1. INTRODUCTION

Virtualization technology enables cloud computing architectures to achieve efficient sharing of system resources between a large number of users. In particular, a cloud computing system consists of a hypervisor that manages several guest virtual machines (VMs) that belong to different users. Each VM consists of an operating system (guest OS) that hosts (guest) applications. The growing popularity of such architectures has led to the development of processors with extensive support for virtualization [2]. These processors usually consist of multiple cores thereby enabling parallel execution of multiple VMs.

Current processors include security extensions (e.g., Intel Trusted Execution Technology (TXT) [4], AMD Secure Virtual Machine (SVM) [1]) to bootstrap a secure execution environment on an untrusted operating system. These security extensions provide a dynamic root of trust for measurement (DRTM), i.e., they allow a user to verify the integrity of the target application at launch-time and protect it against run-time intervention from the OS or other co-resident applications. However, existing architectures support only one secure execution environment to be run on a multi-core system at any point in time (i.e., each core is either part of the single secure environment or is halted [24, 37]). Hence, these architectures cannot be used in scenarios where several users, each running their own VM, concurrently require the setup of their own independent secure execution environment. Such a scenario could easily arise if the users/VMs use these execution environments to, for example, securely manage sensitive keys.

In this work, we address the problem of realizing multiple, parallel verifiable secure execution environments on multi-core processors. More specifically, our goal is to enable one secure environment per VM rather than per system, i.e., each user must be able to dynamically switch from normal VM execution to running his sensitive application in a secure environment, concurrently with other users. Each such environment should protect its target guest application against attacks from the underlying guest OS, DMA-capable peripherals and other co-resident VMs (as shown in Figure 1). Furthermore, the end-user should be able to verify the setup of the secure environment, i.e., obtain and verify a chain of trust from the hardware up to the application that executed in the secure environment.

One approach to address this problem is to set up parallel secure execution environments relying on the hypervisor to multiplex between them. To set up each environment, the hypervisor can use either a software-virtualized TPM [14] or
Hypervisor
Sensitive Application in
Core1Processor
Guest OS

core processor and requires only a lean, ports one secure execution environment per VM on a multi-
systems. It consists of an enhanced x86 architecture that sup-
support concurrent secure environments in multi-core sys-
that does not depend on the hypervisor to create or manage
form itself. Therefore, in this work, we design a solution
cause in cloud environments, hypervisor (e.g., resource allo-
gation support and security extensions, (ii) a disengaged
visor. The disengaged hypervisor is responsible only for the
startup and shutdown of VMs; it does not interact with them
at run-time and is hence, not involved in the setup, execu-
tion or tear-down of secure environments. This is the key
difference between our architecture and existing solutions.
Such hypervisor disengagement not only reduces the risk of
hypervisor compromise but also allows users to monitor ap-
plications without prior interaction with the hypervisor.

Our architecture consists of three main components: (i) a
multi-core processor in which each core has separate virtu-
alization support and security extensions, (ii) a disengaged
hypervisor that is responsible for VM management (i.e., cre-
ation, start-up, shutdown and deletion) and (iii) a hardware-
based virtualized Trusted Platform Module (HV-TPM) with
DRTM capabilities. Our disengaged hypervisor is similar to
noHype [36] in that it assigns resources to VMs and relies on
virtualization extensions of the underlying hardware to al-
low VMs to execute without run-time support from it. How-
ever, unlike noHype [36], it performs additional tasks during
VM creation to enable the VM to dynamically create and
tear-down a secure execution environment without any fur-
ther assistance from it. Modern peripheral and I/O devices
include hardware/firmware support for virtualization to im-
prove performance [3] and therefore enable the deployment of
disengaged hypervisors. However, no platform currently
supports the management of concurrent secure execution en-
vironments in the context of disengaged hypervisors. Our
architecture fills this gap through a new hardware-based vir-
tualized TPM (HV-TPM) and through light-weight proces-
sor modifications, which enable one secure execution envi-
ronment per core. HV-TPM, unlike previous solutions [10,
14, 16, 28, 35], can handle DRTM requests from multiple
cores in parallel without run-time hypervisor support.

We analyze the security of our architecture and show that
it effectively protects sensitive guest applications against
attacks from their underlying guest OS, malicious DMA-
capable peripherals and other VMs. To demonstrate the
feasibility of realizing our architecture, we emulate it using
QEMU [5]. QEMU does not currently support either Intel
virtualization extensions or Intel TXT. As part of our im-
plementation, we add support for special TXT instructions,
TXT specific register sets and DRTM-capable TPM emu-
lators (a modified version of [34]) to QEMU. Finally, since
QEMU does not support Intel virtualization extensions, we
emulate a simplified version of our proposed architecture.
Our emulated architecture consists of a Linux OS (hyper-
vvisor), two untrusted user-space ELF-loaders (guest OSes)
and two target elf binaries (sensitive applications) and we
show that it is equivalent to our proposed architecture.

The rest of the paper is organized as follows. In Sec-
tion 2, we summarize the virtualization and secure execu-
tion technology supported by current Intel architectures. We
then describe our problem statement in Section 3, our solu-
tion in Section 4 and analyze its security in Section 5. We
demonstrate the feasibility of realizing our solution through
a prototype implementation in Section 6. We present related
work in Section 7 and conclude in Section 8.

