The Second International Conference on Emerging Security Information, Systems and Technologies

Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques

Frederic Stumpf
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
stumpf@sec.informatik.tu-darmstadt.de

Claudia Eckert
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
eckert@sec.informatik.tu-darmstadt.de

***Abstract*—We present the design of a trusted platform module (TPM) that supports hardware-based virtualization techniques. Our approach enables multiple virtual machines to use the complete power of a hardware TPM by providing for every virtual machine (VM) the illusion that it has its own hardware TPM. For this purpose, we introduce an additional privilege level that is only used by a virtual machine monitor to issue management commands, such as scheduling commands, to the TPM. Based on a TPM Control Structure, we can ensure that state information of a virtual machine's TPM cannot corrupt the TPM state of another VM. Our approach uses recent developments in the virtualization technology of processor architectures.**

I. Introduction

The Trusted Computing Group (TCG) [1] is a non-profit organization that defines open standards for hardware-enabled trusted computing and security technologies. A core component of the specifications issued by the TCG is the Trusted Platform Module (TPM) [2], that can be viewed as functionally equivalent to a high-end smart card. The TPM can be used to significantly enhance system security, since it offers a means of storing cryptographic keys in a protected manner and of establishing trust relationships between remote systems [3].

However, [4], [5], [6], [7] show that using all mechanisms of the TPM in practice requires a system that is based on virtualization [8]. This is especially true if a system wants to demonstrate to a remote party that it is in a trusted state by using the attestation facilities of a TPM.

Virtualization technologies are actually getting increased interest from both industry and academia because they offer an alternative means of hardening an operating system and thus of increasing system security [9], [10]. This trend is also reflected in the current hardware architectures. Intel [11], [12] and AMD [13] have added virtualization support to their processor architectures in order to accommodate emerging developments that require the support of multiple virtual machines (VMs) on a single entity. This is not only of interest in the server market, where the execution of multiple commodity operating systems should be supported on a single machine, but also in the client area, where there is a need to completely increase system security and reliability [14].

Unfortunately, the TPM was never designed to be used in virtual environments, and is thus not capable of being used in a system that is based on virtualization. This restriction results from the fact that the TPM holds state specific data. If two different VMs are accessing the same TPM, one VM could change the state of the TPM, which would also change the state of the other virtual machine's TPM. To overcome this restriction, Berger et al. [5] propose to virtualize the hardware TPM, by equipping every VM with its own software TPM. This software TPM only uses the underlying hardware TPM for certain operations. However, this approach does not provide the same security level, since the TPM is completely implemented in software and thus cryptographic secrets are temporarily not protected by hardware measures.

To provide life-time protection of cryptographic secrets and the possibility of using the functionalities of a hardware-based TPM inside the VMs, we propose to use a TPM that is capable of supporting virtualization with hardware measures. We present the design of such a TPM and show how the interactions between the VMs and the TPM could be realized. The approach presented in this paper utilizes hardware-based virtualization techniques as provided through Intel's VT processor architecture.

The remainder of this paper is organized as follows: we first look in Section II at other work that is related to our proposal. We then extract requirements for a hardware-based virtual TPM and present the main idea of our approach in Section III. In Section IV, we explain our architecture and the components that are involved in our approach. Section V shows how we enable the different VMs to directly use the underlying hardware TPM. Section VI presents the protocol for securely migrating a TPM context to another VM. In Section VII, we explain how the endorsement credentials for the different TPM contexts are handled and how the TPM management commands are realized. We conclude with Section VIII.

II. Related Work

Berger et al. [5] illustrate how to virtualize a TPM and present a driver-pair that utilizes these concepts. To virtualize a TPM, they propose to create a software TPM instance for every VM. This software TPM is then used by the VM as a full-fledged TPM. The software TPM is able to use the underlying hardware TPM for certain operations, e.g., sealing the storage of the software TPM; however, it does not provide lifetime protection of secrets. Therefore, this approach does

978-0-7695-3329-2/08 $25.00 © 2008 IEEE
DOI 10.1109/SECURWARE.2008.23

not provide the same security capability as a hardware TPM, since cryptographic keys are only protected through software measures. In addition, software TPMs cannot provide the same protection level as a hardware TPM if they are evaluated according to Common Criteria.

Microsoft's Next Generation Secure Computing Base *NGSCB* [15], [10] is an approach that aims at establishing a small trusted computing base, while satisfying the need for open mass-market operating systems. The approach is also based on a microkernel, called *isolation kernel*, which establishes two different execution environments, i.e, VMs, with different trust-levels. NGSCB highly depends on hardware virtualization technology as well as on trusted computing technology. Since in this approach only one VM can access the TPM, virtualizing the TPM is not necessary. However, NGSCB therefore does not support multiple VMs with access to the TPM, leaving NGSCB is un-applicable when a need for multiple VMs occurs.

Current research findings have clearly identified a need for using virtualization techniques to reliably use trusted computing technologies. In this context, two major approaches have taken shape to support trusted computing. One approach builds on the L4 microkernel [16] and the other on the Xen hypervisor [9]. These approaches are mainly bundled in the EMCSB-project [17] and the Open-TC project [18], where both projects aim at building open and trustworthy computing platforms based on trusted computing technology. As in NGSCB, a software TPM is not necessarily required in the L4 micorkernel approach, since this approach only provides one TPM-enhanced VM. However, in that case, the approach is not capable of supporting multiple TPM-enhanced VMs.

