vTPM: Virtualizing the Trusted Platform Module

Stefan Berger *Ramón Cáceres* *Kenneth A. Goldman*
Ronald Perez *Reiner Sailer* *Leendert van Doorn*
{stefanb, caceres, kgoldman, ronpz, sailer, leendert}@us.ibm.com
IBM T. J. Watson Research Center
Hawthorne, NY 10532 USA

Abstract

We present the design and implementation of a system that enables trusted computing for an unlimited number of virtual machines on a single hardware platform. To this end, we virtualized the Trusted Platform Module (TPM). As a result, the TPM's secure storage and cryptographic functions are available to operating systems and applications running in virtual machines. Our new facility supports higher-level services for establishing trust in virtualized environments, for example remote attestation of software integrity.

We implemented the full TPM specification in software and added functions to create and destroy virtual TPM instances. We integrated our software TPM into a hypervisor environment to make TPM functions available to virtual machines. Our virtual TPM supports suspend and resume operations, as well as migration of a virtual TPM instance with its respective virtual machine across platforms. We present four designs for certificate chains to link the virtual TPM to a hardware TPM, with security vs. efficiency trade-offs based on threat models. Finally, we demonstrate a working system by layering an existing integrity measurement application on top of our virtual TPM facility.

1 Introduction

Hardware virtualization has enjoyed a rapid resurgence in recent years as a way to reduce the total cost of ownership of computer systems [5]. This resurgence is specially apparent in corporate data centers such as web hosting centers, where sharing each hardware platform among multiple software workloads leads to improved utilization and reduced operating expenses.

However, along with these cost benefits come added security concerns. Workloads that share the same platform must often be kept separate for a multitude of reasons. For example, government regulations may require an investment bank to maintain a strict separation between its market analysis and security underwriting departments, including their respective information processing facilities. Similarly, commercial interests may dictate that the web sites of competing businesses not have access to each other's data. In addition, concerns about malicious software subverting normal operations become specially acute in these shared hardware environments. For example, a remote client of a medical services site would like to determine that the server is not running corrupted software that will expose private information to a third party or return wrong medical information. The increasing use of virtualization thus gives rise to stringent security requirements in the areas of software integrity and workload isolation.

The combination of a hardware-based root of trust such as the Trusted Platform Module (TPM) [23], and a virtual machine-based system such as Xen [4], VMware [26], or PHYP [14], is exceedingly well suited to satisfying these security requirements. Virtual machine monitors, or hypervisors, are naturally good at isolating workloads from each other because they mediate all access to physical resources by virtual machines. A hardware root of trust is resistant to software attacks and provides a basis for reasoning about the integrity of all software running on a platform, from the hypervisor itself to all operating systems and applications running inside virtual machines.

In particular, the TPM enables remote attestation by digitally signing cryptographic hashes of software components. In this context, attestation means to affirm that some software or hardware is genuine or correct. TPM chips are widely deployed on laptop and desktop PCs, and are becoming increasingly available on server-class machines such as the IBM eServer x366 [12].

Virtualizing the TPM is necessary to make its capabilities available to all virtual machines running on a platform. Each virtual machine with need of TPM functionality should be made to feel that it has access to its own

private TPM, even though there may be many more virtual machines than physical TPMs on the system (typically there is a single hardware TPM per platform). It is thus necessary to create multiple virtual TPM instances, each of which faithfully emulates the functions of a hardware TPM.

However, virtualizing the TPM presents difficult challenges because of the need to preserve its security properties. The difficulty lies not in providing the low-level TPM command set, but in properly supporting higher-level security concepts such as trust establishment. In particular, it is necessary to extend the chain of trust from the physical TPM to each virtual TPM via careful management of signing keys and certificates. As a result, some application and operating system software that relies on TPM functionality needs to be made aware of semantic differences between virtual and physical TPMs, so that certificate chains can be correctly built and evaluated, and trust chains correctly established and followed.

An additional challenge is the need to support migration of a virtual TPM instance between hardware platforms when its associated virtual machine migrates. The ability to suspend, migrate, and resume virtual machines is an important benefit of hardware virtualization. For the virtual TPM, migration requires protecting the secrecy and integrity of data stored in a virtual TPM instance during the transfer between platforms, and re-establishing the chain of trust on the new platform.

This paper presents the design and implementation of a virtual TPM (vTPM) facility. This work makes the following contributions:

- It identifies the requirements for a vTPM, including those related to migration between hardware platforms.
- It introduces a vTPM architecture that meets these requirements, including extensions to the standard TPM command set and a protocol for secure vTPM migration.
- It describes our implementation of this vTPM architecture on Xen, including support for remote integrity attestation of the complete system: boot loader, hypervisor, vTPM subsystem, operating systems, and applications.
- It discusses four alternative schemes for certifying a vTPM's security credentials, including the tradeoffs involved in choosing between them.
- It demonstrates that our vTPM facility works by running an existing TPM application inside Xen virtual machines.

This work can also serve as a template for how to virtualize other security-related devices such as secure co-processors. Virtualizing such devices presents similar challenges to those outlined above for TPMs.

The rest of this paper is organized as follows. Section 2 introduces background concepts useful for understanding the ensuing material. Section 3 presents the requirements on a virtual TPM facility. Sections 4, 5, and 6 respectively describe the design, the implementation, and a sample application of our vTPM facility. Section 7 discusses open issues, Section 8 covers related work, and Section 9 concludes the paper.

2 Background

In this section we give some background on the two technologies that are basic to understanding this paper: the Trusted Platform Module (TPM) and the Virtual Machine Monitor (VMM).

2.1 The Trusted Platform Module

The TPM is a security specification defined by the Trusted Computing Group [23]. Its implementation is available as a chip that is physically attached to a platform's motherboard and controlled by software running on the system using well-defined commands [11]. It provides cryptographic operations such as asymmetric key generation, decryption, encryption, signing and migration of keys between TPMs, as well as random number generation and hashing. It also provides secure storage for small amounts of information such as cryptographic keys. Because the TPM is implemented in hardware and presents a carefully designed interface, it is resistant to software attacks [3].

Of particular interest is the Platform Configuration Register (PCR) *extension* operation. PCRs are initialized at power up and can only be modified by reset or extension. The PCR extension function cryptographically updates a PCR using the following function:

$$Extend(PCR_N, value) = SHA1(PCR_N || value)$$

The cryptographic properties of the extension operation state that it is infeasible to reach a certain PCR state through two different sequences of values. SHA1 refers to the Secure Hash Algorithm standard [19]. The $||$ operation represents a concatenation of two byte arrays.

PCR extensions are used during the platform boot process and start within early-executed code in the Basic Input/Output System (BIOS) that is referred to as the Core Root of Trust for Measurement (CRTM) [24]. Hash values of byte arrays representing code or configuration data are calculated, or *measured*, and PCRs are extended with these values. A final PCR value represents this accumulation of a unique sequence of measurements. Along

with a sequential list of individual measurements and applications' names and information about measured configuration data, PCR values are used to decide whether a system can be trusted. A transitive trust model is implemented that hands off the measuring from the BIOS [24] to the boot loader [18] and finally to the operating system. Procedures have also been developed for operating systems to measure launched applications, scripts and configuration files [21].

Besides the aforementioned cryptographic operations it is possible to *seal* information against the state of the TPM, where its state is represented through a subset of PCRs. Sealed information is encrypted with a public key and can only be decrypted if the selected PCRs are in the exact state that they were at the time of sealing.

