. G. The end of corporate computing. *MIT Sloan Management Review 46*, 3 (2005), 67–73.

[8] Chen, L., Landfermann, R., Loehr, H., Rohe, M., Sadeghi, A.-R., and Stüble, C. A protocol for property-based attestation. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing (STC'06)* (2006), ACM Press.

[9] Goldman, K., and Berger, S. TPM Main Part 3 – IBM Commands. `http://www.research.ibm.com/secure_systems_department/projects/vtpm/mai%nP3IBMCommandsrev10.pdf`, April 2005.

[10] Goldman, K., Perez, R., and Sailer, R. Linking remote attestation to secure tunnel endpoints. In *First ACM Workshop on Scalable Trusted Computing* (Nov. 2006), pp. 21–24.

[11] Haldar, V., Chandra, D., and Franz, M. Semantic remote attestation: A virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium* (May 2004). also Technical Report No. 03-20, School of Information and Computer Science, University of California, Irvine; October 2003.

[12] Intel Corporation. Intel Trusted Execution Technology, Preliminary Architecture Specification, August 2007.

[13] Jiang, S., Smith, S., and Minami, K. Securing web servers against insider attack. In *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC)* (2001).

[14] Karger, P. A., Zurko, M. E., Bonin, D. W., Mason, A. H., and Kahn, C. E. A VMM security kernel for the VAX architecture. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (Oakland, CA, May 1990), IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, pp. 2–19.

[15] Karger, P. A., Zurko, M. E., Bonin, D. W., Mason, A. H., and Kahn, C. E. A retrospective on the VAX VMM security kernel. *IEEE Transactions on Software Engineering 17*, 11 (Nov. 1991), 1147–1163.

[16] Kauer, B. OSLO: Improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium* (2007), pp. 229–237.

[17] Kursawe, K., Schellekens, D., and Preneel, B. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH - CRyptographic Advances in Secure Hardware* (2005).

[18] Kühn, U., Selhorst, M., and Stüble, C. Realizing property-based attestation and sealing with commonly available hard- and software. In *STC '07: Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing* (2007), ACM Press, pp. 50–57.

[19] MacDonald, R., Smith, S., Marchesini, J., and Wild, O. Bear: An open-source virtual secure coprocessor based on TCPA. Tech. Rep. TR2003-471, Department of Computer Science, Dartmouth College, 2003.

[20] Microsoft Corporation. Bitlocker drive encryption. `http://www.microsoft.com/technet/windowsvista/security/bitlockr.mspx`, July 2007.

[21] Poritz, J., Schunter, M., Van Herreweghen, E., and Waidner, M. Property attestation—scalable and privacy-friendly security assessment of peer computers. Tech. Rep. RZ 3548, IBM Research, May 2004.

[22] Rozas, C. Intel's Security Vision for Xen. `http://www.xensource.com/files/XenSecurity_Intel_CRozas.pdf`, April 2005.

[23] Sadeghi, A.-R., and Stüble, C. Property-based attestation for computing platforms: Caring about properties, not mechanisms. In *The 2004 New Security Paradigms Workshop* (2004), ACM Press.

[24] Sadeghi, A.-R., Stüble, C., and Pohlmann, N. European multilateral secure computing base - open trusted computing for you and me. *Datenschutz und Datensicherheit DuD, Verlag Friedrich Vieweg & Sohn, Wiesbaden 28*, 9 (2004), 548–554.

[25] Sailer, R., Valdez, E., Jaeger, T., Perez, R., van Doorn, L., Griffin, J. L., and Berger, S. sHype: Secure hypervisor approach to trusted virtualized systems. Tech. Rep. RC23511, IBM Research Division, Feb. 2005.

[26] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. Design and implementation of a TCG-based integrity measurement architecture. *13th Usenix Security Symposium, San Diego, California* (August 2004), 223–238.

[27] Smith, S. W., and Weingart, S. Building a high-performance, programmable secure coprocessor. *Computer Networks 31*, 8 (Apr. 1999), 831–860. Special Issue on Computer Network Security.

[28] Stumpf, F., Tafreschi, O., Röder, P., and Eckert, C. A robust integrity reporting protocol for remote attestation. In *Proceedings of the Second Workshop on Advances in Trusted Computing (WATC'06 Fall)* (Tokyo, December 2006).

[29] Trusted Computing Group. `http://www.trustedcomputinggroup.org`.

[30] Trusted Computing Group. TPM main specification. Main Specification Version 1.1b, Trusted Computing Group, Feb. 2002.

[31] Trusted Computing Group. TPM main specification. Main Specification Version 1.2 rev. 103, Trusted Computing Group, July 2007.

