Exploring the Integration of Memory Management and Trusted Computing

Dartmouth Computer Science Technical Report TR2007-594

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Science

in

Computer Science

by

Nihal A. D'Cunha

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 31st, 2007

Examining Committee:

(chair) Sean W. Smith

Andrew Campbell

M. Douglas McIlroy

Sergey Bratus

Charles K. Barlowe, Ph.D.
Dean of Graduate Studies

Abstract

This thesis addresses vulnerabilities in current Trusted Computing architecture by exploring a design for a better Trusted Platform Module (TPM); one that integrates more closely with the CPU's Memory Management Unit (MMU). We establish that software-based attacks on trusted memory can be carried out undetectably by an adversary on current TCG/TPM implementations. We demonstrate that an attacker with sufficient privileges can compromise the integrity of a TPM-protected system by modifying critical loaded code and static data after measurement has taken place. More specifically, these attacks illustrate the Time Of Check vs. Time of Use (TOCTOU) class of attacks.

We propose to enhance the MMU, enabling it to detect when memory containing trusted code or data is being maliciously modified at run-time. On detection, it should be able to notify the TPM of these modifications. We seek to use the concepts of selective memory immutability as a security tool to harden the MMU, which will result in a more robust TCG/TPM implementation. To substantiate our ideas for this proposed hardware feature, we designed and implemented a software prototype system, which employs the monitoring capabilities of the Xen virtual machine monitor.

We performed a security evaluation of our prototype and validated that it can detect all our software-based TOCTOU attacks. We applied our prototype to verify the integrity of data associated with an application, as well as suggested and implemented ways to prevent unauthorized use of data by associating it with its owner process. Our performance evaluation reveals minimal overhead.

Acknowledgments

I would like to thank my advisor, Professor Sean W. Smith for his continuous support, encouragement, and assistance over the last couple of years. Sean helped me find a research area that was both challenging and interesting, and then provided me with the necessary direction, insight, and suggestions to see this research to conclusion. His experience and ideas have proved to be invaluable to this thesis research. I could never thank him enough for being an exceptional advisor and feel very fortunate to have had an opportunity to work with him.

I am also very grateful to my thesis committee, Andrew Campbell, Doug McIlroy and Sergey Bratus for their guidance, feedback, and willingness to work with me.

Many thanks are also due to John Baek, Alex Iliev, Evan Sparks, Massimiliano Pala, Patrick Tsang, and the other members of the Dartmouth PKI/Trust Laboratory for answering questions and engaging in exhaustive discussions.

Finally, my warmest thanks to my family for their unconditional love, endless support, patience, and for always encouraging my academic pursuits.

This research was supported in part by the National Cyber Security Division of the Department of Homeland Security and by grant CNS-0524695. The views and conclusions do not necessarily represent those of the sponsors.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For many, ”Trusted Computing” is a term that has come to mean that the system will behave as expected, consistently. The *Trusted Computing Group* (TCG) [1] [19] is a consortium that works toward developing and advancing open standards for trusted computing across platforms of multiple types. Their main goals are to increase the trust level of a system by allowing it to be remotely verifiable and to aid users in protecting their sensitive information, such as passwords and keys, from compromise. The core component of the proposal is the *Trusted Platform Module* (TPM) .

A TPM is a micro-controller chip, mounted on the motherboard of a computer, that can be used to provide a range of hardware-based security features to programs that know how to use them. In the last few years, major vendors of computer systems have been shipping machines that have included TPMs, with associated BIOS support.

TPMs provide a hardware-based *root of trust* that can be extended to include associated software. Each link in the chain of trust extends its trust to the subsequent one. A TPM provides internal storage space for storing cryptographic keys and other security critical information. It provides cryptographic functions for encryption/decryption, signing/verifying as well as hardware-based random number generation. TPM functionalities can be used to

[1]Formerly known as the *Trusted Computing Platform Alliance* (TCPA)

attest the initial configuration of the underlying computing platform, as well as to seal and bind data to a specific platform configuration.

However, current TPM-based approaches of attesting to the integrity of critical code and data are not foolproof and can be circumvented[2] by malicious adversaries. While it is hard to tamper with the hardware, it is much easier to subvert the software.

A major *drawback* of the TCG architecture is that it only provides load-time guarantees. Integrity measurements are taken just before the software is loaded into memory, and it is assumed that the loaded in-memory software remains unchanged. However, this is not necessarily true – an adversary can exploit the difference between when software is measured and when it is actually used, to induce run-time vulnerabilities. This is an instance of the *Time Of Check vs. Time of Use* (TOCTOU) class of attacks.

In its current implementation, the TPM holds only static measurements and so these malicious changes will not be reflected in its state. Code that is correct at the time of hashing may be modified by the time of its use. Change-after-hashing is a considerable threat to securing elements in the TCG architecture.

A possible *solution* to this limitation is to have the CPU's *Memory Management Unit* (MMU) modified to work more closely with the TPM. The MMU should be made aware of the software that has been measured at load-time, and should be able to signal to the TPM when the memory corresponding to that loaded software is being changed in malicious ways at run-time.

To explore and test our ideas with regard to this solution to the limitations in the TCG architecture, we use the monitoring capabilities of Xen – an open-source *Virtual Machine Monitor* (VMM) for x86. Our implementation is a software proof-of-concept demonstration of a proposed hardware (MMU) feature.

[2]Evan Sparks and Kwang-Hyun Baek of the Dartmouth PKI/Trust laboratory are investigating power analysis, physical, and software attacks on TPMs.