There are a number of signing keys associated with a TPM. Each TPM can be identified by a unique built-in key, the Endorsement Key (EK), which stands for the validity of the TPM [10]. The device manufacturer should provide a certificate for the EK. Related to the EK are Attestation Identity Keys (AIKs). An AIK is created by the TPM and linked to the local platform through a certificate for that AIK. This certificate is created and signed by a certificate authority (CA). In particular, a *privacy CA* allows a platform to present different AIKs to different remote parties, so that it is impossible for these parties to determine that the AIKs are coming from the same platform. AIKs are primarily used during quote operations to provide a signature over a subset of PCRs as well as a 160-bit nonce. Quotes are delivered to remote parties to enable them to verify properties of the platform.

2.2 Virtual Machine Monitors

VMMs [8], also known as hypervisors, allow multiple operating systems to simultaneously run on one machine. A VMM is a software layer underneath the operating system that meets two basic requirements:

- It provides a Virtual Machine (VM) abstraction that models and emulates a physical machine.
- It provides isolation between virtual machines.

The basic responsibility of a VMM is to provide CPU time, memory and interrupts to each VM. It needs to set up the page tables and memory management unit of the CPU such that each VM runs in its own isolated sandbox. The hypervisor itself remains in full control over the resources given to a VM. During the boot process of a VMM, often an initial virtual machine is started that serves as a management system for starting further virtual machines.

Depending on the fidelity of the emulation of a physical machine, it may be necessary to make modifications to an operating system for it to run on a VMM. If modifications are required the environment is said to be *paravirtualized*, otherwise the VMM is said to provide a *fully virtualized* environment.

3 Requirements

A virtual TPM should provide TPM services to each virtual machine running on top of a hypervisor. The requirements discussed in this section can be summarized as follows:

1. A virtual TPM must provide the same usage model and TPM command set to an operating system running inside a virtual machine as a hardware TPM provides to an operating system running directly on a hardware platform.
2. A strong association between a virtual machine and its virtual TPM must be maintained across the life cycle of virtual machines. This includes migration of virtual machines together with their associated virtual TPMs from one physical machine to another.
3. A strong association between the virtual TPM and its underlying trusted computing base (TCB) must be maintained.
4. A virtual TPM must be clearly distinguishable from a hardware TPM because of the different security properties of the two types of TPM.

As much software as possible that was originally written to interact with a hardware TPM should run unmodified with a virtual TPM. It should remain unaware of the fact that it is communicating with a software implementation of a TPM in a virtual environment. An example of software that should remain unmodified is the TCG Software Stack (TSS) [25] that issues low-level TPM requests and receives low-level TPM responses on behalf of higher-level applications.

The requirement that software be unaware that it is using emulated devices is basic to virtualization and has already been achieved for a wide range of devices found in modern computers. Open-source software such as QEmu [1], as well as proprietary products like VMWare Workstation [26], have been successful in emulating machine environments for personal computers. They provide transparent emulation for timers, interrupt controllers, the PCI bus, and devices on that bus.

However, as a security device the TPM presents new and challenging issues that preclude fully transparent virtualization. One challenge arises because modern virtual

machine monitors provide suspend and resume capabilities. This enables a user to freeze the state of an operating system and resume it at a later point, possibly on a different physical machine. A virtual TPM implementation must support the suspension and resumption of TPM state, including its migration to another system platform. During normal operation of the virtual TPM, as well as during and after these more sophisticated lifecycle operations, the association between the virtual TPM and its virtual machine must be securely maintained such that secrets held inside the virtual TPM cannot be accessed by unauthorized parties or other virtual machines.

Another challenge is to maintain the association of a virtual TPM to its underlying trusted computing base. PC manufacturers may issue a certificate for the TPM endorsement key (EK) that states that the TPM hardware is tightly coupled to the motherboard and correctly embedded into the BIOS for management. A challenger, validating a digital signature from such a TPM, can thus determine the correct embedding and operation of the remote TPM chip and establish the environmental security properties of the hardware TPM. In a virtualized environment, each operating system communicates with a virtual TPM that may be running as a user-space process inside its own virtual machine. The association of such a TPM with its underlying software and hardware platform is not only loose but also subject to change, e.g., during migration. Tracking this changing trusted computing base forms one major challenge in virtualizing a hardware TPM. Maintaining the ability of the virtual TPM to attest to its mutable trusted computing base forms another major challenge. It is necessary to enable remote parties that have established trust in the initial environment to also establish trust in the vTPM environment at a later point in time.

For example, the strong binding of TPM credentials to those of the hardware platform is important to challenging parties during remote attestation. The challenger must follow the trust chain from the target platform's hardware TPM through a virtual TPM and into the runtime environment of the associated virtual machine.

Further, since software TPM implementations do not usually offer the same security properties as hardware TPM implementations, the different types of TPMs should be distinguishable for remote parties relying on a TPM's correct functioning. A virtualized TPM's certificates can be used to give an interested party enough information to conclude relevant properties of the complete software, firmware, and hardware environment on which this TPM's correct operation depends. In practice, this can be realized by the certificate issuer embedding special attributes into the certificate, and the interested party validating the certificate and translating these attributes during remote attestation of security properties. Interestingly, as will become clear during our exposition, a software TPM can be as secure as a hardware TPM .

In summary, virtualizing the TPM is not achieved by merely providing TPM functionality to a virtual machine through device emulation. A virtual TPM must also provide the means for outside parties to establish trust in a larger software environment than is the case with hardware TPMs. It must also enable reestablishment of trust after a virtual machine is migrated to another platform. These requirements for providing virtual TPM functionality will be used as a guideline for the following sections on architecture and implementation, as well as our final discussion.

4 Architecture

We designed a virtual TPM facility in software that provides TPM functionality to virtual machines. This section first describes the structure of the vTPM and the overall system design. It proceeds with describing our extensions to the TPM 1.2 command set to support virtualization of the TPM. Then it introduces our protocol for virtual TPM migration and concludes with considering security aspects of the vTPM platforms and run-time environments involved in the migration.

Figure 1 illustrates the vTPM building blocks and their relationship. The overall vTPM facility is composed of a vTPM manager and a number of vTPM instances. Each vTPM instance implements the full TCG TPM 1.2 specification [11]. Each virtual machine that needs TPM functionality is assigned its own vTPM instance. The vTPM manager performs functions such as creating vTPM instances and multiplexing requests from virtual machines to their associated vTPM instances.

Virtual machines communicate with the vTPM using a split device-driver model where a client-side driver runs inside each virtual machine that wants to access a virtual TPM instance. The server-side driver runs in the virtual machine hosting the vTPM.