2. BACKGROUND

In this section, we briefly review architectural support
for secure execution environments in current Intel proces-
sors. We begin by describing the Trusted Platform Module
(TPM), which is an important component of Intel’s secure
execution technology.

2.1 Trusted Platform Module

The TPM is a hardware cryptographic module that pro-
vides two important services, namely, secure storage and
platform attestation. It consists of a CPU (execution en-
gine), volatile memory (RAM for execution) and persistent
storage. It also includes cryptographic engines for random
number generation, SHA-1, RSA key generation, encryption
and signing. It has a special set of registers called Platform
Configuration Registers (PCRs). PCRs can only be writ-
ten through an extend operation, i.e., writing ‘X’ to a PCR
actually writes the SHA-1 hash of the concatenation of its
current value and ‘X’ into the PCR. Verifying the hash-
chain starting at the initial content of a PCR to its current
value is referred to as verifying the value of a PCR.
2.2 Intel Architecture Overview

A typical Intel-based system consists of four main components: processor, memory, chipset and peripherals (Figure 2). The processor consists of several cores each of which contains a set of general-purpose registers, virtual memory or paging support (Memory Management Unit (MMU)) and a Local Advanced Programmable Interrupt Controller (LAPIC). Each core also has an internal cache and shares an external cache with all the other cores. The processor accesses memory and peripherals through the chipset over a special bus called the Front-Side Bus (FSB). The chipset contains controllers for memory and peripheral buses like PCI Express (PCIe), Low Pin Count (LPC) bus, etc. It also contains an I/O interrupt controller (IOAPIC) for controlling interrupts from peripherals.

Modern processor cores and chips also include virtualization support (called Intel Virtualization Technology (VT)) to allow efficient sharing of system resources, namely, the CPU, memory and peripherals between a VM and its hypervisor. Each core supports CPU virtualization by allowing the guest OS to share privilege ring-0 of execution with the hypervisor and efficiently context switching between them. Moreover, each core allows the guest OS to maintain its own virtual-to-physical memory mapping as though it were running as the main OS; the guest’s physical addresses are then translated to actual machine memory addresses using a special set of tables called Extended Page Tables (EPT). A subset of Intel VT extensions called Intel VT for Direct I/O (VT-d) enables efficient I/O virtualization.

Depending on their initial value, PCRs are classified as either static or dynamic. PCRs 0–15 are called static because they can only be reset on system boot. PCRs 16–23 are called dynamic; they are initialized to −1 at boot and can be reset at run-time. A subset of the dynamic PCRs, PCRs 17–23 can be reset only with special access privileges also referred to as localities. TPMS support five distinct access privileges/localities using five memory pages and all access are by default at locality 0. Localities 1–4 are used for Intel’s Trusted Execution Technology (Section 2.3).

2.3 Intel Trusted Execution Technology

The growing complexity of operating systems makes them vulnerable to bugs that can be exploited to compromise them. This limits their ability to provide secure execution environments to sensitive applications. Intel TXT addresses this problem by enabling a user to bootstrap a secure execution environment on an untrusted OS. It achieves this using four main trusted components:

1. Processor Extensions: Each processor core supports a special instruction called GETSEC [2] which behaves differently depending on its input. Three special instances of GETSEC, called EXITAC, SENTER and SEXIT are used to set up and tear-down the secure execution environment.

2. Chipset Extensions: Chipset support for Intel TXT includes TXT-configuration registers and special FSB mechanisms to detect SENTER and SEXIT. The chipset also enforces fine-grained access control to certain address spaces, e.g., sensitive TXT-configuration registers and TPM localities 1–4.

3. Trusted Platform Module (TPM): Intel TXT uses the TPM as secure storage for information about the secure environment which can later be verified by the user through TPM attestation services. More specifically, Intel TXT uses PCRs 17–18 to store information about the secure environment.

4. Authenticated Code Module (ACM): The ACM is a signed code module provided by Intel and is used to boot-
Intel TXT combines the above components to bootstrap a secure execution environment called a Measured Launched Environment (MLE). The term MLE is also used to refer to the actual code (e.g., a trusted OS, any sensitive application) that is executed within such an environment. We now summarize this bootstrap process (Figure 3).

The OS first creates a memory region that is protected against DMA access and maps the MLE to this region. The OS also stores information regarding the MLE’s hash and location as well as any data needed for its own resumption against DMA access and maps the MLE to this region. The OS then executes SENTER on one of the cores, henceforth called the primary core. Current Intel processors allow only one primary core per processor and this core is chosen by the BIOS during system boot.

The execution of the SENTER instruction consists of several stages. First, the primary core sends an inter-processor interrupt (IPI) to all other cores requesting them to halt. After receiving an acknowledgment from all cores, the primary core fetches the ACM and loads it into cache. The core then computes its hash and signature and verifies them against their expected values. If this check passes, the processor initiates the reset of PCRs 17–23 of the TPM at locality 4 using special LPC bus cycles and extends PCR-17 with the ACM’s hash. The processor requests the chipset to unlock access to TPM locality 3.

The ACM executes from the cache using a special mechanism called Cache-as-RAM [18]. It first verifies that it is located in protected memory. It then measures the MLE and extends PCR-18 with its hash at TPM locality 3. It then executes EXITAC upon which the chipset closes access to locality 3 and unlocks locality 2 of the TPM. The chipset also unlocks the TXT-configuration registers that should be accessible only within the secure environment. The processor then transfers control to the MLE which wakes up all the other cores that were previously halted. The MLE continues to run until it executes the SEXIT instruction to tear-down the secure environment.