1.1 Contribution

This thesis makes the following contributions:

- demonstration of a vulnerability in the current TCG/TPM architecture, a set of software-based attacks which exploit this vulnerability, and the realization that such vulnerabilities allow for undetectable tampering of trusted memory;
- a working prototype system with minimal overhead that allows for the detection of such attacks by monitoring trusted page table entries and physical frames of RAM;
- the application of our prototype to verify the integrity of data, such as configuration files, associated with an application;
- a suggestion and implementation of how to associate sensitive data with the process that owns it, so as to prevent unauthorized use of the data; and
- the recommendation of having a closer binding between the MMU and the TPM, so that the detection that is currently done by Xen could be done by the MMU.

1.2 Motivating Example

Certification Authorities (CA) are the keystone of most *Public Key Infrastructures* (PKI). They act as trusted third parties by using their private key to sign certificates. Each certificate binds together a public key and some information (usually identity – such as name, address, organization etc.) about the holder of the corresponding private key. However, CAs are known to be expensive and complicated to install and maintain. The CA-in-a-Box [6] project aimed at addressing these problems by helping small enterprises (such as, a university) deploy PKI in an easy and cost-effective manner.

That project developed tools using open-source software (OpenSSL [13] & OpenCA [12]) and a TPM, allowing enterprises to set-up a hardware secure CA by simply booting a CD.

The TPM is used in that project to enforce the following features of the CA's long-lived private key:

- ensuring that it is only used for authorized operations.
- guaranteeing that it cannot be compromised by an adversary.

The TPM is used to hold the CA's private key[3], as well as to add assurance that the key would only be used when the system was correctly configured as the CA – by wrapping the private key to specified values in a specified subset of the PCRs. The TPM would then decrypt and use (i.e., not release to the outside) that key only when those PCRs have those values.

However, as described earlier, current TCG/TPM implementations suffer from the TOCTOU limitation. Therefore, at run-time, there should be a way of detecting compromise to the platform configuration and thus preventing the CA from using its private key.

We will use this example as the motivating problem for addressing the TOCTOU issues within the TCG architecture.

1.3 Thesis Outline

This thesis is organized as follows: Chapter 2 briefly describes the build blocks needed for our prototype. Chapter 3 gives a detailed explanation of the design and implementation of our prototype system. Chapter 4 describes software-based TOCTOU attacks on TPM measured memory, and ways to detect them using our prototype system. We also show how to apply our prototype to verify the integrity of data associated with an application,

[3]We assume that the CA's private key is implemented as a RSA credential, that is used only within the TPM and is protected by it.

and ways to tie data with the process that owns it. Chapter 5 discusses some areas for possible future work. Chapter 6 examines related work. Finally, Chapter 7 presents a summary and concluding remarks.

Chapter 2

Background

This chapter gives a brief summary of the building blocks of our prototype.

2.1 The TPM

TPMs have shielded locations called *Platform Configuration Registers* (PCR) , each 160-bits long, that hold a digest of integrity measurements. The TPM has to provide at least 16 PCRs in TCG 1.1b [21] and 24 PCRs in TCG 1.2 [20] inside its protected memory.

The contents of a PCR can only be *extended*, and can only be reset by a reboot[1]. Extending a PCR implies that when a new value is to be stored in a particular PCR, it is concatenated with the previously stored PCR value, and a hash of the combined value is taken. The syntax of the operation is:

```
Extend(PCR[i], new_value):  PCR[i] = SHA1(PCR[i], new_value)
```

While this allows a large number of values to be measured and stored without simply overwriting previous measurements, it also prevents malicious users from substituting a known good value for one that indicates tampering.

[1]This concerns only PCR[0-15]. PCR[17-20] (as provided in TCG 1.2) can be reset anytime by Locality 4. “Locality” is a concept that allows the TPM to be aware of which trusted process on the platform is sending it commands. There are six Localities defined (numbers 0 to 4 and Legacy).

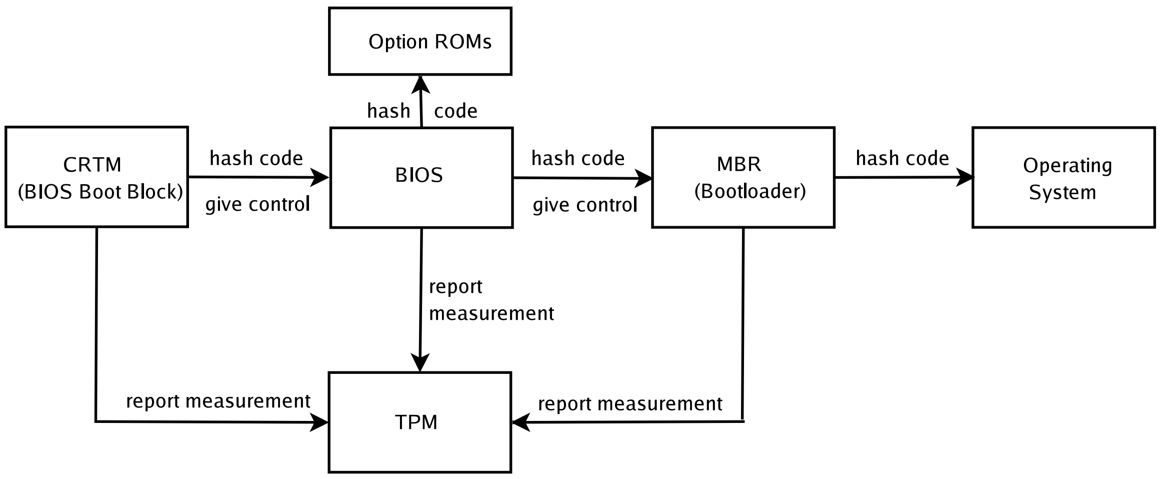

Figure 2.1: Generic Architecture Diagram of the TCG Authenticated Boot Process.