4.1 Associating vTPM Instances with their Virtual Machines

As shown in Figure 1, multiple virtual machines send TPM commands to the virtual TPM facility. A difficulty arises because it cannot be determined from the content of a TPM command from which virtual machine the command originated, and thereby to which virtual TPM instance the command should be delivered. Our solution is for the server-side driver to prepend a 4-byte vTPM instance identifier to each packet carrying a TPM command. This number identifies the vTPM instance to which a virtual machine can send commands. The in-

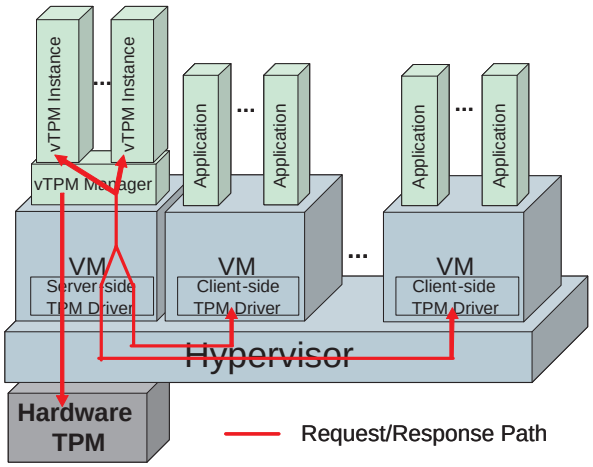

Figure 1: vTPM Architecture

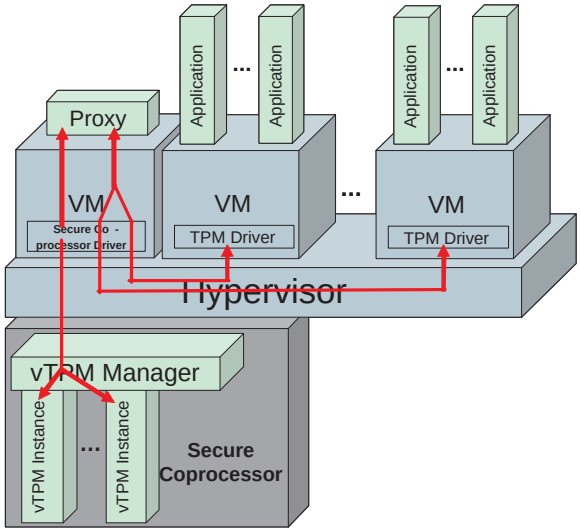

Figure 2: vTPM running inside a Secure Co-processor.

stance number is assigned by the virtual TPM when the vTPM instance is created.

Every VM must associate with a unique vTPM instance. The vTPM instance number is prepended on the server side so that virtual machines cannot forge packets and try to get access to a vTPM instance that is not associated with them. A command's originating virtual machine can be determined from the unique interrupt number raised by each client-side driver.

Since a TPM holds unique persistent state with secret information such as keys, it is necessary that a virtual machine be associated with its virtual TPM instance throughout the lifetime of the virtual machine. To keep this association over time, we maintain a list of virtual-machine-to-virtual-TPM-instance associations.

Figure 1 shows our architecture where TPM functionality for all VMs is provided by a virtual TPM running in the management VM. TPM functionality for this VM is provided by the hardware TPM, and is used in the same way as in a system without a hypervisor where the operating system owns the hardware TPM.

A variation of this architecture is shown in Figure 2 where virtual TPM functionality is provided by an external secure coprocessor card that provides maximum security for sensitive data, such as private keys, through a tamper-responsive environment. Here the first VM is the owner of this hardware and uses one virtual TPM instance for its own purposes. All other instances are reserved for usage by other virtual machines. A proxy process forwards TPM messages between the server-side driver and the external card.

4.2 The Root vTPM Instance

Our design was driven by the goal of having a virtual TPM implementation that can be run on an external co-processor card as well as executed as a process running within a virtual machine. We designed the virtual TPM such that the interaction of applications with either implementation would be the same. New commands and APIs that we introduce should work the same for both implementations. Considerations regarding reducing the trusted computing base of the environment hosting the virtual TPM did not directly influence the design, although the intention is to have a virtual machine that is dedicated exclusively to providing virtual TPM functionality.

Further, modern hypervisors support advanced features such as virtual machine hibernation and migration from one physical platform to another. The straightforward approach to supporting such features is to hibernate and migrate a virtual TPM instance along with its associated virtual machine, thus preserving existing measurements and avoiding the complexity of remeasuring running software in a new environment and accounting for the loss of measurements representing software that was loaded but is no longer running. However, the virtual TPM migration process must offer more security guarantees for the virtual TPM instance state than is usually provided for an operating system image that is being transferred. The virtual TPM migration process must guarantee that any vTPM instance state in transit is not subject to modification, duplication, or other compromise.

This set of requirements led us to design a virtual TPM as a TPM capable of spawning new vTPM child instances. Having an always available vTPM *root in-*

stance provides an entity that has cryptographic capabilities for generating asymmetric keys, handling encryption and decryption of data, and migration of asymmetric keys between virtual TPMs. The ability to handle keys and encrypt data with them enables us to encrypt the state of a vTPM instance when migrating it. The virtual TPM's ability to migrate keys to another virtual TPM makes it possible to exchange encrypted data between virtual TPMs.

Since the ability to spawn – and generally to manage – new virtual TPM instances is a fairly powerful feature, this capability should only be accessible to the owner of the root instance. The administrator of the initial virtual machine, who has the ability to start new virtual machines, would own this capability. We designed all TPM command extensions to require owner authorization in the same way as some of the existing TPM commands do. In effect, the TPM verifies that such command blocks are authorized with the owner's password.

We introduced the concept of a privileged vTPM instance. A privileged vTPM instance can spawn and manage child vTPM instances. Since being a privileged instance is an inheritable property, an instance may pass this privilege on to its own children. Using this inheritance scheme, we can support building a hierarchy of vTPMs in parallel to a hierarchy of virtual machines where each virtual machine is given the privilege of starting other virtual machines.

4.3 Independent Key Hierarchies

The TPM specification demands that a TPM establish a storage root key (SRK) as the root key for its key hierarchy. Every key that is generated has its private key encrypted by its parent key and thus creates a chain to the SRK. In our virtual TPM we create an *independent key hierarchy* per vTPM instance and therefore unlink every vTPM instance from the key hierarchy of a possible hardware TPM. This has the advantage that key generation is much faster since we do not need to rely on the hardware TPM for this. It also simplifies vTPM instance migration.

Similarly, we generate an endorsement key (EK) per vTPM instance. This enables TPM commands that rely on decrypting information with the private part of the EK to also work after a virtual TPM has migrated.

If the SRK, EK or any other persistent data of virtual TPMs are written into persistent memory, they are encrypted with a symmetric key rooted in the hardware TPM by for example sealing it to the state of the hardware TPM's PCRs during machine boot, or by encrypting it using a password-protected key. We therefore earn the flexibility of managing each virtual TPM's key hierarchy independently. In addition, by using file-level data encryption we mitigate the cost of not directly coupling the key hierarchy of a virtual TPM instance to that of the hardware TPM.

4.4 Extended Command Set

In order to realize our design of a virtual TPM, we extended the existing TPM 1.2 command set with additional commands in the following categories.

- Virtual TPM Management commands
  - CreateInstance
  - DeleteInstance
  - SetupInstance
- Virtual TPM Migration commands
  - GetInstanceKey / SetInstanceKey
  - GetInstanceData / SetInstanceData
- Virtual TPM Utility commands
  - TransportInstance
  - LockInstance / UnlockInstance
  - ReportEnvironment

The *Virtual TPM Management commands* manage the life-cycle of vTPM instances and provide functions for their creation and deletion. The *SetupInstance* command prepares a vTPM instance for immediate usage by the corresponding virtual machine and extends PCRs with measurements of the operating system kernel image and other files involved in the boot process. This command is used for virtual machines that boot without the support of a TPM-enabled BIOS and boot loader, which would otherwise initialize the TPM and extend the TPM PCRs with appropriate measurements.