Intel and AMD have recently introduced the Trusted Execution Technology (TXT) [12], [19] and the Secure Virtual Machine technology [13], respectively. Both architectures extend the processor instruction set with a number of additional special purpose instructions, that directly communicate with a TPM. In addition, these architectures implement virtualization technology and are thus capable of supporting different efficient VMs with hardware measures.

Our approach uses functionalities of the Intel VT-X/I architecture. This architecture augments the x86-processor architecture with two new forms of CPU operation: *VMX root operation*, in which the VMM runs, and *VMX non-root operation*, in which the guest systems run. Both operations support the privilege set of the x86 architecture. The VMM is then typically run in CPU ring 0 of the VMX root mode and the guest system is run in CPU ring 0 of the VMX non-root mode. The processor additionally provides a special purpose structure called *virtual machine Control Structure* (VMCS). In this structure, state information of the virtual machine is stored and loaded into the processor if a state transition is performed. A state transition to a VM is called `vmentry` and the transition back to the VMM is called `vmexit`. But in contrast to directly integrating a TPM into the processor core, both architectures use a TPM that is attached to the Low Pin Count-bus. Therefore, both approaches do not provide any means of virtualizing a TPM.

III. Virtualizing the TPM

One design criteria of currently available TPMs was that they should be easily attachable to a mainboard of a PC. Attaching a TPM to an Low Pin Count (LPC) bus seemed to be a very convenient solution; however, this approach is very problematic as it does not enable the establishment of trust relationships using the TPM in a system that is based on virtualization [4]. One solution for this problem is to use software TPMs [5], [4] that only use the underlying hardware TPM for certain operations. This approach allows the virtualization of the TPM and thus the establishment of trust in a system that is based on virtualization. On the downside, this approach does not satisfy the same security needs as a hardware TPM, since the TPM is implemented in software. In addition, a software TPM cannot provide the same protection level if it is evaluated according to Common Criteria, because software TPMs does not provide mechanisms for preventing unauthorized access to protected data, such as active shields or active security sensors. However, a high evaluation may be necessary if VMs are using a TPM and rely on trusted computing technology. To enable a VM to use the full functionality of a hardware TPM without accepting security restrictions, we propose a *multi-context TPM* that is completely realized in hardware.

A. Requirements

In this section we present the design goals for using a TPM in virtual environments. These goals apply to a multi-context TPM as well as a software TPM. The approach presented in this paper is adapted to fulfill the presented design goals. These goals are:

a) Performance: Performance is a measure for the virtualization overhead. This goal states that the overhead from using a TPM in a VM should be negligible compared to directly executing commands on the TPM. The VM should therefore be able to execute TPM commands at nearly the same speed as when these commands are used on a non-virtualized machine.

b) Compatibility: It should be possible to execute TPM command code in a VM without modifying the code or adapting it to the virtualized environment. Therefore, we explicitly forbid using paravirtualization techniques [20].

c) Simplicity: A fault in a VMM can cause a failure in all VMs which could result in a crashing VM. A VMM that provides abstraction and sharing of a TPM should therefore be as simple as possible [21].

d) Security: A TPM that is used in a VM should provide the same security properties as if the TPM would be accessed natively.

e) Minimal modifications to the specification: If the specifications by the TCG must be modified, these modifications should be as slight as possible, leaving the modifications and the specification as compatible as possible.

B. Our approach

The main challenges to providing hardware enhanced virtualization of TPMs is determining how to handle TPM data that is specific for a certain platform, e.g., data that is specific for the physical machine. Such data includes the owner-password of the TPM, the PCRs, the Storage-Root-Key (SRK) and the EK. This data cannot be shared across all VMs, because if it were, a VM could modify this data, which would result in a TPM state change and thus influence the TPM state of the other VMs.

It is thus necessary to provide every VM with its own instance of a full-fledged hardware TPM, including its own owner-password, PCRs, SRKs and EKs. Since it should be possible to use the hardware TPM for a theoretical unlimited number of VMs, the multi-context TPM should be flexible and not associate one specific TPM, non-volatile memory region for every VM.

We propose that the multi-context TPM operates on a Control Structure that is loaded into the TPM each time a particular VM operates on its TPM. A VMM is responsible for providing an abstract interface to the underlying hardware TPM and for isolating the different TPM instances. It also performs scheduling operations and is responsible for assigning an unique ID to every VM. This unique ID refers to a specific TPM context. If a VM wants to issue a command to the hardware TPM, the VMM loads the corresponding TPM Control Structure (TPMCS) into the TPM (if not already loaded) and the TPM then operates on this structure. The TPM Control Structure is a data structure that encapsulates all the information needed to capture the state of a TPM or to resume a TPM. This approach enables the direct execution of the TPM instructions from a VM on a TPM, and is thus very efficient. To provide a hardware protection mechanism, we introduce another privilege level to the TPM. A virtual machine runs in a lower TPM privilege level and thus can only operate on its own TPM Control Structure. Operations on other TPMCS can only be done by management commands issued by a VMM. The issuing of such a command by the VM must be intercepted and thus results in a controlled context switch to the VMM. We will discuss this mechanism in detail in Section V.

IV. TPM Architecture

The layout of our multi-context TPM is shown in Figure 1. Despite the components of a generic TPM, it also provides a second non-volatile storage in which the active TPM Control Structure is loaded. The data of the TPM Control Structure can only be loaded and unloaded into the TPM by TPM management commands. When a TPM Control Structure is unloaded and written back to background storage, it is always protected by secrets that are stored in the root-data structure. Both the root-data structure and the TPM Control Structure hold TPM specific data that is expected to never leave the TPM, as specified by the TCG. This includes Endorsement keys (EKs), Attestation Identity Keys (AIKs), etc. Only TPM commands that are issued by a VMM can operate on the root-data structure.