During tear-down, the MLE first deletes all secrets from memory and executes SEXIT. As part of SEXIT, the primary core again sends an IPI to all other cores requesting them to halt. Once they have all halted, the chipset locks access to TPM localities 1 and 2 and TXT registers that should be accessible only within the secure environment. Finally, the primary core wakes up the other cores using an IPI and resumes normal OS execution.

1. It should allow the co-existence of multiple, parallel secure execution environments so that multiple security-sensitive applications can execute concurrently. By securing environments, we mean an environment with a verifiable hardware-rooted chain of trust and secure run-time isolation. This is useful in cloud systems where multiple users may each require their own independent secure execution environment.

2. It must allow untrusted system components, including software, to execute alongside the secure execution environment(s) but prevent these untrusted components from interfering with the secure environment(s). This is important for example in cloud computing where a subset of VMs on a platform may be malicious and suspending all VMs to create a secure environment is not an option.

3. It must guarantee strong isolation between independent secure environments themselves. This is important to ensure that different sensitive applications that do not trust each other (e.g., belong to different users in the cloud) can execute in their own isolated environment.

4. It must allow the setup and tear-down of individual secure environments with minimal support from the trusted software base (hypervisor). This reduces the complexity of the trusted software base (hypervisor) and its interaction with untrusted system components (VMs) thereby lowering the risk of system compromise.

Our architecture relies on the following architectural components and enhancements.

Disengaged Hypervisor

Our architecture relies on the following architectural components and enhancements.
and virtual instances of physical devices to each VM and configures virtualization extensions such that VMs continue to execute independently without requiring run-time support from it. Using a disengaged hypervisor necessitates static allocation of resources to a VM during its execution, i.e., once resources are allocated to a VM, they become available again to the system only when the VM shuts down or is forcibly terminated. Using a disengaged hypervisor also requires virtualization-aware resources (CPU, memory, I/O devices); we note that this is not a limitation since modern processors and I/O devices have extensive virtualization support (as described in Section 2.2). Hypervisor disengagement minimizes interaction between the hypervisor and VMs and therefore reduces the risk of hypervisor compromise.

**Extended Access to the GETSEC Instruction**

In current Intel processors, the GETSEC instruction (in the form of EXITAC, SENTER and SEXIT) can only be executed by the hypervisor on a BIOS designated core. In our architecture, we require each VM or guest OS to be able to execute this instruction on its primary core (chosen by the hypervisor) without elevating privileges, i.e., without entering hypervisor mode. However, we do not allow nested execution of SENTER.

**Inter-processor Interrupt (IPI) Router**

Our IPI router ensures that when a VM executes SENTER or SEXIT, only its cores are affected (e.g., halted, woken-up) and that other VMs continue to execute uninterrupted. It obtains the mapping between VMs and their cores from the hypervisor. The router intercepts all IPs and ensures that only cores belonging to the same VM can interrupt each other. The IPI router also controls routing of the non-maskable interrupt (NMI), system management mode interrupt (SMI) and INIT interrupt to all cores.

**Per Core TXT Configuration Registers**

Since we intend to support potentially one secure environment per core, we need one set of TXT-configuration registers for each core in the chipset. Each register set also has an extra register that is used to specify the base address of the associated TPM address range (details below). The chipset enforces access control over all the sets of TXT registers depending on the core that originated the request. Important TXT-configuration registers and their access permissions are shown in Table 1 in Appendix A.

**Hardware-Virtualized TPM (HV-TPM)**

Figure 4 shows an overview of our HV-TPM. In addition to the usual TPM components, the HV-TPM maintains one TPM context per VM. The HV-TPM interface supports direct access from each VM to its TPM context and concurrent DRTM requests from VMs. This is unlike existing virtualized TPMs [10, 35] which need run-time hypervisor intervention and do not support DRTM requests from VMs.

Our HV-TPM enables native access from each VM to its TPM context through a separate memory-mapped interface. The hypervisor configures the number of such virtual interfaces that the HV-TPM must support and the mapping between each VM and the virtual interface assigned to that VM. The HV-TPM uses this mapping to identify the source of a TPM request and then schedules it for processing. The HV-TPM does not allow any VM to access or modify this mapping information to prevent malicious VMs from accessing TPM contexts belonging to other guests. Each virtual interface exposed by the HV-TPM is identical to the memory-mapped interface of current TPMs, i.e., it consists of a set of registers accessible at any of five localities.

Unlike the current TPM that can support just one DRTM request at a time, our HV-TPM supports concurrent DRTM requests from multiple VMs. The HV-TPM requires the hypervisor to specify each VM’s virtual interface in the VM’s TXT-configuration registers. Each VM’s primary core uses this information to identify the correct virtual interface to send the ACM and MLE hash during the execution of SENTER. The chipset also uses this information to control access to localities 1–4 of each VM’s virtual interface depending on the corresponding VM’s execution mode.

We note that such a HV-TPM interface can be realized by using a PCI Express interface that supports virtualization [7] instead of the traditional LPC interface that is used in current TPMs. When a processor core (guest OS) executes SENTER, the chipset can use special PCI Express packets analogous to special LPC bus cycles (Section 2.3), to send a DRTM request to HV-TPM. Using a PCI Express interface for the TPM increases the size of the ACM because it now needs a bigger driver module compared to the standard TPM’s LPC driver. This is not a problem because in our architecture, the ACM does not have any size restrictions as it executes from secure RAM memory and not from the cache which is the case in the current TXT implementation.