The *Virtual TPM Migration commands* support vTPM instance migration. We implemented a secure virtual TPM instance migration protocol that can securely package the state of a virtual TPM instance and migrate it to a destination platform. Our extended commands enforce that the content of a vTPM instance is protected and that a vTPM instance can only be migrated to one target platform destination, thus preventing duplication of a vTPM instance and ensuring that a virtual TPM is resumed in association with its VM.

One of the *Virtual TPM Utility commands* offers a function for routing a limited subset of TPM commands from a vTPM parent instance to one of its child instances. This command works similar to IP tunneling, where an embedded packet is unwrapped and then routed to its destination. Embedding a command is useful since the association of a virtual machine to a privileged virtual TPM does not allow direct communication with a child

vTPM instance. For example, we use this command to create an endorsement key for a virtual TPM after the child instance has been created and before it is used by its associated virtual machine. Other functions in the utility category include locking a vTPM instance to keep its state from being altered while its state is serialized for migration, and unlocking it to make it available for use after migration has completed.

4.5 Virtual TPM Migration

Since vTPM instance migration is one of the most important features that we enabled through the command set extension, we explain how it works in more detail. The virtual TPM migration procedure is depicted in Figure 3.

We enabled vTPM instance migration using asymmetric and symmetric keys to encrypt and package TPM state on the source virtual TPM and decrypt the state on the destination virtual TPM. We based vTPM migration on migrateable TPM storage keys, a procedure that is supported by the existing TPM standard.

The first step in our vTPM instance migration protocol is to create an empty destination vTPM instance for the purpose of migrating state. The destination virtual TPM generates and exports a unique identifier (Nonce). The source vTPM is locked to the same Nonce. All TPM state is exported with the Nonce, and the Nonce is validated before import. This enforces uniqueness of the virtual TPM and prevents TPM state from being migrated to multiple destinations.

The next step involves marshaling the encrypted state of the source vTPM. This step is initiated by sending to the source vTPM a command to create a symmetric key. The key is encrypted with a parent TPM instance storage key. The blobs of state encrypted with a symmetric key are then retrieved from the source vTPM. This includes NVRAM areas, keys, authorization and transport sessions, delegation rows, counters, owner evict keys, and permanent flags and data. While the state is collected, the TPM instance is locked so the state cannot be changed by normal usage. After each piece of state information has been serialized, an internal migration digest is updated with the data's hash and the piece of state information becomes inaccessible. The migration digest is embedded into the last piece of state information and serves for validation on the target side.

To recreate the state of the virtual TPM on the destination platform, the storage key of the vTPM parent instance (used to encrypt the symmetric key used to protect the vTPM instance state) must be migrated to the destination vTPM parent instance. After the decryption of the symmetric key, the migrating vTPM's state is recreated and the migration digest recalculated. To detect possible Denial of Service (DoS) attacks where untrusted software involved in migration alters or omits state, operation of the vTPM instance can only resume if the calculated migration digest matches the transmitted one.

Support for Live Migration Modern virtual machine monitors support *live migration* [2] of virtual machines from one platform to another. Live migration tries to shorten downtimes by replicating the running system's image on a destination machine and switching execution to that machine once all pages have been replicated. Live migration can be supported with our virtual TPM migration protocol, but will in the worst case extend the downtime of the migrated system by the time it takes to complete an outstanding TPM operation, transfer the vTPM state, and recreate it on the destination platform.

4.6 Linking a vTPM to its TCB

Both architectures we introduced in Section 4.1 – a vTPM hosted in a virtual machine or in a secure coprocessor – provide TPM functionality to virtual machines. It is therefore possible to enable an integrity measurement facility [13] in each virtual machine and record application measurements in the virtual TPM. However, it is necessary that a challenger can establish trust in an environment which consists of more than the content of the virtual machine. The reason is that each operating system is running inside a virtual machine that is fully controlled by the hypervisor. Furthermore, a virtual TPM can be running as a process inside a VM whose own execution environment must be trusted. Therefore it is necessary that attestation support within the virtualized environment not only allows a challenger to learn about measurements inside the virtual machine, but also about those of the environment that provides virtual TPM functionality. In addition, these measurements must include the hypervisor and the entire boot process.

Our architecture therefore merges the virtual TPM-hosting environment with that of the virtual machine by providing two different views of PCR registers. Figure 4 shows these two views. The lower set of PCR registers of a vTPM show the values of the hardware TPM and the upper ones reflect the values specific to that vTPM. This way, a challenger can see all relevant measurements. The providers of the measurements extended into the different PCRs –BIOS, boot loader, and operating system– are denoted beside the PCRs. BIOS measurements include measurements of the boot stages and various hardware platform configurations. The boot loader measures, for example, the hypervisor and its configuration, the virtual machine monitor operating system kernel, initrd, and configuration. Then the VMM takes over and measures the dynamically activated VMM environment, such as

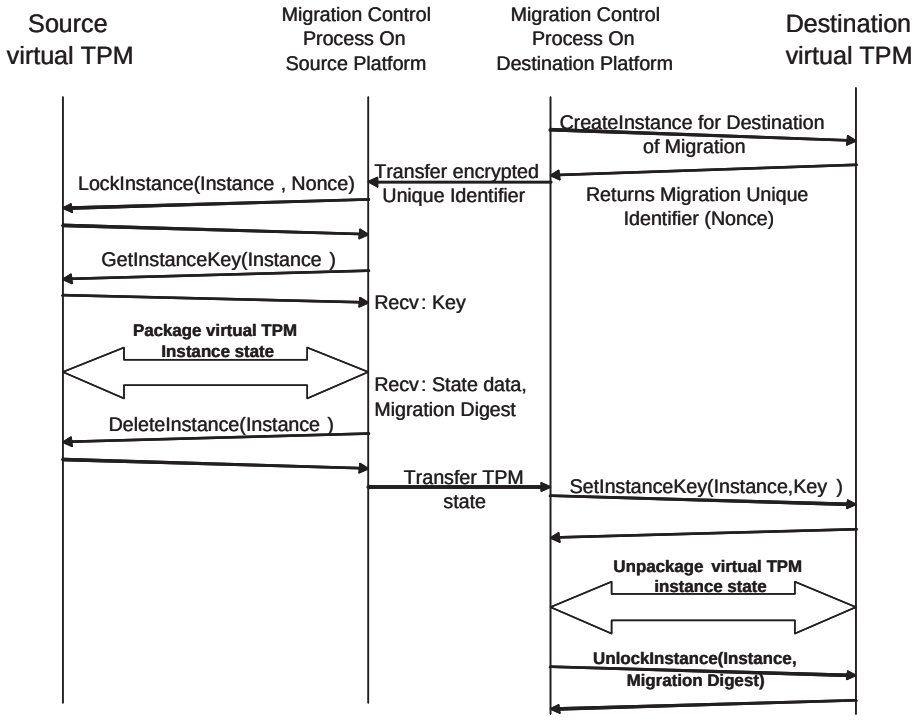

Figure 3: Virtual TPM Migration Protocol

the vTPM manager, and other components on which the correct functioning of the virtual environment and the vTPM depends.

As previously mentioned, the certificate for a virtual TPM instance does not necessarily stand for the same security guarantees as that of a hardware TPM. If a challenger decides that the security guarantees of the virtual TPM are not sufficient he can directly challenge the virtual machine owning the hardware TPM to verify the basic measurements including the one of the virtual TPM. Section 7.2 describes how certificates can be issued to mitigate this problem.