2.1.1 Attestation

On system start-up, a hardware component, called the *Core Root of Trust Measurement* [2] (CRTM), hashes the BIOS and other firmware related configuration files and extends the result into a specific PCR on the TPM. The BIOS proceeds to hash the next piece of software that is executed, which is usually the boot-loader contained in the *Master Boot Record* (MBR) of the system, and extends the result into a specific PCR on the TPM. If the boot-loader is TCG-enabled (e.g., Trusted-Grub [2]) it will continue the chain of trust by extending the hash value of the software it loads (e.g., the operating system) into another PCR. This chain of hash values stored in the PCRs is called the *platform configuration*. The TPM can sign this platform configuration using a protected key, to attest it to a remote challenger. The TCG authenticated boot process is shown in Figure 2.1.

2.1.2 Sealing and Wrapping

To ensure that certain sensitive data on a platform can only be accessed under a specific platform configuration, a sealing mechanism is provided. The platform configuration is defined by the set (or subset) of values contained in the PCRs. Sensitive data is encrypted

[2] This is usually the BIOS boot block, as it is the first piece of code that executes on system start-up.

using a public key[3] and cannot be decrypted unless the system is in the same platform configuration as it were at the time of sealing.

The TPM can also be used to create RSA key pairs (a public key and a private key), where the usage of the private key is bound to a platform configuration. More specifically, such key pairs, called *wrapped* keys, have their private key encrypted by a specified parent key, and are bound to a set of values in a specified subset of the PCRs. The PCR subset is specified at key pair creation time. The TPM will decrypt and use (i.e., not release to the outside) the private key, only if those subset of PCRs have the same values as were present at key pair creation time.

Essentially, secrets are accessible only when the platform is in a defined configuration.

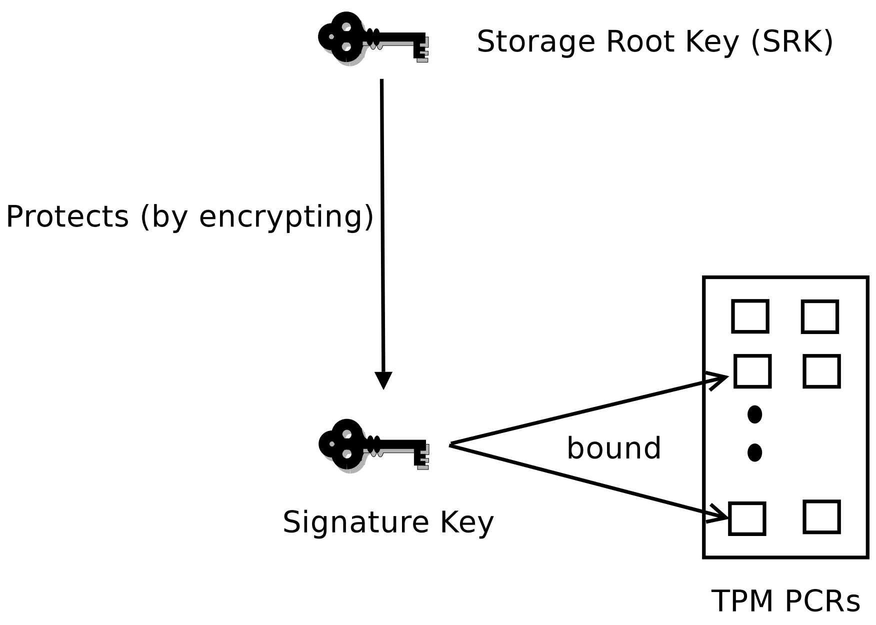

Figure 2.2: TPM Wrapped Key

[3]The corresponding private key is stored encrypted within the TPM.

2.2 x86 Memory Management

Our prototype is implemented on the Intel x86 architecture. We chose this because it is the most popular processor architecture in use today. This section recaps features of the x86 virtual memory architecture that are relevant to understanding the working of our prototype.

Page Tables (PT) are data structures used by the virtual memory subsystem to store mappings between virtual memory addresses and physical addresses. When an instruction attempts to access a virtual memory address, the hardware (MMU) converts that virtual address to a physical address by walking the page tables, and then accesses the physical memory.

Linux on the x86 architecture usually uses a two-level page table structure for virtual address translation, as shown in Figure 2.3. The first level table is called a *Page Global Directory* (PGD) . The x86's `CR3` privileged register holds the physical base address of the currently active process's PGD. The PGD points to second-level PT pages.

With 32-bit virtual addresses, the most significant 10 bits (bit 31 to bit 22) of the address are used as an index into the PGD. The PGD is the size of a page, i.e. 4 Kilobytes, and has 2^{10} entries, each of which is 4 bytes long. Each entry in the PGD contains the physical address of the base of a second-level PT page.

The next 10 bits (bit 21 to bit 12) of the virtual address determine an index into the PT page. The PT page is also the size of a page. The PT page has 2^{10} entries, each of which is 4 bytes. The most significant 20 bits of a *Page Table Entry* (PTE) contain the most significant bits of the physical address of a frame in RAM, while the other 12 bits are used as permission and status bits.

The last 12 bits (bit 11 to bit 0) of the virtual address are used as an offset within the physical frame.