### 4.2 Secure Execution Environment Lifecycle

In order to understand how all of the above components can be used to create, manage and tear-down multiple secure execution environments dynamically, we describe the lifecycle of a user’s VM. Each such VM can execute either in legacy mode where the guest OS manages guest applications or in secure mode where a sensitive application runs in a dynamically-created secure environment.

**VM Creation**

The end user requests the creation of a new VM and specifies its resource requirements, i.e., the number of physical CPU cores, the total size of physical memory, the size of memory that should be available in a secure environment and necessary peripherals. The hypervisor creates a VM accordingly.
and assigns it a new unique and permanent identifier (ID). Figure 5 shows the contents of two guest VMs just after their startup. First, the hypervisor allocates dedicated cores to each VM and designates a primary core for each VM. For example, it allocates cores Core1 and Core2 to VM1 and fixes Core1 as its primary core. Each VM is allowed to execute EXITAC, SENTER and SEXIT only on its primary core.

Second, the hypervisor allocates dedicated physical memory to each VM by configuring its EPT. It also creates a special secure partition in this memory to which DMA access is disabled and which has write-back cacheability\(^1\). These protections are enforced by the hypervisor by appropriately configuring EPT and DMAR tables. We refer to this special partition simply as *secure memory*. The hypervisor then loads a copy of the ACM into one part of the secure memory region and configures the VM's EPT to allow read-only access to the ACM. It includes the ACM's authentic hash and signature as part of the ACM's header. Each VM uses the rest of its secure memory for loading the MLE (i.e., the sensitive application) and the MLE's page tables. Each guest can also store the information needed to resume its OS after tear-down of its secure environment.

Next, the hypervisor requests the HV-TPM to create a new TPM context or instance (TPMI) for each VM based on its ID. If an existing VM is re-started, then the HV-TPM loads an existing context instead of creating a new one. The hypervisor adds an entry that relates the ID and virtual TPM interface belonging to each VM into the HV-TPM's mapping table. We note that at this point, each VM can access its TPMi only under locality 0 and memory ranges corresponding to the other localities remain locked.

Fourth, the hypervisor also maps the set of TXT-configuration registers (labelled as \(R_{txt}\) Instance in Figure 5) corresponding to each VM's primary core into the VM's memory. It also writes the details regarding the VM's ACM (its location, size and signing key), secure memory and virtual TPM interface into these registers (Appendix A).

The hypervisor configures the IPI router such that only the primary cores of VMs can send IPIs and only cores belonging to the same VM can interrupt each other. The hypervisor also ensures that it controls the NMI interrupt line to each VM core through the IPI router so that it can stop the VM anytime. Finally, it maps interrupts from devices directly to each VM's LAPIC (similar to [36]).

From here on, each VM continues to run in its normal execution mode without hypervisor support. Until its shutdown, each VM will have exclusive access to its assigned I/O resources, continue to execute on the cores and use the physical memory assigned to it.

### Secure Execution Mode Setup

When the user wants to execute a sensitive application (MLE), he requests the guest OS to switch to the VM's secure execution mode. We note that although the user depends on the guest OS to execute these preparation steps, he does not have to trust it to do them correctly. Instead, our architecture enables any malicious behavior from the guest OS (e.g., change of the MLE, mapping of MLE to insecure memory) to be detected by the user.

Figure 6 shows the process through which two VMs (shown in blue and green) enter secure execution mode concurrently. Each VM's switch into secure execution consists of two phases: (i) measurement of its ACM copy and (ii) measurement and execution of its MLE. First, each guest OS maps its target MLE and the MLE's page tables into the secure memory region (Step 1). It then disables paging and all I/O interrupts and executes SENTER on its primary core (Step 2). The execution of SENTER results in the measurement and execution of the VM's ACM copy which in turn measures and executes the VM's MLE (Steps 3-7).

As one sees from Figure 6, each VM's entry into secure mode closely resembles the launch of a sensitive application using the current Intel TXT implementation (Section 2.3). However, there are some important differences. Unlike the current Intel TXT implementation where the ACM is forced to execute from cache because it has no secure memory, in our architecture each VM's ACM executes from secure memory protected by the hypervisor. As a result, our architecture has to ensure that the ACM is not executed from a cached version that differs from the one in memory. This is not easy to achieve in Intel processors which do not support VM-specific cache flushes. Current Intel processors support a CLFLUSH instruction that allows flushing of individual cache lines and this can be used multiple times to flush lines corresponding to the secure memory region before the ACM is executed. Similarly, address translation caches/translation look-aside buffers (TLBs) can also be flushed core-wise to remove old translations. These mechanisms can also be used to prevent executing the previously cached MLE versions.

Unlike Intel TXT, our architecture enables multiple VMs to enter secure execution mode concurrently. Below, we highlight the role of our extensions — the chipset registers, IPI router and HV-TPM in securely enabling this.