[32] Yee, B. S. *Using Secure Coprocessors.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1994. CMU-CS-94-149.

A Background on TPM

In this section we briefly review the TPM functionalities first. The TPM provides certain cryptographic functions (like asymmetric key generation, encryption and signing) and protected storage for small data (e.g. keys). The main key (pairs) are the following: The first one is the *Endorsement Key* (EK) representing the TPM identity. Usually the manufacturer creates and stores the EK together with a certificate before the TPM is shipped. The second one is the *Storage Root Key* (SRK), which the TPM generates by command of the owner/user of the platform. The SRK is used to protect other keys generated by the TPM, which are stored outside the TPM in an encrypted way. The private parts of these keys never leave the TPM. The third one is a signing key *Attestation Identity Key* (AIK), which is under the sole control of the TPM and is used to sign the content of specific registers called PCRs (see below). The AIK is used as a pseudonym of the TPM's identity for privacy reasons, but it has to be certified by a trusted third party called *Privacy Authority* (Privacy-CA) who verifies the TPM's EK certificate to assure that the AIK is valid and from a genuine TPM. In order to overcome the problem that the Privacy-CA can link transactions to a certain platform or to each other, TPM version 1.2 specification defines a cryptographic protocol called *Direct Anonymous Attestation* DAA [6] to eliminate this CA and to provide unlinkability of transactions.

The TPM contains specific volatile storage *Platform Configuration Registers* (PCRs) to hold the so-called integrity measurement values. A *measurement* function takes a piece of code as input to compute a value from which one can derive the integrity of the code by comparing it to a reference value. Typically, a cryptographic hash function like SHA-1 is used. The PCRs can only be *written* using the function $\mathsf{TPM\_Extend}(i, m)$, which means a newly measured value m is stored in the i-th PCR by concatenating it to the previous hash value stored there:

$$\mathsf{TPM\_Extend}(i,\, m)\colon\ PCR_i \leftarrow SHA1(PCR_i||m).$$

The function $\mathsf{TPM\_PCRRead}(i)$, in turn, returns the current content of PCR_i. Moreover, since TPM version 1.2 specification [31], TPMs provide at least four monotonic counters. Values of such a counter can only increase and never decrease. Based on these TPM features the following main functionalities are provided:

Trusted Boot: At system start (e.g., on a PC) the BIOS extends the PCRs with measurements of the hardware configuration and the bootloader code. The bootloader extends the PCRs with measurements of the operating system, which in turn can extend measurements of applications. Finally, integrity measurements of loaded software components are stored by the PCRs.

Sealing: This functionality can encrypt data with a TPM key and (usually) binds the decryption to certain PCRs. The TPM can decrypt sealed data only if the corresponding PCRs have the same values as at the encryption time, i.e., the system was booted with the same platform measurement.

Attestation is used to report the system state to a (remote) party by presenting the integrity measurements of the PCRs. For authenticity of the measurement, the TPM digitally signs the current PCR values (and a nonce for freshness) with an AIK.

Trust model: The TPM is mainly intended to protect against software attacks according to the trust model of the TCG. Though, TPM chips are derived from smartcard technology and provide measures against certain side channel or hardware attacks. However, sophisticated physical attacks on the communication link (LPC bus) between CPU and the TPM are possible [17, 16]. However, new CPU generations integrate the TPM to the processor chip or certain trusted computing functionality (e.g., [12]), which makes such attacks more difficult. In this paper we adhere to the trust model of the TCG.

B Trusted Channel Protocol

As described in [3], the TPM of the destination endpoint creates a binding key pair (PK_{Bind},SK_{Bind}), whereas the private key SK_{Bind} can only be used by the TCB configuration as measured in the TPM of the destination platform at its initialization. The TPM further signs the binding key PK_{Bind} and the configuration of the TCB at the destination platform with an AIK, resulting in the certificate $cert_{Bind}$. The source endpoint receives (PK_{Bind}, $cert_{Bind}$) from the destination and verifies the configuration of the target TCB as certified by $cert_{Bind}$. It creates a new symmetric encryption key sk and binds it to PK_{Bind}, i.e., sk is encrypted with the binding key, resulting in esk. Subsequently, data can be encrypted with sk. The encrypted data is transferred together with the encrypted key esk to the destination.

On the destination, the trusted channel operation unbind is used to retrieve the symmetric key sk. This operation uses the physical TPM to decrypt esk under the condition that the measurement values in the PCRs map to the TCB configuration the key was bound to. After successful unbinding the destination platform has the key sk and can decrypt the data.