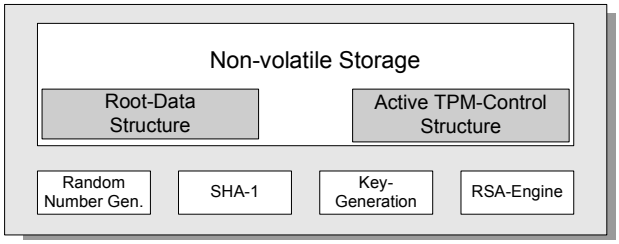

Fig. 1. Layout of a multi-context TPM

We assume that our multi-context TPM is either integrated on the CPU or on a fast bus, and thus has a direct connection to the CPU, leaving it essentially free of hard speed constraints compared to the LPC bus.

A. TPM Protection Rings

To provide the TPM with a hardware-based protection architecture and to isolate one VM TPM context from another, we introduce hierarchical protection domains (protection rings) into the TPM. These protection rings are shown in Figure 2 and distinguishes two different TPM modes of execution. For this purpose, we introduce a 1-bit non-volatile control register (CR), which the actual TPM state refers to. Every time a context switch occurs, i.e., a controlled state transition from non-privilege TPM mode to the privilege TPM mode and vice versa, the TPM control register is set appropriately. The Figure also shows which x86 CPU modes, and thus which software (VMM or guest-OS), is executed in which ring. If our TPM is directly integrated inside the CPU, and is therefore able to use the protection rings of the CPU, the integration of the CR inside the TPM could be omitted. Nevertheless, our proposed architecture requires a direct interaction with the CPU in order to satisfy the necessary protection domains and to ensure that context switches are reliably enforced. Hence, we require a strong collaboration between CPU and TPM as we will see in the following sections.

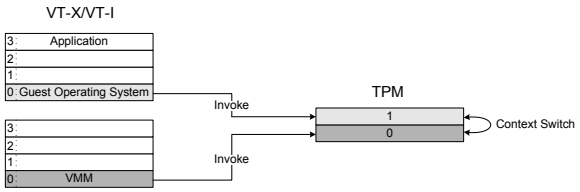

Fig. 2. Privilege level of a multi-context TPM

The VMM ought to be run in the privilege VMX root-operation of the CPU as well as in the TPM privilege mode. The VMM runs on a higher privilege level of the TPM and is thus able to manage TPM state transitions. However, under the assumption that the authorization data (e.g. owner password) of the TPM are kept secure inside the VM, the VMM cannot inspect the communication between VM and TPM. All communication between TPM and the software

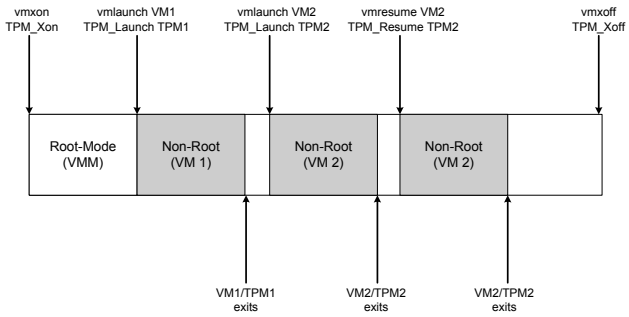

Fig. 3. Transition between different TPM contexts

stack is encrypted with a session-key that is derived from the authorization data (Cf. [22], pp. 60).

B. TPM back-end device driver

The TPM back-end device driver is integrated into the VMM and therefore runs in ring 0 of the CPU's VMX-root mode. Its main purposes are isolating the different TPM interfaces and scheduling the TPM commands invoked by the VM. It also maintains a data set including the unique IDs of each TPM context as well as their associations to the VMs. Note that this TPM back-end device driver does not need to implement the full software stack of the TPM, since conventional operations on the TPM are issued by VMs that implement their own software stacks.

Figure 3 shows a typical sequence of operations inside a virtualized TPM environment. The white areas of the Figure show the operations of a VMM that set up the environment for every VM and its corresponding TPM context. The gray areas represent two different VMs with their corresponding TPM contexts. The transition between VMM and VM are invoked by special instructions, which can only be executed in the TPM privilege mode.

Every TPM that has been turned on in normal mode can be made to enter the virtualization mode by executing the `TPM_Xon` operation. The transition of the TPM contexts needs to be synchronized with the transitions between the different modes of execution on the Intel VT-X/I architecture, in order to support direct native execution on the TPM. This property is discussed in detail in Section V.

The VMM assigns for every TPM context, as well as for every VM instance, an execution time in which the VM is allowed to execute operations on the TPM. This execution time not only includes the time a VM is allowed to use the TPM, but also how long the VM is allowed to use the CPU. Determining the exact execution time is task of a scheduling algorithm, such as round-robin. Thus, all VM does not necessarily receive the same execution time to execute instructions on the underlying ressources. If this execution time has passed, the VMM regains control of the TPM and assigns the next context of the TPM. To keep the VMM from losing control of the TPM, the VMM must set an *interval timer*, that states how long the VM is allowed to use the TPM and the underlying processor. This interval timer is set by the VMM before passing control to a VM. It counts down clock circles and if it reaches zero, triggers an interrupt. The interrupt then passes the control back to the VMM.