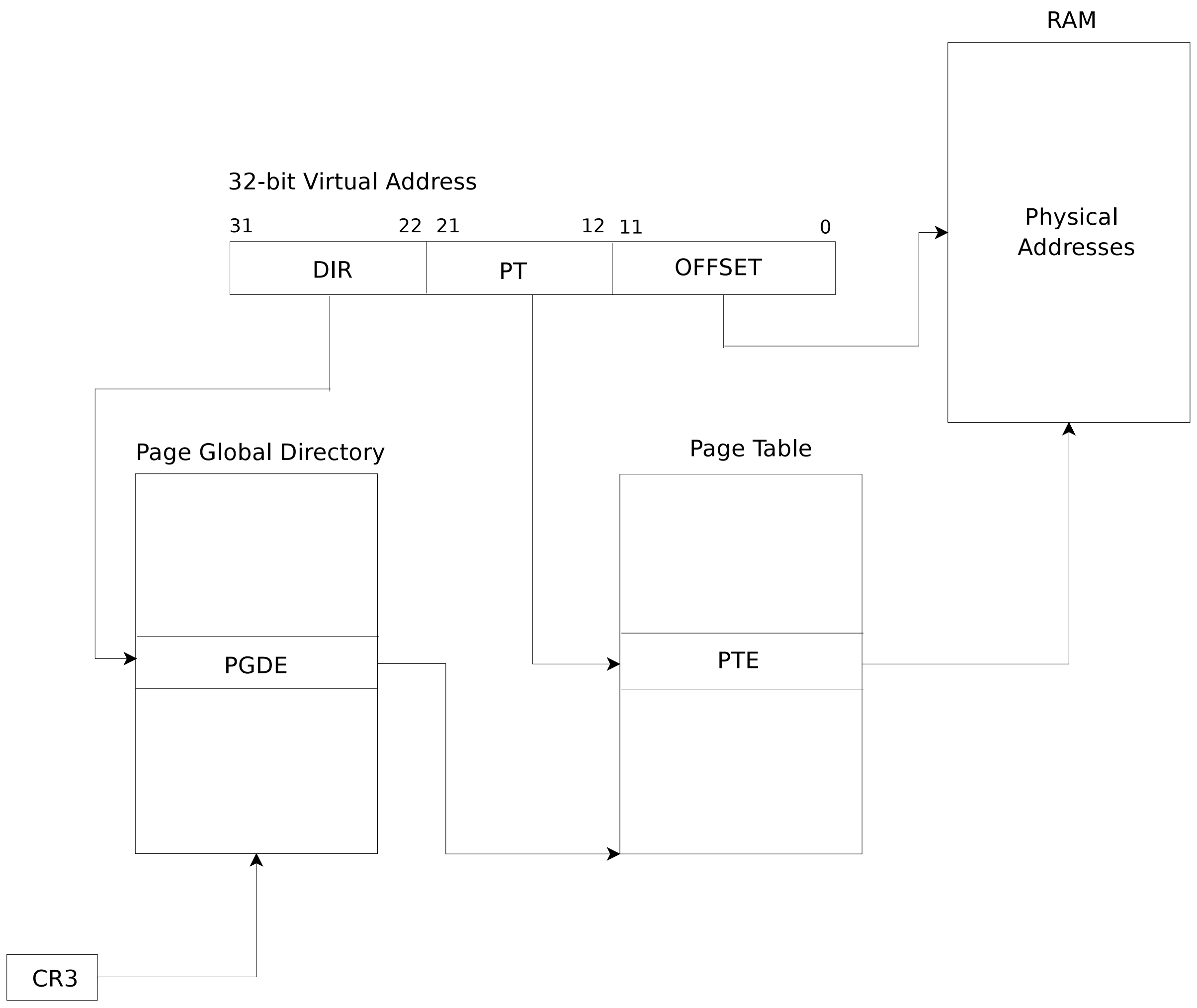

Figure 2.3: Two-level x86 Page Table structure.

2.3 The Xen Hypervisor

Xen [22] is a free and open-source VMM (also called a hypervisor) for x86 that allows for the simultaneous execution of multiple guest operating systems on the same physical hardware. It provides features like secure isolation, resource control, quality-of-service guarantees, and live migration of virtual machines [3]. Operating systems need to be explicitly ported/modified to run on Xen, although compatibility is maintained for user-level applications and libraries. The latest Intel *Vanderpol Technology* (VT) [8] and AMD *Pacifica* [1] processors support hardware-assisted virtualization, which allow Xen to execute unmodified operating system binaries.

The components of a Xen-enabled system, as sketched in Figure 2.4, include the Xen hypervisor (Ring[4] 0), a privileged domain[5] (Ring 1), multiple unprivileged domains (Ring 1) and some user-level control and management tools (Ring 3). The privileged domain is commonly referred to as *Domain-0* , and the unprivileged domains as *Domain-U*. Control software running in the privileged domain has access to a control interface running in the hypervisor that can be used to create, suspend, migrate and terminate unprivileged domains.

Xen replaces the interrupts mechanism from devices with an asynchronous event mechanism. CPU resources are dynamically distributed among domains. To provide strong isolation, main memory is statically partitioned between domains by specifying the initial memory allocation for each domain at the time of their creation. The hypervisor is responsible for managing its own memory as well as that of the domains it hosts. Memory management in Xen is further discussed in Section 2.3.3.

[4]In x86 architecture, there are four privilege levels, called rings, numbered from 0 to 3, with 0 being the most privileged.

[5]"Domain" is Xen terminology for a virtual machine.