5 Implementation

In this section we present our implementation of virtual TPM support for the Xen hypervisor [27]. We expect that an implementation for other virtualization environments would be similar in the area of virtual TPM management, but will differ in the particular management tools and device-driver structure.

We have implemented the two previously discussed solutions of a virtual TPM. One is a pure software solution targeted to run as a process in user space inside a dedicated virtual machine (Figure 1) and the other runs on IBM's PCI-X Cryptographic Coprocessor (PCIXCC) card [15] (Figure 2).

5.1 Implementation for Xen

Xen is a VMM for paravirtualized operating systems that can also support full virtualization by exploiting emerging hardware support for virtualization. In Xen-speak, each virtual machine is referred to as a *domain*. *Domain-0* is the first instance of an OS that is started during system boot. In Xen version 3.0, domain-0 owns and controls all hardware attached to the system. All other domains are *user domains* that receive access to the hardware using *frontend device drivers* that connect to *backend device drivers* in domain-0. Domain-0 effectively proxies access to hardware such as network cards or hard drive partitions.

We have implemented the following components for virtual TPM support under the Xen hypervisor:

- Split device-driver pair for connecting domains to a virtual TPM
- Scripts to help connect virtual machines to virtual TPM instances
- Virtual TPM management tools
- Virtual TPM-specific extensions to Xen's management tools (e.g., xend, xm)
- Full TPM 1.2 implementation extended with our virtual TPM command set

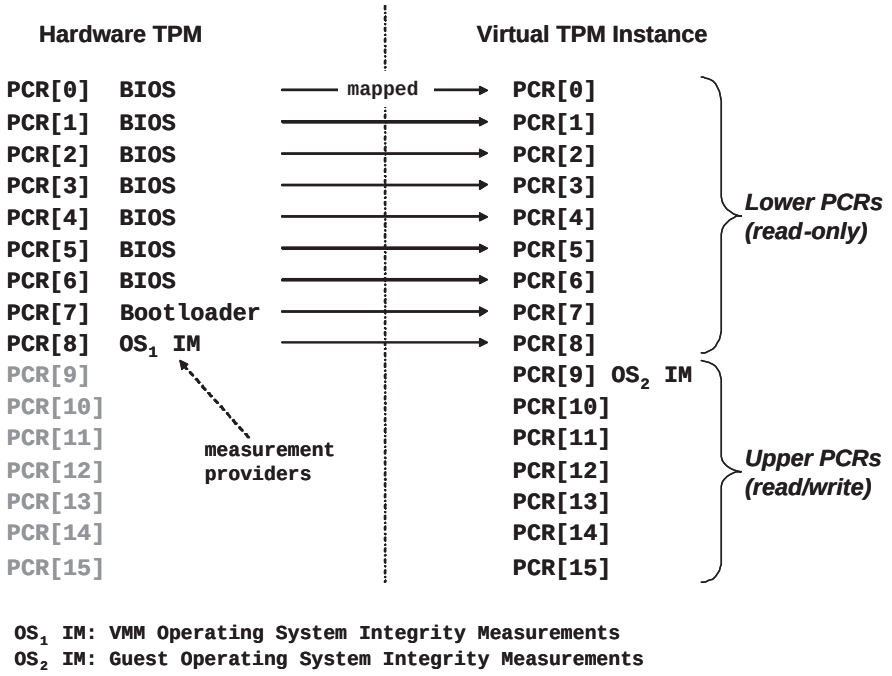

Figure 4: Mapping of Virtual TPM to Hardware TPM PCRs

We have extended the Xen hypervisor tools to support virtual TPM devices. *xm*, the Xen Management tool, parses the virtual machine configuration file and, if specified, recognizes that a virtual TPM instance must be associated with a virtual machine. *xend*, the Xen Daemon, makes entries in the *xenstore* [22] directory that indicate in which domain the TPM backend is located. Using this information, the TPM frontend driver in the user domain establishes a connection to the backend driver. During the connection phase, the backend driver triggers the Linux hotplug daemon that then launches scripts for connecting the virtual TPM instance to the domain.

Within our virtual TPM hotplug scripts, we need to differentiate whether the virtual machine was just created or whether it resumed after suspension. In the former case, we initialize the virtual TPM instance with a reset. In the latter case, we restore the state of the TPM from the time when the virtual machine was suspended. Inside the scripts we also administer a table of virtual-machine-to-virtual-TPM-instance associations and create new virtual TPM instances when no virtual TPM exists for a started virtual machine.

Figure 5 shows an example of a virtual machine configuration file with the virtual TPM option enabled. The attributes indicate in which domain the TPM backend driver is located and which TPM instance is preferred to be associated with the virtual machine. To eliminate configuration errors, the final decision on which virtual TPM instance is given to a virtual machine is made in the hotplug scripts and depends on already existing entries in the associations table.

```
kernel = "/boot/vmlinuz -2.6.12-xen"
ramdisk = "/boot/vmlinuz -2.6.12-xen.img"
memory = 256
name = "UserDomainWithTPM"
vtpm = ['backend=0, instance=1']
```

Figure 5: Virtual Machine Configuration File with vTPM Option

We have implemented the Xen-specific frontend driver such that it plugs into the generic TPM device driver that is already in the Linux kernel. Any application that wants to use the TPM would communicate with it through the usual device, /dev/tpm0. The backend driver is a component that only exists in the virtualized environment. There we offer a new device, /dev/vtpm, through which the virtual TPM implementation listens for requests.

Our driver pair implements devices that are connected to a Xen-specific bus for split device drivers, called *xenbus*. The xenbus interacts with the drivers by invoking

their callback functions and calls the backend driver for initialization when a frontend has appeared. It also notifies the frontend driver about the request to suspend or resume operation due to suspension or resumption of the user domain.

Suspension and resumption is an important issue for our TPM frontend driver implementation. The existing TPM protocol assumes a reliable transport to the TPM hardware, and that for every request that is sent a guaranteed response will be returned. For the vTPM driver implementation this means that we need to make sure that the last outstanding response has been received by the user domain before the operating system in that domain suspends. This avoids extension of the basic TPM protocol through a more complicated sequence number-based protocol to work around lost packets.

We use Xen's existing shared memory mechanism (*grant tables* [6]) to transfer data between front- and back-end driver. Initially a page is allocated and shared between the front and back ends. When data is to be transmitted they are copied into pages and an access grant to the pages is established for the peer domain. Using an event channel, an interrupt is raised in the peer domain which then starts reading the TPM request from the page, prepends the 4-byte instance number to the request and sends it to the virtual TPM.

The virtual TPM runs as a process in user space in domain-0 and implements the command extensions we introduced in Section 4.4. For concurrent processing of requests from multiple domains, it spawns multiple threads that wait for requests on /dev/vtpm and a local interface. Internal locking mechanisms prevent multiple threads from accessing a single virtual TPM instance at the same time. Although a TPM driver implementation in a user domain should not allow more than one unanswered TPM request to be processed by a single TPM, we cannot assume that every driver is written that way. Therefore we implemented the locking mechanism as a defense against buggy TPM drivers.

The virtual TPM management tools implement command line tools for formatting and sending virtual TPM commands to the virtual TPM over its local interface. Requests are built from parameters passed through the command line. We use these tools inside the hotplug scripts for automatic management of virtual TPM instances.