C. TPM Control Structure

A transition from one TPM context to another is controlled by the VMM through a TPM Control Structure (TPMCS). The VMM associates to every TPM context its own TPM Control Structure. This structure holds all state specific TPM data as shown in Figure 4 and only the TPM can operate on this structure. Each time the VMM closes a TPM context and spawns a new context, the old Control Structure is saved on the background storage and the new Control Structure is loaded into the TPM. Since this Control Structure holds sensitive TPM data, cryptographic measures must be in place to prevent unauthorized modifying of a Control Structure.

To protect this TPM structure from unauthorized modifications, we propose using the Storage Root Key of the TPM to seal this structure to the current platform configuration of the system. This SRK is stored inside the TPM root-structure and is only accessible by the TPM in the root-mode of the TPM. Sealing the structure to a set of platform configuration registers is necessary for ensuring that the VMM is in its initial state and thus trusted. In contrast to the SRK of the VM, the SRK of the root-mode will never leave the TPM.

Fields of the TPM Control Structure
Storage Root Key (SRK)
PCRs [16..23]
Attestation Identity Keys (AIK)
Endorsement Key (EK)
Endorsement Credential
Monotonic Counter Values
Values of the non-volatile storage
Delegation Tables
TPM context data
DAA TPM specific secret (f)
TPM_PERMANENT_FLAGS/DATA
TPM_STCLEAR_FLAGS/DATA
Authorization data

Fig. 4. TPM Control Structure

The TPM also possesses a number of special purpose registers, such as the tick counter or the PCRs. Tick counter and the lower values of the PCRs are special, since both should be consistent in all VMs. We suggest that the tick counter and the PCRs [0..15], which hold the data from the bootstrap procedure and the VMM's integrity, be stored in a special region of the TPM. Every TPM context should be able to read the values stored in these registers. Writing to this registers is only possible if the TPM is in the privilege mode and its CR is set to 0. To enable a VM to attest its configuration, the TPM computes the signature on PCRs [0..15] and PCRs [16..23] using one AIK stored inside the loaded TPM Control Structure.

D. Extended Instruction Set

To realize the management features of the TPM, we extend the specification to a number of additional TPM commands. These commands manage the TPM state and are used to control the different TPM contexts. All commands are executable only by the VMM, since they must be executed in ring 0 of the TPM. If one such command is executed in ring 1 of the TPM, they trap into the VMM, where a dispatcher emulates the instruction. In the remainder of this paper, these instructions are referred to as sensitive instructions. The TPM also supports a number of instructions that allow a TPM context of being migrated to another TPM. We will explain these instructions in detail in the remainder of this work.

`TPM_Xon, TPM_Xoff` enables/disables the second privilege level of the TPM. `TPM_Launch` creates and launches a new TPM context and an empty TPMCS. `TPM_Resume` loads an existing TPMCS into the TPM and launches the corresponding TPM contexts. `TPM_Exit` saves an existing TPMCS and seals it to certain PCRs. `TPM_Clear` deletes an existing TPMCS and the corresponding TPM context. `TPM_Migrate` migrates an existing TPMCS from a source TPM to a destination TPM. `TPM_InitializeMigration` initializes the migration procedure on the source TPM. `TPM_InitializeImport` initializes the migration procedure on the destination TPM. `TPM_Import` imports a migrated TPMCS into the destination TPM.

These instructions also control which TPM context and which corresponding VM will receive the underlying hardware TPM thus which VM is allowed to operate on the TPM.

V. Direct Native Execution

For virtualizing a resource, there exist a number of different techniques. A processor is typically *shared* by a number of processes or VMs and every process is allowed to use the resource for a particular time period. This technique is also adapted to our approach. However, other techniques also exist, for example, *partitioning* which is normally done for background storage (disks); and *emulation*, which is for example used in the software TPM approach [5] or if a processor is completely emulated [23].

The main advantage of the sharing technique is that it enables *direct native execution* and thus the use of efficient virtualization, as stated by Popek and Goldberg's first Theorem [24]. To enable an efficient virtualizable TPM, we adapt the TPM so that it is possible to execute nearly all instructions directly on the TPM and thus to satisfy Popek and Goldberg's first Theorem.

In order to achieve direct native execution, it is absolutely necessary that sensitive instructions trap into the VMM. Sensitive instructions are instructions which can result in an illegal TPM state change. If such a sensitive instruction is executed by a VM it must trap into the VMM where a dispatcher routine emulates the instruction. However, since the TPM does not possess its own program counter, it cannot autonomously trap into the VMM if it encounters a sensitive instruction. This property is similar to sensitive x86 processor instructions such as `POPF`, which do not perform a trap if they are executed in the non-privilege processor mode [25].

Since TPM state transitions and VM state transitions are synchronized and controlled by the VMM, the VM, which is actually the owner of the processor also occupies the TPM. Therefore, only the VM that currently has execution time on the processor can issue commands to the TPM. Consequently, we only have to consider the case in which a sensitive instruction is executed in ring 1, such as `TPM_Exit`.

A. Handling sensitive instruction

If a sensitive instruction, such as `TPM_Exit`, is executed in ring 1, the TPM must trap into the VMM. Sensitive commands must be executable only by the VMM, otherwise, a VM could overwrite values of a TPM Control Structure which belongs to another TPM context. Since a VMM cannot inspect all commands sent by a VM, the TPM must decide whether or not this command is to be executed. If the TPM receives a management command, it should check internally whether the TPM CR is in ring 0. If not, the TPM should jump to the dispatcher routine running inside the VMM. Note that either the VMM or the VM could issue a management command while the TPM's privilege level is set to 1. This is due to the fact that the status register of the CPU and the TPM are not necessarily synchronized. Thus, it could happen, that the CPU is operating in VMX-root mode, while the CR of the TPM is still set to 1. In that case, the CR of the TPM must first be set to 0 before the TPM can execute a VMM issued management command.