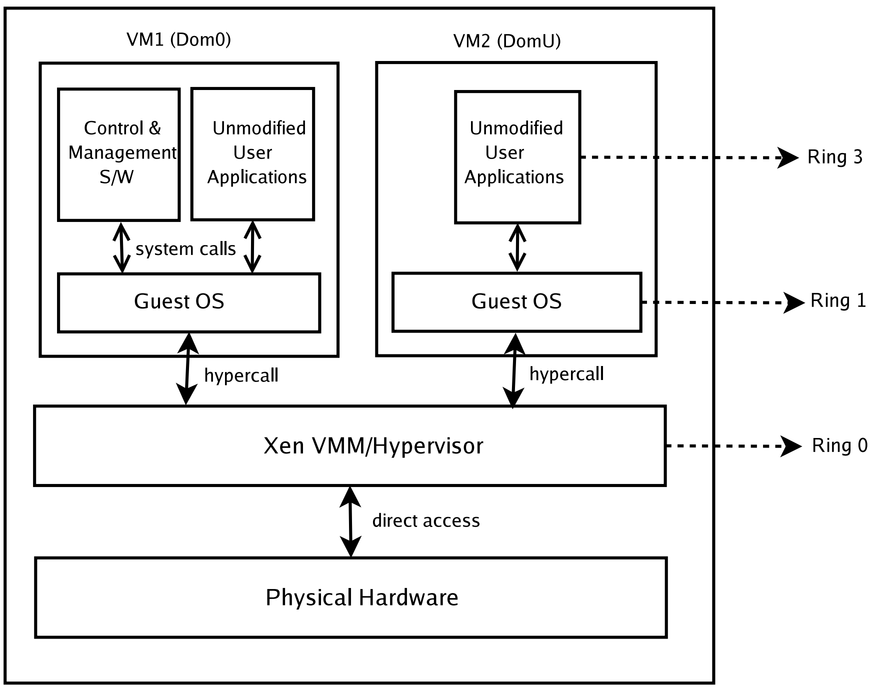

Figure 2.4: Standard layout for a Xen-enabled system hosting a privileged domain (Domain-0) and an unprivileged domain (Domain-U).

2.3.1 Paravirtualization

Xen's high performance virtualization is achieved by employing a technique called *paravirtualization*. Paravirtualization requires operating systems to be aware that they are running in a virtual machine. This virtualization technique presents a software interface to virtual machines, that is similar, but not exactly alike, to that of the underlying hardware. This requires explicit porting of the operating systems to be able to run atop the VMM. The porting essentially entails replacing the guest OS's privileged instructions with appropriate calls (hypercalls, Section 2.3.2) to the VMM.

The hypervisor also maintains a *virtual CPU* for each domain. When a guest OS would like to write to a protected register (which is disallowed since it takes a privileged instruction to do so), the hypervisor writes the value to the corresponding virtual register in the virtual CPU.

2.3.2 Hypercalls

Domains communicate with Xen using software interrupts called hypercalls. Hypercalls are calls from Ring 1 to Ring 0 that allow Guest OSes to request Xen to perform operations on their behalf. This is done in a manner similar to how system calls, from Ring 3 to Ring 0, allow applications to invoke privileged operations in a traditional OS.

On x86/32 machines the instruction required is `INT 0x82`. Currently there are about thirty five hypercalls. We have added three additional hypercalls for our prototype.

2.3.3 Memory Management

Memory management is one of the more important aspects of Xen. As the same physical memory is used by multiple domains, caution has to be taken to preserve isolation and security. Unprivileged domains should be restricted from accessing each other's memory. Guest OSes are responsible for allocating and managing their own hardware page tables, which have to be registered with Xen. Guest OSes are limited to read-only access to their page tables, i.e. they are disallowed from creating writable mappings to frames containing active page tables. Each page table update is intercepted and validated by the hypervisor to ensure that domains only manipulate their own page tables. Domains may batch these operations to make sequential updates more efficient.

Domains are allocated physical memory at creation time by the hypervisor. The memory is not necessarily a contiguous chunk in the physical RAM. However, as most operating systems are not equipped to operate in a fragmented[6] physical address space, Xen introduces a new type of memory, referred to as *pseudo-physical* memory, which is distinct from *machine* memory. Machine memory refers to the physical memory (RAM) installed in the machine, while pseudo-physical memory is a per-VM abstraction, providing a guest OS

[6] In most operating systems physical addresses of kernel symbols, given its linear address, are calculated by subtracting a constant offset. For example, for the Linux Kernel the offset is 0xC0000000.

with the illusion that its memory is a contiguous range of physical pages. The translation between these two addresses is not transparent, and guest OSes need to do this translation for themselves. Xen maintains globally readable tables that provide the mapping between machine and pseudo-physical memory addresses and vice versa.

Xen Page Tables

To make PT updates by the guest OS visible to the hypervisor, all of the pages that are currently part of a PT are mapped read-only in the guest. Guest OSes are only allowed read access to their PTs, with all updates being performed by the hypervisor. Xen currently offers three PT update modes to the guests that it hosts:

Hypercall Mode In this mode, guest OSes have to explicitly make hypercalls (`mmu_update`) to update their PTs. The hypervisor validates the updates being requested. If none of the memory constraints are violated, the PT is updated.

Writable Page Table Mode In this mode, guest OSes are led to believe that their PTEs are directly writable. Guest attempts to write to their PTEs cause a page fault (because the PTs are mapped read-only), which is trapped and emulated by the hypervisor. If none of the memory constraints are violated, the PT is updated.

Shadow Page Table Mode This mode is mainly used by the guest OS when live migration is being performed. In this mode, there are two sets of PTs; the guest OS sees a set of PTs, that are distinct from the ones seen by the hypervisor. The hypervisor sees the actual hardware page tables (pointed to by the `CR3` register). The hypervisor is responsible for propagating changes made to the guest's PTs to the real ones, and vice versa.

2.3.4 Event Channels

The hypervisor controls access to physical devices to ensure isolation, dependability and efficient usage. Access is managed using an event mechanism.