5.2 Implementation for the PCI-X Cryptographic Coprocessor

IBM's PCIXCC secure coprocessor is a programmable PCI-X card that offers tamper-responsive protection. It is ideally suited for providing TPM functionality in security-sensitive environments where higher levels of assurance are required, e.g., banking and finance.

The code for the virtual TPM on the card differs only slightly from that which runs in a virtual machine. The main differences are that the vTPM on the card receives its commands through a different transport interface, and it uses built-in cryptographic hardware for acceleration of vTPM operations. To use the card in the Xen environment, a process in user space must forward requests between the TPM backend driver and the driver for the card. This is the task of the proxy in Figure 2.

Table 1 describes the properties that can be achieved for TPM functionality based on the three implementation alternatives: hardware TPM, virtual TPM in a trusted virtual machine, and virtual TPM in a secure coprocessor.

6 Sample Application

We ran an existing TPM application to show that our virtual TPM implementation provides correct TPM functionality to virtual machines. As a sample application we chose IBM's open-source Integrity Measurement Architecture (IMA) for the Linux operating system [13].

IMA provides to a remote system verifiable evidence of what software is running on a measured system. It maintains a list of hash values covering all executable content loaded into a system since startup, including application binaries. It brings together measurements made by the BIOS, boot loader and OS, and it offers an interface to retrieve these hash values from a remote system. IMA returns its list of measurements as well as a quote of current PCR values signed by the TPM. The signed quote from the TPM proves the integrity of the measurements. The remote system can then compare the measurements against known values to determine what software was loaded on the measured system.

IMA was originally written to run in a non-virtualized environment, where the Linux kernel has direct access to a hardware TPM. As a test of our vTPM facility, we ran IMA in a Xen virtual machine with access to a vTPM instance.

The complete attestation sequence in our virtualized environment is as follows. The virtual TPM runs as a process in Xen's management virtual machine, domain-0. We boot the system using a trusted boot loader, Trusted GRUB [9, 18]. We measure the Xen hypervisor executable, the domain-0 kernel and initial RAM disk, as well as the initial Xen access control policy [20], and extend a PCR in the hardware TPM with these measurements. The resulting hardware PCR value thus attests to the integrity of the vTPM's trusted computing base (TCB), namely the hypervisor plus the management virtual machine.

When a user virtual machine starts, we measure its kernel image and initial RAM disk, and extend a PCR

Properties	Implementation		
	Hardware TPM	PCIXCC	Trusted VM TPM
HW Tamper	no protection	responsive	no protection
SW Tamper	BIOS or software protected	crypto protected (signed software)	SW protected
TPM TCB	TPM chip, BIOS	tamper responsive environment, signed software	vTPM VM, hypervisor
Platform-Binding	physical (e.g., soldering)	PCI-X bus	logical (H/W TPM or BIOS)
OS-Binding	BIOS or hypervisor (if VM-owned)	hypervisor	hypervisor
Support OS	1	n	n
Cost in USD	1	$>$ 1,000	0

Table 1: Comparison of TPM Implementations

in the virtual TPM with these measurements. This sequence of measurements is part of the setup process of the vTPM instance (see Section 4.4). As the user virtual machine continues to run, the IMA-enhanced kernel in that virtual machine also extends a virtual PCR with measurements of every application that is loaded.

IMA attests to the integrity of both the vTPM TCB and the user virtual machine by returning PCR values from both the hardware TPM and the virtual TPM. We achieve this with our vTPM by projecting the lower PCRs of the hardware TPM (e.g., PCRs 0 through 8) to all virtual TPMs. This means that if a user VM reads one of those PCRs, the vTPM facility actually fetches the value from the hardware TPM. Extending hardware PCRs from user VMs is therefore disabled since these registers are logically owned by the management VM as depicted in Figure 4. Upper PCRs are accessible by user VMs as usual.

Therefore, we have the management VM extend the lower PCRs with measurements of the vTPM TCB. We have the user VM extend the upper PCRs with measurements of the user VM itself. IMA reports then combine lower PCR values, higher PCR values, and the measurement list from both the user VM and the management VM to provide a comprehensive view of the system. To relay the names of applications measured into the hardware TPM, we implemented a small extension to Integrity Measurement Architecture that retrieves this information from the vTPM-hosting domain using the ReportEnvironment command. Other aspects of IMA were left unmodified.

7 Discussion and Future Work

In section 3 we introduced the requirements that an architecture for enabling TPM support in a virtual environment must fulfill. So far we have presented solutions for the first three items and described their implementations. We will now revisit our initial requirements and compare them against our implementation. We then discuss solutions for the remaining items.

7.1 Requirements Revisited

Unmodified TPM Usage Model We provide TPM support by emulating device functionality through a software implementation. We designed and implemented an architecture for the Xen hypervisor that enables us to connect each user domain to its own TPM instance. With our command set extensions we can create as many virtual TPM instances as needed. All existing TPM V 1.2 commands are available to a user domain and the TPM command format remains unchanged.

Strong Virtual Machine to Virtual TPM Association We have shown a design that supports strong virtual machine to virtual TPM association. Components that enforce this need to be implemented in the backend driver such that TPM packets can be routed to the appropriate domain. Also, a table of virtual machine to virtual TPM instance must be maintained.

We introduced new TPM commands for secure migration of TPM state between two virtual TPM implementations. Our migration protocol guarantees TPM uniqueness and prevents attacks on the TPM state information such as alterations to or omission of pieces of state infor-

mation. We based virtual TPM migration on TPM key migration.

In our design we assume the trustworthiness of the destination TPM implementation and the uniqueness of migration identifiers (which can all be verified). HMAC values and migration digests are verified such that our security features can be enforced. It is important for virtual TPM migration that the asymmetric storage key is only migrated into a trusted virtual TPM. A possible solution for determining the trustworthiness of a destination TPM is to require a certificate of the destination TPM's storage key where the signature key is an externally certified Attestation Identity Key (AIK).

Strong Association of the Virtual TPM with the Underlying TCB Using an existing attestation architecture for Linux, we showed how a strong association between a virtual TPM instance and the hardware root of trust (hardware TPM) of the platform can be established.

Our architecture and virtual TPM have been designed such that a challenger not only sees measurements taken inside the virtual machine OS, but can establish trust into the virtualization environment, including the boot process, hypervisor and the operating system that is hosting the virtual TPM.

7.2 Trust Establishment

We have so far reported several solutions from our experience providing TPM support to virtual machines. However, there are a number of issues that still need to be investigated. Whereas other devices can be satisfactorily virtualized through device emulation, more support is needed in our case, particularly on the treatment of security credentials such as TPM keys and associated certificates.

From our experience we can claim that it is easy to create an endorsement key for a virtual TPM instance, but some questions arise around the certificate that needs to be issued:

- Who would provide a certificate for an endorsement key of a virtual TPM?
- What guarantees would this certificate stand for?

A certificate authority, i.e., a privacy CA, bases its decision to certify an AIK of a TPM on the certificate of the EK that a manufacturer provides along with the device. This certificate vouches for the TPM being a hardware device and that it is firmly attached to the motherboard of the computer. Since the availability of an EK certificate plays this important role in receiving a certificate for AIKs, the EK certificate should also be available to a virtual TPM instance even if it does not stand for the same security guarantees as those provided by a hardware TPM. However, virtual TPMs can be dynamically created whenever a new VM is created, and therefore requests for EK certificates can become more frequent and their management becomes much more dynamic.