### Role of the TXT-Configuration Registers

Our architecture includes one set of TXT-configuration registers per core instead of per platform as in the current Intel TXT implementation. If a core is the primary core

---

\(^1\) The write-back cacheability is chosen to ensure conformance to the current Intel TXT requirements.
of any VM, then the hypervisor maps its associated TXT register set into that VM's address space. The hypervisor specifies details regarding each VM's ACM copy (location, key), secure memory range and TPM interface in the VM's TXT registers, i.e., in the ACM BASE, ACM KEY, SM BASE, SM SIZE and TPM BASE registers respectively (Appendix A). TPM BASE allows each VM's primary core to access the correct TPM interface, i.e., the interface that belongs to its parent VM on the execution of SENTER and SEXIT. It also allows the chipset to control access to localities 1–4 of each VM's TPM interface. Furthermore, each MLE can access and disable access to localities 1–2 of its TPM, through its primary core's TXT registers. We note that no VM (MLE) can access any other VM's (MLE's) TXT registers because of the EPT-based protection configured by the hypervisor.

**Role of the IPI Router**

In the current Intel TXT implementation, executing SENTER halts all cores except the primary core of the host to ensure that they do not interfere with the measurement and execution of the ACM and MLE. However, in our architecture, we require that only the cores belonging to the same VM are halted when a VM executes SENTER. The IPI router achieves this by intercepting all IPIs and re-routing them appropriately. Therefore, the IPIs issued by any VM's primary core during the execution of SENTER only affect the other cores of the same VM. The IPI router gets this mapping between each VM and its cores from the hypervisor. Furthermore, the IPI router ensures that only primary cores of VMs can generate IPIs and hence, execute SENTER and SEXIT. This is similar to the current Intel TXT design where only a BIOS designated core can execute these instructions.

The IPI router has another important responsibility in our architecture: it controls the NMI interrupt of all VM cores so that they can be forced to shutdown by the hypervisor. This is also true when a VM is executing in secure mode, i.e., any non-terminating MLE can be forcibly terminated by the hypervisor. We enable this in our architecture to ensure that the hypervisor is in complete control of system resources and to prevent DoS attacks from malicious VMs and MLEs. We note that this is in contrast to the current implementation of Intel TXT where MLEs can only terminate voluntarily by executing SEXIT, i.e., even the NMI interrupt is masked once SENTER is executed.

**Role of the HV-TPM**

In the current Intel TXT implementation, the execution of SENTER results in the TPM (i) resetting PCRs 17–23, (ii) extending PCR–17 with the ACM's hash and (iii) extending PCR-18 with the MLE's hash. In our architecture, we require the HV-TPM to perform similar operations on a VM's TPM context when the VM executes SENTER. Our architecture enables this as follows. The hypervisor specifies the TPM virtual interface of each VM in the TPM BASE register. During the execution of SENTER, the VM's primary core uses this information to send the ACM and MLE hash to the

---

Figure 6: Summary of the process of VMs switching from their normal execution into secure execution mode to execute a sensitive application (MLE).
HV-TPM. The HV-TPM identifies the source of the request using its internal table that maps between virtual interfaces and VM IDs. The hypervisor restricts each VM to accessing its own TPM instance by configuring the EPT appropriately.

Secure Execution Mode Tear-down

The secure execution environment in each VM is torn down either when the MLE executes SEXIT or when the hypervisor triggers the VM’s NMI interrupt as we describe below.

Each MLE prepares the secure environment for tear-down before executing the SEXIT instruction. It writes back all its current cache lines and then deletes secrets from memory. Each MLE extends the dynamic PCRs of its TPM, with a random value to prevent any other entity outside the secure execution environment from unsealing any of its secrets. Finally, each MLE re-locks access to locality 1 of its TPM, and executes SEXIT on its primary core. As part of the execution of SEXIT, first, each MLE’s primary core uses the IPI router to inform its other VM cores to prepare for tear-down and the address (resumption point) at which they should continue execution after SEXIT. Once the target cores acknowledge their readiness for tear-down and halt, the chipset locks access to locality 2 of the corresponding TPM, and other TXT registers that should only be accessible during secure execution. Each primary core then issues another IPI to other cores of its VM to signal that they can continue execution at the resumption point. Once SEXIT execution completes, each primary core itself continues execution at the next instruction after SENTER in its corresponding guest OS. In case an MLE misbehaves and does not terminate, the hypervisor issues an NMI interrupt to the VM cores through the IPI router. On receiving an NMI, each core resets and enters a halt state and is then ready to be reallocated.

5. SECURITY ANALYSIS

We analyze the security of our architecture and show that it supports multiple parallel secure execution environments.

5.1 Attacker Model and Security Goals

We consider a remote adversary (A) that controls one or more guest OSes and/or guest applications in a virtualized system but does not have physical access to the system. More specifically, A may compromise software corresponding to the guest OS or applications that are part of the same VM as the sensitive application and/or other VMs in the system. A may also control one or more DMA-capable peripherals. However, we assume A cannot gain control over the hypervisor or the underlying hardware (e.g., chipset, TPM). This model closely resembles the cloud computing setting where a collocated VM may be under an attacker’s control and even the user’s VM may be remotely compromised.

Given this attacker model, we show that our architecture allows each guest’s sensitive application to execute in a secure environment despite the presence of other potentially compromised system components. More specifically, we show that our design enables the secure and verifiable launch of each MLE, i.e., the user obtains accurate information regarding the sensitive application that executes as part of the secure environment in his VM (Property I). This information consists of a chain of trust from the hardware to the sensitive application with the assurance that other compromised system components have not tampered with the information. We also show that as long as the sensitive application does not modify its own code at run-time, any code that was not measured by the ACM did not run within the secure environment (Property II). Finally, we show that our architecture guarantees these properties even when multiple VMs execute their own sensitive application concurrently and alongside other untrusted software, and that compromised applications of VMs cannot undermine the hypervisor’s overall control over system resources (Property III).