Interrupts are handled within the hypervisor. On receiving an interrupt, the hypervisor issues to the corresponding Domain an asynchronous event notication. Notications are delivered to Domains through event channels. The hypervisor communicates these events with a Domain through a shared memory page.

Domain-0 has direct access all the system hardware. A driver used by Domain-0 to provide access to a virtual device to other domains is called a *backend device driver*, while a driver that a domain uses to control a virtual driver is called a *frontend device driver*. This set-up is shown in Figure 2.5.

A Domain has a set of event channel connection points, called *ports*, which connect to one end of a channel. The opposite end of the channel connects to either a *physical Interrupt Request* (IRQ) , a *virtual IRQ* or a port in an alternate Domain. A physical IRQ is synonymous with the native IRQ of a device, while a virtual IRQ refers to an IRQ managed by the hypervisor. This mechanism allows an event generated by a backend driver to be delivered to the frontend driver as a virtual IRQ.

We created a new Xen virtual IRQ for our prototype to notify Domain-0 of tampering with trusted memory.

2.4 Virtual TPMs

For our prototype we will be using the unprivileged Domain-1 as our test system. Unprivileged VMs cannot access the system's hardware TPM, and so, to provide Domain-1 with TPM access, we need to make use of virtual TPMs.

In a virtual machine-based environment, *Virtual TPM* (vTPM) [4] support provides

TPM functionality to the different virtual machines running on the platform. vTPM support has to be explicitly requested for by the VMs, by having it specified in their creation configuration files. The vTPM interface gives each domain the impression that it is accessing its own exclusive TPM, as if it were a hardware TPM.

The vTPM interface is implemented using a split device driver architecture, as shown in Figure 2.5. The virtual device is provided using two collaborating drivers (as discussed earlier): the frontend device driver, which runs in an unprivileged user Domain (Domain-U), and the backend device driver, which runs in a privileged domain with access to the real device hardware (currently Domain 0). The frontend exports a character device `/dev/tpm0` to user-level applications for communicating with the vTPM. This is consistent with the interface provided if a hardware TPM is available on the system. The backend provides a single interface `/dev/vtpm` where the vTPM has threads waiting for requests from the different domains that have a corresponding frontend.

The frontend driver receives IO requests from its kernel and forwards these onto the backend. The backend receives these IO requests, and is responsible for verifying that they are safe and do not violate isolation guarantees. It then issues these requests to the actual hardware TPM. On IO completion, the backend notifies the frontend, which correspondingly reports IO completion to its own kernel.

The vTPM implementation exists as a user-level process in Domain0. The vTPM manager (`vtpm_managerd`) is used for the creation, deletion, suspension and migration of vTPM instances. It is also responsible for multiplexing requests from the different VMs to their respective vTPM instances. TPM commands are delivered from a Domain-U to the vTPM manager, which dispatches it to a software TPM. The software TPM provides TPM functionality to virtual machines.

In order to distinguish which VM the TPM command was issued from, a 4-byte vTPM instance number is concatenated to the beginning of each TPM command packet by the backend device driver. The instance number pinpoints which unique vTPM instance a VM can interface with.