Algorithm 1: Handling sensitive instructions

```
TPM receives TPM_Exit ;
if CR0 == 0 then
    Execute TPM_Exit;
else
    CR0 = 0 ;
    Execute INTR ;
    Interrupt causes vmexit ;
    CPU loads exit information from VMCS ;
    CPU sets PC to VMM entry point address ;
    VMM executes/emulates TPM_Exit ;
    CR0 = 1 ;
    vmentry (VMCS) ;
```

Since the TPM cannot directly modify the program counter of the CPU if it receives a privilege command, an exception must be caused that allows the CPU to jump into the correct dispatcher routine inside the VMM. We propose to use *interrupt requests* as they are typically used by other PCI devices through the `INTR` bus signal [26]. If the TPM is integrated inside the CPU, the TPM might be able to directly modify the program counter and thus jump directly into the dispatcher routine of the VMM.

Algorithm 1 shows the steps performed by the CPU and the TPM respectively. If an exception happens, the TPM

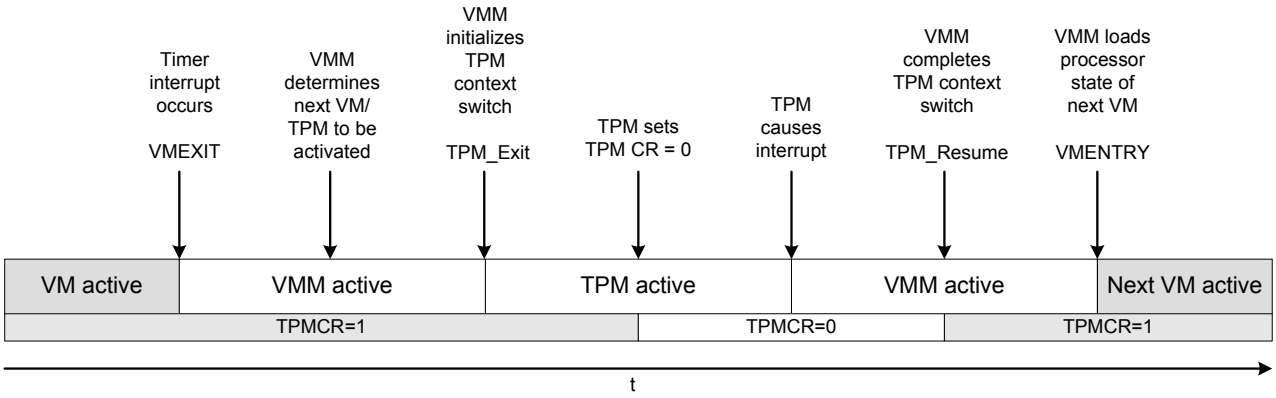

Fig. 5. Workflow of retiring one VM/TPM and activating the next VM/TPM

signals an interrupt by emitting the `INTR` bus signal. The processor then reads from the system bus the interrupt vector number provided by an external interrupt controller. This vector number signals to the CPU which interrupt is caused. Based on the information given inside the VMCS held by the VT-I/X CPU, the CPU performs a `vmexit` and jumps into the corresponding dispatcher routine of the VMM. For this purpose, the VMCS of the VT-I/X architecture must hold the information where the TPM dispatcher routine is located in memory.

B. Scheduling the TPM

The VMM is responsible for the transitions between different VM contexts. This approach is analogous to scheduling VMs. However, the problem is that if the TPM is not directly integrated into the CPU, the privilege level of the TPM does not directly depend on the privilege level of the CPU. Thus, a `vmexit` does not directly set the control register of the TPM to zero. Therefore, a controlled state transition must be passed to the TPM, which then issues an interrupt that jumps to a concrete entry point of the VMM.

Figure 5 shows the actions performed by the VMM in retiring one VM and activating the next VM. The figure also shows the value of the TPM control register. The CPU performs a `vmexit` that is caused by the interval timer and sets the PC to a specific entry point of the VMM. This controlled state transition also stores the actual state of the VM inside the VMCS. The VMM then determines which VM is next in line to use the processor. The VMM then closes the TPM context by sending a `TPM_Exit` command to the TPM. The TPM resets its control register to zero, issues an interrupt and delivers the corresponding interrupt vector number using an interrupt controller to the CPU [26]. Based on the actual interrupt descriptor table (IDT), the CPU again jumps to a specific entry point of the VMM, where the instruction is again sent to the TPM. The VMM then resumes the next TPM context by sending the `TPM_Launch` command together with the stored and encrypted TPMCS to the TPM. Afterwards, the VMM resets the interrupt timer and relinquishes control to the VM by loading the VM state information into the CPU (`vmentry`).

VI. Secure TPM context Migration

VMs can be migrated to other platforms. Since a VM might have stored cryptographic keys inside our multi-context TPM, we have to consider how a complete TPM context is securely migrated to another TPM. Basically, all state information of a TPM context is stored inside the TPM Control Structure. However, since this Control Structure is bound to a specific TPM, it cannot be transferred directly.