In what follows, we first show how our architecture achieves the Properties I through III by detailing how it prevents attacks during the setup and operation phases of a secure execution environment within a target VM. We then extend our analysis to the scenario where multiple VMs enter secure execution concurrently and run alongside other untrusted software. For brevity, we refer to attackers that control the target VM’s guest OS or applications, other VMs in the system and malicious peripherals as $A_{os}$, $A_{vm}$, and $A_p$, respectively. We also refer to the target VM’s ACM copy and its MLE simply as the ACM and MLE respectively.

5.2 Security Analysis of the Setup Phase

The security of the setup phase is crucial for guaranteeing Property I, i.e., ensuring that $A_{os}$, $A_{vm}$, and $A_p$ do not interfere with the process of measuring the ACM and the MLE. Given that A cannot modify the underlying hardware or compromise the hypervisor, he can affect the setup of the secure environment only by: (i) interfering with the measurement of the ACM or (ii) modifying the ACM after it has been measured or interfering with its execution.

ACM Protection

Execution of the ACM always begins at the location specified in the target VM’s ACM BASE register which is only writable by the hypervisor. Therefore, no other entity, namely $A_{os}$, $A_{vm}$ or $A_p$ can force a different ACM to execute. Furthermore, the hypervisor maps the ACM into memory pages that are read-only to the target VM. This prevents $A_{os}$ from modifying it. Similarly, the EPT together with the hypervisor prevent $A_{vm}$ from accessing memory belonging to the target VM and hence, its ACM copy. Finally, since the hypervisor maps the ACM to memory pages that are not accessible by DMA, $A_p$ also cannot modify the ACM. Additionally, the IPI router and masking of all interrupts before the execution of SENTER (and its subsequent verification before the ACM measurement) prevent $A_{vm}$ and $A_p$ from interrupting the measurement of the ACM during SENTER.

The above protections equally apply during the execution of the ACM. Therefore $A_{os}$, $A_{vm}$, and $A_p$ cannot affect the ACM execution and thereby, the measurement of the MLE in any way. However, $A_{os}$, $A_{vm}$, and $A_p$ may try to modify PCR-17 and PCR-18 of the target VM’s TPM, that holds the ACM and MLE measurements directly. Our architecture prevents such attacks from $A_{vm}$ by ensuring that any VM can only access its own TPM. It enforces this by using the hypervisor to configure (i) each VM’s EPT to allow access only to the VM’s own TPM’s memory-mapped interface and (ii) the TPM instance that each VM’s primary core is allowed to access during SENTER using the core’s TPM BASE register.

Although $A_{vm}$ which has access to its VM’s TPM can directly modify (extend) PCR-17 and PCR-18 outside the secure environment, it cannot change them to match a correct value of its choice. This is because these PCRs are initialized to -1 at boot, can only be reset during the execution
5.3 Security of MLE Execution

The main security requirement for MLE execution is that no code that was not originally measured as part of the MLE must be allowed to execute *(Property II)* within the secure environment. This must hold as long as the MLE does not modify its own code at run-time. However, if the MLE is inherently vulnerable (*e.g.*, has a buffer overflow) or if it depends on other compromised system components (*e.g.*, malicious peripheral), there is little we can do to protect it. Our architecture protects vulnerability-free MLEs as follows.

Similar to Intel TXT, once the SENTER instruction is executed, the guest OS is completely suspended and therefore, a malicious guest OS (*A*$_{os}$) cannot interfere with the MLE’s execution. Flushing the cache lines corresponding to the secure memory regions before the ACM executes ensures that any modified versions of the MLE in the cache are written back to memory. Similarly, invalidating TLBs before ACM execution guarantees that old memory translations (which are controlled by the guest OS) are discarded. Since the ACM measures the MLE only after these operations, it ensures that it hashes the MLE code that finally executes.

The EPT-based memory protections configured by the hypervisor prevents A$_{vm}$ from accessing the MLE. Since the hypervisor has exclusive control over the EPT, A$_{vm}$ cannot modify these EPT-based protections to gain access to another VM’s memory. Similarly, memory accesses from DMA-capable peripherals are restricted by the DMAR tables. The hypervisor configures the DMAR tables with information regarding which memory pages of the guest should be accessible via DMA. As long as the MLE and its page tables are not mapped into one of these pages, they are protected against unauthorized modification by malicious peripheral devices (*A*$_{p}$). However, this mapping of the MLE to memory is done by the guest OS which is also untrusted (*A*$_{os}$), i.e., the guest OS may map the MLE into insecure memory that is accessible by DMA.

Our architecture prevents such attacks by using the ACM to ensure that all the MLE’s pages are inside the secure memory range using the SM BASE and SM SIZE registers. This forces the guest OS to map the MLE and its page tables correctly to the secure memory regions. Additionally, the guest OS has to ensure that the MLE’s page tables allow access only to pages within the secure memory boundaries because otherwise, the violation will be detected by the ACM. This prevents attacks from an *A*$_{os}$ that maps the MLE twice: once into the secure region and again into the insecure region of memory and configures the MLE’s page tables to point to the copy in the insecure region. Finally, by preventing recursive calls to SENTER, we prevent *A*$_{os}$ from executing a modified MLE that uses SENTER to reset PCRs 17–23 and extend them with correct values.