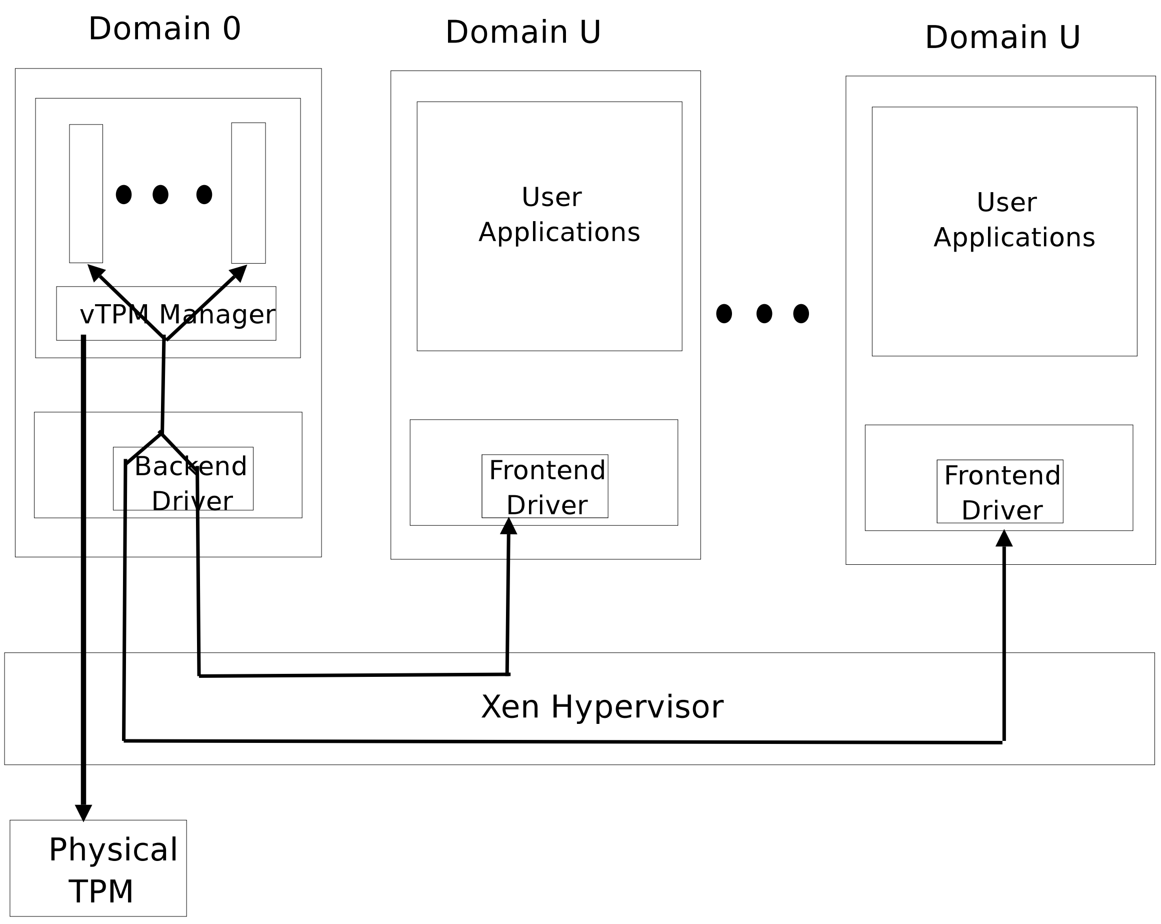

Figure 2.5: vTPM architecture for Xen.

Chapter 3

Design and Implementation

In this chapter, we discuss the major design choices behind this implementation. We will also explore the actual execution of this prototype.

3.1 Reasons for using Xen

The reasons for using a virtual machine-based system, such as Xen, to explore and test our ideas with respect to the TOCTOU issues with the TCG architecture are as follows:

- Xen is being used in this project not for its virtualization features, but as a layer that runs directly below the operating system – similar to the placement of the hardware layer in a non-virtualized environment. Its placement helps us study possible hardware features.
- In a Xen based system, all memory updates trap into the thin hypervisor layer – making it easy to monitor and keep tabs on changing memory.
- Redesigning the MMU hardware is tricky, so we do not want to attempt that until we were certain that the end goal was useful.

- A potentially better alternative to using Xen would have been to use an open-source x86 emulator (such as Bochs [5] or QEMU [14]). However, as of their current implementation, none of these emulators have support for emulating a TPM. Also, the only currently existing software-based TPM emulator [18] does not integrate with any of these. Integrating them would be a major task in itself.
- In the long run, Xen might prove to be an end in itself and we might be able to just use Xen, rather than modifying the MMU, even for real deployment.

3.2 Role of Xen

We could have used Xen in our implementation to address the TCG/TPM TOCTOU limitation via two different approaches, as explained below. We choose to go with the first of these.

3.2.1 Transparent layer

The strategic placement of the thin Xen hypervisor layer between the machine's hardware and the operating system could be seen as a way to prototype changes that could be made in hardware (i.e. in the MMU).

With this approach, the purpose of Xen would be to solely demonstrate a proposed hardware change, and would not be intended to be integrated into the *TCG Software Stack* [1] (TSS) .

Xen's role would be that of a "transparent" layer, manifesting features that would ideally be present in hardware.

[1]The TCG Software Stack is the software supporting the platform's TPM.

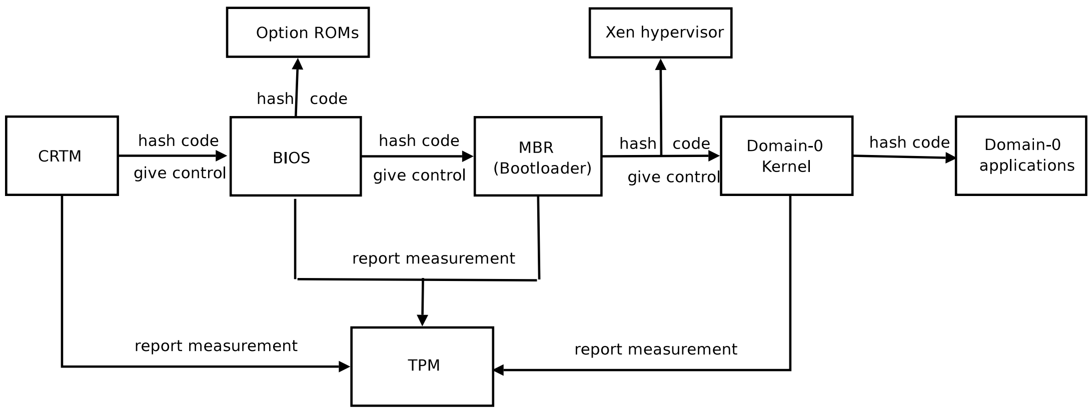

Figure 3.1: Extending the Authenticated Boot Process into the virtualized environment.

3.2.2 Part of the TSS

Alternatively, Xen could be used with the purpose of incorporating it into the TSS. The trusted boot sequence would now include the measurement of the Xen hypervisor executable, the Domain-0 Kernel and applications running in Domain-0, subsequent to the system being booted by a trusted boot-loader. The advantage of this model is that our *Trusted Computing Base* (TCB) will be extended all the way up to the hosting virtual machine environment.

The TCG trust management architecture is currently defined only up to the bootstrap loader, for this implementation we will need to extend the chain of trust up to applications running in Domain-0, as shown in Figure 3.1.

However, as the hypervisor layer is not currently part of the TCG trust management architecture, incorporating it into the TSS will necessitate a revision of the TCG specification.

3.3 TPM Status

Employing Xen to monitor measured memory and update the TPM when that measured memory is altered could have two possible implications on the TPM's status, as explained below. We choose to go with the second implementation.

3.3.1 Dynamic TPM

The idea here is to update the TPM's state every time the measured memory of an application is changed. At all times then, the TPM's state will reflect the current memory configuration of a particular application, and of the system as a whole. This would allow a remote verifier to be aware of the current state of the application in memory, and to make trust judgments based on these presently stored PCR values.