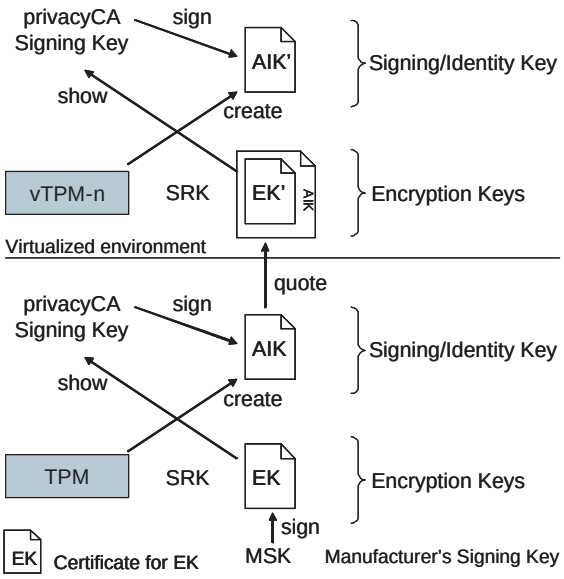

Figure 6: Certification of Endorsement Key using an AIK

We have found several solutions for the creation of EK certificates, each having advantages and disadvantages. We discuss those solutions below and, after looking at virtual TPM migration, provide a comparison between them.

1. Our first solution creates a certificate chain by connecting the certificate issued for the EK of a virtual TPM instance to that of an AIK of the hardware TPM. Figure 6 depicts this relationship. It shows that a privacy CA issues certificates for AIKs of a virtual TPM based on the certificate of its endorsement key EK'. The advantage of this scheme is that we have preserved the normal procedure of acquiring an AIK' certificate by submitting the certificate of EK' to a privacy CA for evaluation.

   In this and the following solutions we are using an (attestation) identity key and the TPM's Quote command to issue a signature over the current state of PCRs and a user-provided 160bit number. We provide as 160bit number the SHA1 hash of the certificate contents of the EK'. The resulting signature ties this EK' certificate and the virtual TPM instance to the underlying platform. In addition to the PCRs, the certificate can also contain the measurement list of the VM environment to enable the establishment

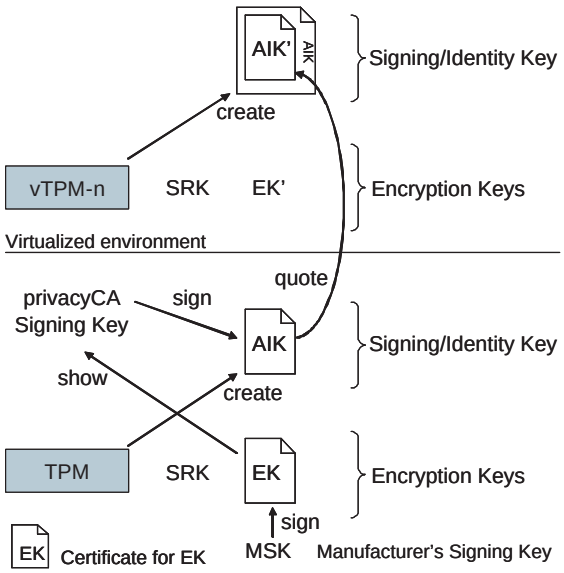

Figure 7: Certification of AIK' using AIK

of trust into the certificate-signing process [21].

2. Our second solution, depicted in Figure 7, does not use a certificate for a virtual EK', but issues certificates for virtual TPM AIKs based on an AIK issued for the hardware TPM. The resulting certificate chain ties the virtual TPM's AIK' to the AIK of the hardware TPM, and thus to the hosting system. The advantage of this solution is that once an AIK has been issued for the hardware TPM, virtual TPM AIKs for guest VMs can also be quickly certified. Through the chain, a link is established to the hardware-TPM platform. The disadvantage of this solution is that it requires changes to the normal procedure of acquiring an AIK certificate from a certificate authority.

3. A third solution relies on a local authority to issue the certificate for the virtual TPM instance's EKs. The benefit of this procedure compared to the previous ones is that the resulting virtual TPM's EK certificate is not tied to the hardware platform, since no certificate chain is established to credentials of the hardware TPM. A local EK certificate authority can also be used for hardware TPMs if they are not equipped with a platform certificate, as is often the case today. Beyond this, this third solution offers the advantage over the second one of not changing the procedure for acquiring certificates for AIKs.

4. A fourth solution is based on a secure coprocessor that replaces the hardware TPM used in the other solutions to provide a hardware root of trust. The manufacturer links the endorsement key certificate to the secure coprocessor certificate and enables remote parties to establish security properties for the virtual TPM runtime environment as described in Section 5.2.
Starting with this manufacturer-provided EK certificate, all the previously described solutions for creating certificate chains for virtual TPM instances can be applied.

Depending on which solution for issuing certificates is chosen, the migration of a virtual TPM to another platform can affect the validity of certified TPM keys. If, for example, AIKs have been certified based on an EK that was previously tied to the hardware platform through a chain, as we have shown in the first two solutions, the AIKs must be invalidated once the VM is resumed on the target platform since the link to the old platform has now been broken. Our third solution avoids this problem, because it does not establish a firm link with the VM-hosting platform.

What makes the realization of an architecture based on certificate chains more difficult is that AIKs and certificates may be maintained by programs inside the operating system. The TSS stack must be aware of migration and destroy AIKs once the OS resumes on the target platform. After the AIKs have been recreated, they must be certified for usage on the new platform. Applications must also be made aware of the new certificates and remove old ones from memory.

Another problem can be certificates that clients examine while a VM is migrating to a new platform. Based on the evaluation of the certificate, the client may treat the peer system as trusted, although it is now running inside a new environment. For practical purposes, a migrating partition should offer a subscription service for any party interested in learning about migration. Notifications can be sent that inform subscribers that migration has happened and trigger a reestablishment of trust. We do not currently offer such a service.

Another question that arises due to virtual machine migration is: When a virtual machine is migrated from one system to another, should all virtual machine environments' measurements be recorded and a history be established? We feel the answer to this question is "yes", but we have not yet explored efficient ways to support this capability.

Table 2 gives an overview of the properties of the first three of our proposed solutions. A decision about which method to implement for certifying EKs must weigh the advantages and disadvantages of each solution. If a

Method / Support	AIK certifies EK'	AIK certifies AIK'	Local authority certifies EK'
Needs a CA	privacy CA	No	local and privacy CA
Establishes link to hosting platform	Yes, EK' is linked	Yes, AIK' is linked	No
Needs AIKs and certificates to be invalidated after migration	Yes; must also invalidate EK	Yes	No
How external party verifies AIK' certificate	Need to know public signing key of privacy CA	Need to know public AIK used for quoting	Need to know public signing key of privacy CA
Software in VM needs to be aware of how to have AIK' certified	No - contact privacy CA as normal	Yes. AIK of parent environment is used to have AIK' certified.	No - contact privacy CA as normal
Credentials a CA must interpret to issue AIK certificate	AIK: EK certificate AIK': TPM-quote for EK' and associated public AIK	AIK: EK certificate AIK': not involved	AIK: EK certificate AIK': EK' certificate and (local) CA's public signing key

Table 2: Comparison of Methods to Issue Certificates for AIKs

strong connection between the virtual TPM and the hardware TPM is desired, then one of solution 1,2 or 4 should be implemented. However, it will be necessary in this case to invalidate the chained certificates and keys after migration in order to reestablish a chain to the new hardware root of trust. In that respect our second solution offers better support for a dynamic environment, since here only the AIKs of the virtualized environment need to be recreated and certified. The first solution would eventually have to place the EK' certificate on a revocation list and create a new EK.