Therefore, as long as the MLE does not modify its own code during execution, no other code that was not previously measured by the ACM can execute within the secure environment *(Property II)*.

5.4 Security of Concurrent MLE Executions

The above discussion on how *A*$_{vm}$ cannot affect any other target VM’s secure execution environment holds even when *A*$_{vm}$ is a malicious VM executing in secure mode, i.e., *A*$_{vm}$ is part of a malicious MLE. More specifically, a malicious MLE in one VM cannot affect any other VM that is executing in secure mode because of the strong CPU, memory and I/O isolation enforced by the EPT, DMAR-tables and virtualized I/O devices together with the hypervisor. Hence, our architecture allows multiple secure execution environments to co-exist alongside other untrusted VMs and applications.

In our architecture, the hypervisor will always be able to stop a VM through the IPI router that controls the VM’s NMI interrupt line. This prevents DoS attacks from malicious guest OSes that refuse to launch a secure environment and malicious MLEs that never exit the secure environment. Hence, our architecture satisfies *(Property III)*.

6. IMPLEMENTATION

To demonstrate the feasibility of realizing our architecture, we implement it using QEMU [5]. Since our architecture requires changes to the x86 platform, we could only build an emulator-based prototype. Currently, QEMU x86-emulation does not support Intel VT or Intel TXT. As a first step, we implement a basic Intel TXT framework.

Our prototype is based on QEMU’s PC35 machine. We configure it to emulate four Intel x86 cores and a chipset consisting of Intel’s Q35 and Intel’s ICH9 chips. We extend the ICH9 chip to support two sets of TXT-configuration registers and appropriate access control mechanisms. We extend QEMU’s existing x86 instruction set with EXITAC, SENTER and SEXIT instructions. Since QEMU does not support Intel VT, we simplify the implementation of SENTER to work with paging enabled. We modify the existing TPM interface in QEMU [6] to use the TPM Emulator from [34] as the backend engine. We also add DRTM-support to the interface and emulator. Our prototype uses two such DRTM-enabled TPM interfaces, each with its own backend emulator.

Figure 7 shows a schematic of our prototype and how it maps to our architecture (Section 4). Our prototype consists of Linux running inside QEMU with the above TXT frame-
work extensions. Two (untrusted) user-space ELF-loaders are used in place of guest OSes. We use ACM stub code that currently only executes EXITAC but we note that it can be extended to include any additional functionality similar to the Intel-supplied ACM. Our prototype uses two simple binaries that print "Hello World" and execute SEXIT as MLEs. Linux (hypervisor) allocates secure memory for the ACMs and MLEs using kernel buffers. It loads two ACMs, one for each ELF-loader (guest OS), into separate buffers and ensures that the ELF-loaders cannot overwrite ACM buffers. It then assigns each ELF-loader to a core for execution.

Our prototype semantically matches our design. Linux (hypervisor) confines each ELF-loader (VM) to its own core, memory and TPM instance. Each ELF-loader (VM) also cannot modify either the security (DMA, cache) settings of its ACM’s or MLE’s memory or its ACM copy itself. Although our architecture requires SENTER to work with paging disabled because the guest page tables are not trustworthy, this is unnecessary in our prototype because Linux (hypervisor) exclusively controls each ELF-loader’s page tables. Finally, our prototype does not include the IPI router because each ELF-loader is assigned only one core and only Linux’s cores can initiate IPIs; therefore, the ELF-loaders (VMs) and ELF binaries (MLEs) cannot interrupt each other.

The prototype works as follows. First, Linux allocates four secure memory regions: two for the ACMs that Linux itself loads and two for the sensitive applications (MLEs) that are loaded by their own ELF-loader. Then, Linux chooses two cores, one for each ELF-loader and writes the location of their respective ACMs and TPM instances into their TXT registers. Finally, it starts each ELF-loader on its dedicated core using CPUSETS. Each ELF-loader loads its target ELF binary, writes its ELF’s start address into its core’s TXT registers and executes EXITAC. During the execution of SENTER, first, each ELF-loader’s core informs the chipset about its entry into secure execution. In response, the chipset unlocks locality 4 of that core’s TPM instance. Next, each ELF-loader’s core fetches the ACM and sends it to its designated TPM emulator using locality 4 privileges. The TPM emulator resets PCRs 17–23 and extends PCR-17 with the hash of the received data. Then, each core closes locality 4, unlocks locality 3 of its TPM instance and starts executing its respective ACM.

When each ACM executes the EXITAC instruction, its core requests the chipset to lock locality 3 and unlock locality 2 of its TPM. At this point, the chipset also unlocks access to a subset of the core’s TXT registers that must only be accessible within the secure environment. Each ELF-loader’s core, then, transfers control to its target ELF. When an ELF executes the SEXIT instruction, its core locks localities 1 and 2 of its TPM as well as access to the previously unlocked TXT registers. Finally, each core transfers execution control back to its parent ELF-loader.

7. RELATED WORK
Realizing Secure Execution Environments. Recently, Intel announced a new range of processor extensions called Software Guard Extensions (SGX) that support multiple concurrent secure execution environments called enclaves [8, 19, 25]. Intel SGX protects the confidentiality and integrity of code and data within each enclave against unauthorized access from other enclaves and their underlying untrusted hypervisor/OS. However, it still relies on the (untrusted) hypervisor/OS to manage the setup and tear-down of these enclaves. Each enclave can also invoke (non-critical) hypervisor/OS services, e.g., networking, storage, etc.