To prevent a TPMCS's being migrated to multiple contexts or an old TPMCS's being again replayed into the system, we propose the use of the monotonic counter of the TPM to synchronize all existing TPM contexts with the root-structure of the TPM. For this purpose, each TPMCS and the root-TPM structure hold a register in which both are incremented each time a TPMCS is migrated. We refer to this register as the *migration counter*. Before a TPMCS can be migrated, the TPM internally verifies whether both migration counters have the same value. If an already migrated TPMCS is loaded again in the TPM, its migration counter differs from the one stored inside the TPM and the TPM will refuse to operate on this structure. This check must always be performed before a TPM context is allowed to operate on a TPMCS. Note that the migration counter cannot be reset and can only be incremented. Before a TPMCS can be migrated to another context, it must be ensured that the destination platform is in a trustworthy state. This should be done by remotely verifying the platform state of the destination platform using platform attestation. For this purpose, a number of proposed solutions exists that establish a secure attestation channel [27], [28]. We assume that a secure attestation channel has been established between both entities and that our migration protocol runs inside the resulting channel.

Figure 6 shows our migration protocol. It has similarities to the concept the TCG introduced with migratable keys and simply transfers an encrypted TPMCS blob to the other TPM. The TPMCS is directly bound to random nonces generated on

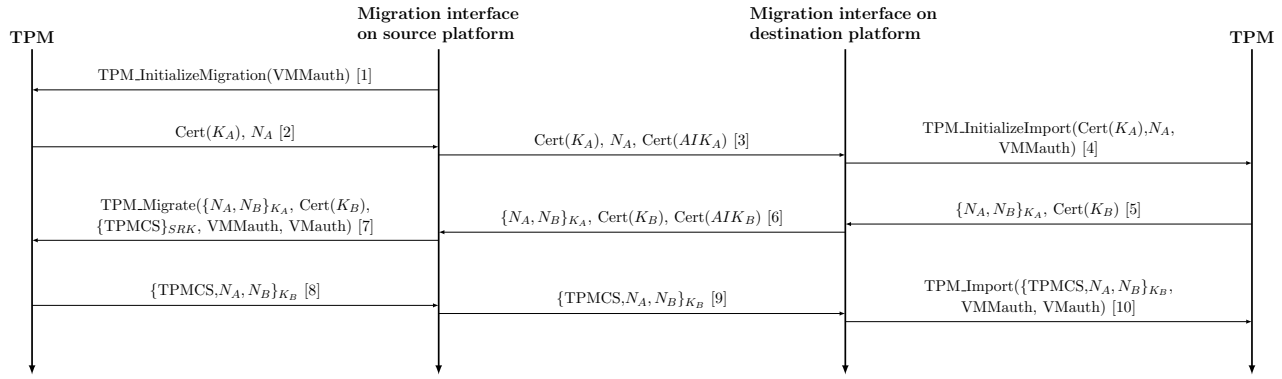

Fig. 6. High-Level Migration Protocol

the source and the destination platform to prevent a TPMCS from being migrated to multiple TPMs.

In the first two steps, the TPM generates a non-migratable TPM key K_A and certifies this key to provide assurances that it is held in a protected storage of a genuine TPM. This key is then transferred together with a nonce (N_A) and the AIK certificate to the destination platform, where the import is initialized. The migration interface verifies that the query is coming from a genuine TPM and that the K_A is protected by a TPM, using the provided certificates (Cf. [29], pp. 130). The destination TPM initializes the import procedure by generating and certifying its own non-migratable TPM key (K_B). For this purpose, the interface of the migration platform must provide owner authorization, including N_A and the certificate of the source platform. Then, the TPM delivers the encrypted package consisting of N_A and N_B to the migration interface. The migration interface collects all certificates that are necessary to prove that K_B is a TPM protected key and transfers the package back to the source TPM. The source migration interface then provides owner authorization to the TPM and delivers the received encrypted package, including the TPMCS that is to be migrated to the TPM. The TPM verifies authorization of the command and checks internally whether the migration counter conforms to the migration counter of the TPMCS that is to be migrated. If everything checks out, the TPM increments the internal migration counter and the migration counter of all other TPMCS that reside on this system. The TPM must ensure that the migration counter of each TPMCS is only incremented once, since otherwise, an old TPMCS could be replayed into the system.

The TPM then encrypts the TPMCS and the nonces with the delivered TPM key of the destination platform and transfers the package to the destination platform. The TPM of the destination platform imports the package and activates the TPMCS if the nonces and the asserted authorization data are correct. The migration procedure finishes by setting the migration counter of the TPMCS to the destination's TPM actual value of the internal migration counter and by re-certifying the EK of the TPMCS.

VII. TPM Credentials

Every TPM is uniquely identifiable by the Endorsement Key (EK). This EK is pre-installed by the manufacturer of the TPM and is used to generate the owner-password.

In order to provide full-fledged TPMs for every TPM context, every context needs to possess its own EK. However, it should also be possible for a verifier to decide whether the EK of a TPM context is an authentic EK, i.e., generated and protected by a hardware TPM. It is possible for a manufacturer to generate a number of EKs and integrate them on our multi-context TPM. However, we believe that this process is not feasible, since this would require an additional overhead for the manufacturer. In addition, all EKs and certificates could be migrated out of the domain of the multi-context TPM and deplete the device of its pre-installed EKs.

We propose to establish a certificate chain with the EK, which is held in the root-TPM structure as a root node. For every TPM context, the TPM generates its own EK, which then becomes certified by the root EK. Generating EKs directly on the TPM is a feature that current TPMs already support (Cf. [29], pp. 141). This process allows us to obtain AIKs for every TPM context simply by verifying the certificate chain of the EK. If a TPM is migrated, the EK of the migrated structure must then be re-certified with the EK that resides on the destination platform.

In the following, we provide the protocols for creating and exiting a TPM context. These protocols are exemplary for the other TPM commands that we introduced in IV-D.