If a local certification process has already been established to certify EKs for hardware TPMs, this or a similar process can be applied to EKs of virtual TPMs as well. It would simplify an implementation for virtual TPM migration with its VM since in this case there is no link to the parent environment. Therefore migration would not break any certificate chains. It can be regarded as the least complicated solution, since neither side of the attestation procedure would have to forget about credentials that applied to the pre-migration environment.

8 Related Work

The Xen open-source repository [27] contains a limited virtual TPM implementation comprised of combined contributions by Intel Corporation and the authors of this paper. Our contributions to Xen so far include the virtual TPM driver pair (front- and back-end drivers), hotplug scripts, and changes to Xen's management tools. We kept this infrastructure modular so that different realizations of virtual TPMs can work with it. The virtual TPM design and implementation presented in this paper adds the following to what is currently available in Xen: support for migrating a vTPM instance alongside its associated virtual machine, support for attestation of the complete vTPM environment along with the contents of a virtual machine, and an entirely separate software implementation of the TPM specification. In addition, the virtual TPM now in Xen is a partial implementation based on version 1.1 of the TPM specification, while we have updated our virtual TPM to be a complete implementation of version 1.2.

Previous research in the area of trusted computing examined how data that is protected (sealed) by a hardware TPM can be moved to another platform. Kuehn et al. [17]

proposed a protocol for migrating the key-related hardware TPM security state from one hardware platform to another involving a separate *TPM Migration Authority* (TMA). Our protocol differs from the one presented there in many significant ways. Most notably, we migrate the complete virtual TPM state, we do not require a third party for migration, we maintain associations of virtual TPMs to their VMs and the operating system, and we can seamlessly integrate our protocol into the automated VM migration process. In addition, the extensions we introduce to the TPM standard do not require changes to existing commands and semantics. Similar to their concern about security of the destination TPM, we have pointed out that secure migration relies on a decision process that determines the safety of migrating a key pair to another TPM based on trust in that other TPM implementation.

The Terra project [7] investigated trusted virtual machine monitors. They developed a prototype based on VMWare's GSX server product that performs attestation of virtual machines and applications launched therein. Their publications recognize the availability of TPM 1.1b, but do not describe the design of a virtual TPM to run their attestation scheme against. Terra could use something like our vTPM facility to make a virtual TPM instance available to each of their virtual machines.

9 Conclusions

We have designed and implemented a system that provides trusted computing functionality to every virtual machine on a virtualized hardware platform. We virtualized the Trusted Platform Module by extending the standard TPM command set to support vTPM lifecycle management and enable trust establishment in the virtualized environment. We added support for secure vTPM migration while maintaining a strong association between a vTPM instance and its associated VM.

We uncovered the most important difficulties that arise when virtualizing the TPM. Whereas usually virtualization of hardware devices can be achieved through software emulation, we have demonstrated that this is not sufficient in the case of the TPM. Certificates that may exist for hardware TPMs and vouch for strong security properties need to be issued for virtual TPM instances' endorsement keys . These certificates can naturally not represent the same properties for a virtual TPM process running in user space. Trust chains that are usually owned by a single OS now *pass through* a hierarchy of virtual machines. Virtual TPM migration can create further problems if certificate chains that have been established break or trust must be reestablished.

We virtualized the Trusted Platform Module by making all low-level TPM 1.2 commands available to every virtual machine. Applications that don't handle certificates related to TPM-generated keys or do not deal with the concept of trust can remain unchanged. Applications challenging a virtual machine or those following certificate chains, like for example a privacy CA, must be aware of the modifications that were necessary for the virtualized environment. Those modifications include certificate chains that consist of different types of certificates issued through special signing mechanisms of the virtual TPM, or certificates provided by the manufacturer of the device or those issued through a certificate authority such as a privacy CA. Applications that have been adapted to work in the virtualized environment will be backwards compatible with platforms using a singleton hardware TPM.

Our proposed architecture for virtualizing the TPM is a major building block for establishing trust in virtualized environments. For example, Trusted Virtual Data Centers [16] create distributed virtual domains offering strong enterprise-level security guarantees in hosted data center environments. In such an environment, virtual TPMs help to establish trust in strong domain security guarantees through their remote attestation and sealing capabilities.

Acknowledgments

We would like to thank our shepherd, Peter Chen, and the anonymous reviewers for their valuable comments. We also thank the Xen community for their feedback on the vTPM work.

References

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, 2005.

[3] Common Criteria. Trusted Computing Group (TCG) Personal Computer (PC) Specific Trusted Building Block (TBB) Protection Profile and TCG PC Specific TBB With Maintenance Protection Profile, July 2004.

[4] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[5] R. Figueiredo, P. A. Dinda, and J. Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(5):28–31, 2005.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the OASIS ASPLOS Workshop*, 2004.

[7] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 193–206, 2003.

[8] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer Magazine*, 7(6):34–45, 1974.

[9] Applied Data Security Group. What is TrustedGRUB, http://www.prosec.ruhr-uni-bochum.de/trusted_grub.html.

[10] Trusted Computing Group. TCG TPM Specification Version 1.2 - Part 1 Design Principles, 2005.

[11] Trusted Computing Group. TCG TPM Specification Version 1.2 - Part 3 Commands, 2005.

[12] IBM. IBM eServer x366. http://www-03.ibm.com/servers/eserver/xseries/x366.html.

[13] IBM. Integrity Measurement Architecture for Linux. http://www.sourceforge.net/projects/linux-ima.

[14] IBM. PHYP: Converged POWER Hypervisor Firmware for pSeries and iSeries. http://www-1.ibm.com/servers/enable/site/peducation/abstracts/abs_2bb2.html.

[15] IBM. Secure Coprocessing. http://www.research.ibm.com/secure_systems_department/projects/scop/index.html.

[16] IBM. Trusted Virtual Data Center. http://domino.research.ibm.com/comm/research_projects.nsf/pages/ssd_trustedvirtualdatacenter.index.html.

[17] U. Kühn, K. Kursawe, S. Lucks, A. Sadeghi, and C. Stüble. Secure data management in trusted computing. In *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2005)*, 2005.

[18] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, S. Yoshihama, and T. Nakamura. Trusted platform on demand. Technical Report RT0564, IBM, February 2004.

[19] National Institute of Standards and Technology. Secure Hash Standard (SHA-1). Federal Information Processing Standards Publication 180-1, 1993.

[20] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Griffin, and L. van Doorn. Building a MAC-based security architecture for the Xen opensource hypervisor. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2005.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.

[22] The Xen Team. Xen Interface Manual - Xen v3.0 for x86.

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

[24] Trusted Computing Group. TCG PC Specific Implementation Specification, 2003.

[25] Trusted Computing Group. TCG Software Stack (TSS) Specification - Version 1.10 Golden, 2003.

[26] VMware, Inc. http://www.vmware.com.

[27] Xensource. Xen Open-Source Hypervisor. http://www.xensource.com/products/downloads.