One way to create secure environments using Intel SGX would be to use an engaged hypervisor to setup, service and tear-down enclaves but this increases the risk of hypervisor compromise. Although hypervisor compromise does not violate the security guarantees that Intel SGX provides to individual enclaves, it could have serious implications for the usability of other co-resident enclaves, VMs and the platform itself. For example, compromise of the hypervisor’s resource allocator or scheduler can lead to unacceptably low availability and performance of a service running inside an enclave. Alternatively, one could allow each guest OS to manage its own enclaves by using a disengaged hypervisor to partition the protected memory region shared by different enclaves among different guests.

We note that Intel SGX is a new development and is not available in current processors. At present, we cannot comprehensively describe how it can be used or modified to achieve our goals. For example, Intel SGX assumes that a single system software will load and evict secure memory pages; this might have to be modified to restrict every guest OS to evicting only pages belonging to its enclaves. It is also unclear if Intel SGX can distinguish between two enclaves (of different VMs) running the same program because it identifies enclaves using only their initial code and data.

SICE [9] creates multiple secure execution environments by isolating sensitive applications using the System Management Mode (SMM) of x86 processors. However, using SMM for secure execution has three important drawbacks: (i) running sensitive tasks that require multiple cores (e.g., VMs) is complicated because it requires communication among different cores running in SMM, which is possible only through the untrusted OS, (ii) SMM does not allow interrupts and therefore requires (untrusted) OS support for access to peripherals, (iii) since SMM is the highest privilege execution mode on x86 systems, malicious guests could misuse this mode to affect the operation of the rest of the system. Our solution does not have these drawbacks, i.e., it enables direct access from secure environments to peripherals, ensures that a malicious MLE can affect only its own VM and allows secure environments to use all the cores available to them.

The TEE framework [14] creates secure execution environments dynamically for guest applications using software-based TPM virtualization. However, TEE requires extensible hypervisor support – the hypervisor hosts the software TPMs and controls the setup and tear-down of secure environments. Such additional functionality usually increases hypervisor complexity and its interaction with VMs resulting in an increased risk of system compromise. Similar arguments hold for other hypervisor-based solutions to protect guest OSes and applications in general [11, 13, 15, 17, 21, 20, 22, 26, 29, 30] and to secure sensitive guest applications against their underlying malicious guest OS [12, 31, 23, 27, 32, 33] in particular. Our solution instead uses a disengaged hypervisor (similar to [36]) that only manages system resources and provides no other run-time services to VMs. Finally, current processor extensions like Intel TXT [1, 4] support secure launch and run-time isolation of sensitive applications on a compromised OS but they support only one such environment even on multi-core processors.
8. CONCLUSION

We presented an extended x86-architecture that enables the creation and co-existence of multiple, parallel secure execution environments. Our architecture can be used in virtualized environments to protect sensitive applications not only against their own underlying guest OS but also other malicious VMs and peripherals. Our solution scales Intel TXT using a disengaged hypervisor and light-weight architectural extensions, namely, an IPI router, per core TXT registers and an HV-TPM. Finally, we showed the feasibility of realizing our architecture by emulating it on QEMU.

9. ACKNOWLEDGMENTS

The research leading to these results was supported, in part, by the Hasler foundation (project number: 09080). The authors also thank Dr. Kari Kostiainen for his feedback which helped improve this work.

10. REFERENCES


APPENDIX

A. TXT-CONFIGURATION REGISTERS

Table 1 summarizes important per-core TXT-configuration registers required in our architecture. The ACM key register is initialized at system boot and cannot be modified even by the hypervisor. Only the hypervisor can write to registers containing information related to the ACM, secure memory (SM) and the TPM. The guest can only write registers to control access to TPM localities 1–2 when it is in secure mode; this functionality is not available (N/A) otherwise.

<table>
<thead>
<tr>
<th>Register</th>
<th>Hypervisor Permissions</th>
<th>Guest OS Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Secure Mode</td>
<td>Normal Mode</td>
</tr>
<tr>
<td>ACM BASE</td>
<td>RV</td>
<td>RO</td>
</tr>
<tr>
<td>ACM SIZE</td>
<td>RV</td>
<td>RO</td>
</tr>
<tr>
<td>ACM KEY</td>
<td>RV</td>
<td>RO</td>
</tr>
<tr>
<td>SM BASE</td>
<td>RV</td>
<td>RO</td>
</tr>
<tr>
<td>SM SIZE</td>
<td>RV</td>
<td>RO</td>
</tr>
<tr>
<td>TPM BASE</td>
<td>RV</td>
<td>RO</td>
</tr>
<tr>
<td>TOGGLE LOC1</td>
<td>-</td>
<td>WO</td>
</tr>
<tr>
<td>TOGGLE LOC2</td>
<td>-</td>
<td>WO</td>
</tr>
<tr>
<td>STATUS</td>
<td>-</td>
<td>RO</td>
</tr>
<tr>
<td>ERROR</td>
<td>-</td>
<td>RO</td>
</tr>
</tbody>
</table>

Table 1: Summary of the access permissions of important TXT-configuration registers with respect to the hypervisor and guest OS.