A. Spawning a TPM context

The protocol for creating a new TPM context is shown in Figure 8. First, a TPM Object-Specific Authorization Protocol (OSAP) session is created between the TPM and the VMM. This command computes a cryptographic secret and a handle (H) to this session; both to protect the whole session traffic (Cf. [22], pp. 71). The returned handle is then used together with the authentication data (owner-password and SRK-password) to issue the `TPM_Launch` command. The TPM then internally creates a new TPM context and adds an EK and the corresponding credential to this structure.

1.	VMM $\rightarrow$	TPM	: `TPM_OSAP()`
2.	VMM $\leftarrow$	TPM	: H
3.	VMM $\rightarrow$	TPM	: `TPM_Exit`(H, *authData*)
4.		TPM	: Store TPMCS values in D_{n+1}
			: Create PCRInfo Structure
			: $\{D_{n+1}\}_{SRK}$ =*Seal*(D_{n+1}, PCRInfo, H)
			: $sig_{n+1} = D(D_{n+1}, K_{TPM}^{-1})$
5.	VMM $\leftarrow$	TPM	: $\{D_{n+1}\}_{SRK}, sig_{n+1}$
6.	VMM $\rightarrow$	TPM	: `TPM_Resume`(H, $\{D_n\}_{SRK}, sig_n$, *authData*)
7.		TPM	: Check if *migration counter* $\stackrel{?}{=}$ *TPMCS.migration counter*
8.			: D_n =*Unseal*($\{D_n\}_{SRK}, H$)
			: Verify sig_n with D_n
			: Load D_n into TPMCS
			: CR0 = 1
9.	VMM $\leftarrow$	TPM	: TPM_SUCCESS

Fig. 7. Exiting and Resuming a TPM context (CR of the TPM is set to 0)

1.	VMM $\rightarrow$	TPM	: `TPM_OSAP()`
2.	VMM $\leftarrow$	TPM	: H
3.	VMM $\rightarrow$	TPM	: `TPM_Launch`(H, *authData*)
4.		TPM	: Create empty D
5.		TPM	: Create 2048-bit Endorsement key-pair (K_{vEK}, K_{vEK}^{-1})
			: Sign K_{vEK} with K_{EK}^{-1}
			: Add Cert(K_{vEK}) and K_{vEK} to D
6.		TPM	: Load D into TPMCS data field
7.		TPM	: CR0 = 1
7.	VMM $\leftarrow$	TPM	: `TPM_SUCCESS`
8.	VMM		: `LaunchandMeasureVM`

Fig. 8. Creating a new TPM context (CR of the TPM is set to 0)

After the TPMCS has been created, the VMs integrity is measured [4] and the obtained measurements are added to the PCR value of the current loaded TPMCS.

B. Exiting a TPM context

Exiting requires that the current loaded TPMCS is stored on the background storage. To prevent an attacker from being able to inject corrupt data structures, a TPMCS is sealed and signed with keys that are protected by the TPM. Figure 7 shows how this is realized. The TPM creates a special purpose data structure D_n in which the current values of a TPMCS are stored. This data structure is then sealed to the actual platform configuration, which must also include the integrity of the VMM.

This can easily be achieved by using the Safer Mode Extension (SMX) of the Intel TXT technology [12] or the AMD Secure Virtual Machine technology [13]. Both provide a special CPU command; in the case of Intel, it is called *getsec* and in the case of AMD, it is called *skinit*. These commands measure the VMM into a PCR and thus create a so-called *Dynamic Root of Trust for Measurement* (DRTM).

We denote the sealing of the structure D_{n+1} at a specific time Δt with *Seal*(D_{n+1}, PCRInfo, H). H is a key-handle for the storage root key and PCRInfo is a `TPM_PCR_INFO` structure that contains the information to which PCRs D_{n+1} will be bound. The operation to unseal is denoted as *Unseal*($\{D_{n+1}\}_{SRK}$) where for simplicity reasons $\{D_{n+1}\}_{SRK}$ also includes the structure of the platform configuration registers. Unsealing the $\{D_{n+1}\}_{SRK}$ at $\Delta t+x$ is then only possible if the current platform configuration is equal to the platform configuration that $\{D_{n+1}\}_{SRK}$ is bound to.

The data structure (D_n) is additionally signed using a TPM specific signing key (K_{TPM}^{-1}), which is stored in the root-structure of the TPM only for that purpose. The sealed data structure and the corresponding signature is then passed to the VMM, which stores it locally. The VMM then executes the `TPM_Exit` command and delivers the sealed data structure and the corresponding signature of the TPM context which should be resumed by the TPM. The TPM internally verifies whether the *migration counter* of the TPMCS is consistent with the one stored inside the *migration counter* of the root-data structure. This verification prevents against an already migrated TPM context or a replayed TPM context being again loaded into the TPM. The TPM then unseals the structure and verifies that the signature sig_{n+1} is a valid signature of D_{n+1}. If the signature is valid, implying that the data structure was generated on this specific TPM, the TPM loads the values D_{n+1} into its internal memory.

VIII. Conclusions

Trusted Computing technologies provide a sound way of securing computer systems and also a technological means for trust establishment. For this purpose, the Trusted Computing Group introduced a hardware module called trusted platform module (TPM) that protects cryptographic secrets and is capable of acting as a trust anchor. However, the TPM cannot be used directly in next-generation operating systems that utilize virtualization technologies. In this paper we proposed an efficient approach for using TC-technology in virtual environments. Our approach extends the TPM specification and

shows how a hardware TPM that is capable of supporting virtualization with hardware measures should be designed. To provide hardware-based protection domains, we introduced a second TPM privilege level and a TPM Control Structure. The combination of both concepts allows a virtual environment to directly operate on the TPM without loss of security properties. Since the approach we presented in this work utilizes recent developments in the virtualization technology of processor architectures, it could easily be adapted to integrate trusted computing technology in next-generation processor architectures. In that case, highly-efficient and secure trusted computing technology would be available to next-generation operating systems that are based on virtualization.

We are currently working on the implementation of our proposed TPM architecture. In this context, we are planing to extend the TPM emulator with our proposed concepts and to integrate it into the Xen-hypervisor. Based on the implementation we will then be able to evaluate the exact performance of our proposal.

IX. Acknowledgments

We would like to thank our colleagues Omid Tafreschi, Lars Fischer, Christian Schneider, Thomas Stibor and Christoph Krauß for fruitful discussions that helped to improve the quality of this paper. We would also like to thank Nicolai Kuntze and Andreas Schmidt from the Fraunhofer Institute for Secure Information Technology (SIT) and the anonymous reviewers for their valuable comments. This work was funded in part by the German Research Foundation (DFG) under grant EC 163/4-1, project *TrustCaps*.

References

[1] Trusted Computing Group, "Trusted Comput Group specifications," https://www.trustedcomputinggroup.org/specs/, Tech. Rep., 2008.

[2] Trusted Computing Group, "Trusted Platform Module (TPM) specifications," https://www.trustedcomputinggroup.org/specs/TPM, Tech. Rep., 2008.

[3] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn, "Attestation-based policy enforcement for remote access," in *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 2004, pp. 308–317.

[4] F. Stumpf, M. Benz, M. Hermanowski, and C. Eckert, "An Approach to a Trustworthy System Architecture using Virtualization," in *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC-2007)*, ser. Lecture Notes in Computer Science, vol. 4158. Hong Kong, China: Springer-Verlag, 2007, pp. pp. 191–202.

[5] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vtpm: virtualizing the trusted platform module," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006, pp. 21–21.

[6] A.-R. Sadeghi, M. Scheibel, C. Stüble, and M. Wolf, "Play it once again, sam - enforcing stateful licenses on open platforms," in *2nd Workshop on Advances in Trusted Computing (WATC '06 Fall)*, Tokyo, Japan, November 2006.

[7] B. Jansen, H. Ramasamy, and M. Schunter, "Flexible integrity protection and verification architecture for virtual machine monitors," in *Second Workshop on Advances in Trusted Computing*, 2006.

[8] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[9] 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. [Online]. Available: citeseer.ist.psu.edu/dragovic03xen.html

[10] M. Peinado, Y. Chen, P. England, and J. Manferdell, "Ngscb: A trusted open system," in *Proceedings of the 9th Australasian Conference (ACISP 2004)*, ser. Lecture Notes in Computer Science. Springer Berlin, 2004, pp. 86–97.

[11] Intel, "Intel virtualization technology for directed i/o architecture specification," Intel, Tech. Rep., 2006.

[12] Intel, "Intel(r) trusted execution technology preliminary architecture specification," Tech. Rep., November 2006.

[13] AMD, "Amd secure virtual machine architecture reference manual," Tech. Rep., 2005.

[14] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can We Make Operating Systems Reliable and Secure?" *Computer*, vol. 39, no. Issue 5, pp. 44–51, May 2006.

[15] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A trusted open platform," *Computer*, vol. 36, no. 7, pp. 55–62, 2003.

[16] J. Liedtke, "On Micro-Kernel Construction," in *SOSP '95: Proceedings of the fifteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM Press, 1995, pp. 237–250.

[17] EMSCB, "Towards Trustworthy Systems with Open Standards and Trusted Computing," http://www.emscb.de/, 2006.

[18] OpenTC, "Open trusted computing (opentc)," http://www.opentc.net/, 2007.

[19] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kaegi, F. Leung, and L. Smith, "Intel virtualization technology," *IEEE Computer Society*, vol. 5, pp. 48–56, July 2005.

[20] A. Whitaker, M. Shaw, and S. Gribble, "Denali: Lightweight virtual machines for distributed and networked applications," in *Proceedings of the 5th USENIX Symposium on Operating Systems*, 2002. [Online]. Available: citeseer.ist.psu.edu/whitaker02denali.html

[21] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, July 2005.

[22] T. C. Group, "TCG TPM Specification Version 1.2 Revision 103, Design Principles," Tech. Rep., July 2007.

[23] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the USENIX 2005 Annual Technical Conference*, 2005, pp. 41–46.

[24] R. P. Goldberg, "Survey of virtual machine research," *Computer*, June 1974.

[25] J. Robin and C. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th USENIX Security Symposium, Denver, CO*, August 2000. [Online]. Available: citeseer.ifi.unizh.ch/robin00analysis.html

[26] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide," Intel, Tech. Rep., February 2008.

[27] F. Stumpf, O. Tafreschi, P. Röder, and C. Eckert, "A Robust Integrity Reporting Protocol for Remote Attestation," in *Second Workshop on Advances in Trusted Computing (WATC'06 Fall)*, Tokyo, Japan, November 2006.

[28] Y. Gasmi, A.-R. Sadeghi, P. Stewin, M. Unger, and N. Asokan, "Beyond secure channels," in *STC '07: Proceedings of the 2007 ACM workshop on Scalable trusted computing*. New York, NY, USA: ACM, 2007, pp. 30–40.

[29] T. C. Group, "TCG TPM Specification Version 1.2 Revision 103, Commands," Tech. Rep., July 2007.