When the hypervisor detects a write to the monitored area of an application's memory, it would invoke a re-measurement of the application in memory. The re-measurement would involve, calculating a SHA1 hash of the critical area of the binary in memory (as opposed to the initial measurement stored in the PCR, which was of the binary image on disk). This remeasured value would be extended to the TPM.

In this case, monitoring of memory writes would be enabled for the entire lifetime of an application, as the TPM state would need to be updated each time the application's measured memory changed.

3.3.2 Static (Tamper-indicating) TPM

The idea here is, to update the TPM's state only the first time that the measured memory of an application is changed. This would allow a remote verifier to easily recognize that the state of the application in memory has changed, and hence detect tampering.

When the hypervisor detects the first write to a critical area of an application's memory, it would *not* invoke a re-measurement of the application; instead, would merely extend the TPM with a random value.

In this case, monitoring of memory writes could be turned off after the first update to the TPM, as that update would be sufficient to indicate tampering. Monitoring subsequent writes (tampering) will not provide any further benefit. This strategy will not have as much of a negative impact on performance as the first approach.

3.4 Implementation Outline

The prototype implementation consists of three primary components: the instrumented Linux Kernel for reporting, the modified Xen hypervisor for monitoring, and the invalidation in the TPM.

The steps below are carried out after the application is measured and extended into the vTPM of the Domain under test.

3.4.1 Reporting

The paravirtualized Kernel of the Domain under test (in our prototype – Domain-1), has been instrumented to allow it to report to the hypervisor – the PTEs, and physical frames that these PTEs map to, of the memory to be monitored, as shown in Figure 3.2

To enable this feature, two new hypercalls have been added to the Xen hypervisor:

- `HYPERVISOR_report_ptes` reports to the hypervisor a list of PTEs that map the memory that needs to be monitored. The PTEs are essentially the entries that map the `.text` section of the binary into memory.

- `HYPERVISOR_report_frames` reports to the hypervisor a list of physical memory addresses that need to be monitored. The addresses are the physical base addresses of each frame that contain memory that needs to be monitored.

These hypercalls make use of a new function that we have added to the Kernel:

- `virt_to_phys()` walks a process's page tables in software to translate virtual addresses to physical addresses. We pass to this function the start and end virtual ad-

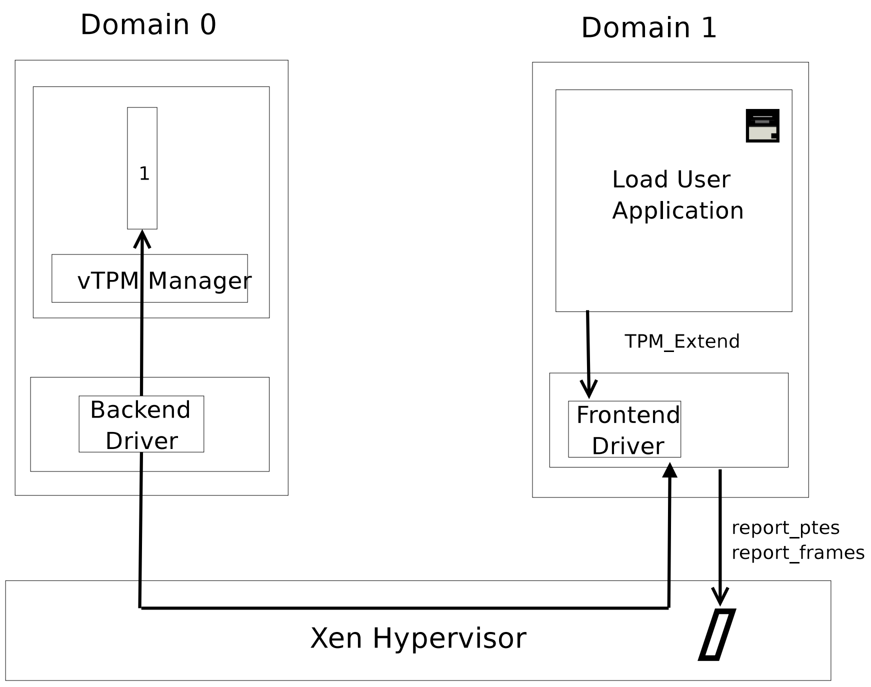

Figure 3.2: Reporting to the hypervisor the PTEs and frames to be monitored.

dresses of the `.text` section of the binary to be monitored. Using the fact that there are 4096 bytes of data on each page[2], it calculates the number of virtual pages spanned by the address range passed to it. It then accesses an address on each page of the range, so as to have it mapped into memory. This step is required to overcome potential problems due to *demand loading*.[3] At this point, the whole of the `.text` section of the binary is mapped into memory. This step however, has performance implications in that it slows down application start-up, as shown in Section 4.4. The function then walks the page tables of the process to translate the virtual addresses to physical addresses (physical base address) of each frame in the range. A data structure containing a list of these addresses is returned to the calling function.

Also, on program exit (normal or abnormal), we need to have the monitored PTES and frame addresses removed from the monitored list. This requirement is fulfilled by

[2]Our experimental system has a 4Kb page size.
[3]Demand loading is a lazy loading technique, where only accessed pages are loaded into memory.

instrumenting the Kernel's `do_exit` function to invoke a new hypercall:

- `HYPERVISOR_report_exit` reports to the hypervisor when an application that is being monitored exits. The hypervisor's monitoring code then deletes the relevant entries from its monitored lists.

3.4.2 Monitoring

Once the required PTEs and frame addresses are passed down to Xen, it will monitor them to detect any modifications made to them, as shown in Figure 3.3.

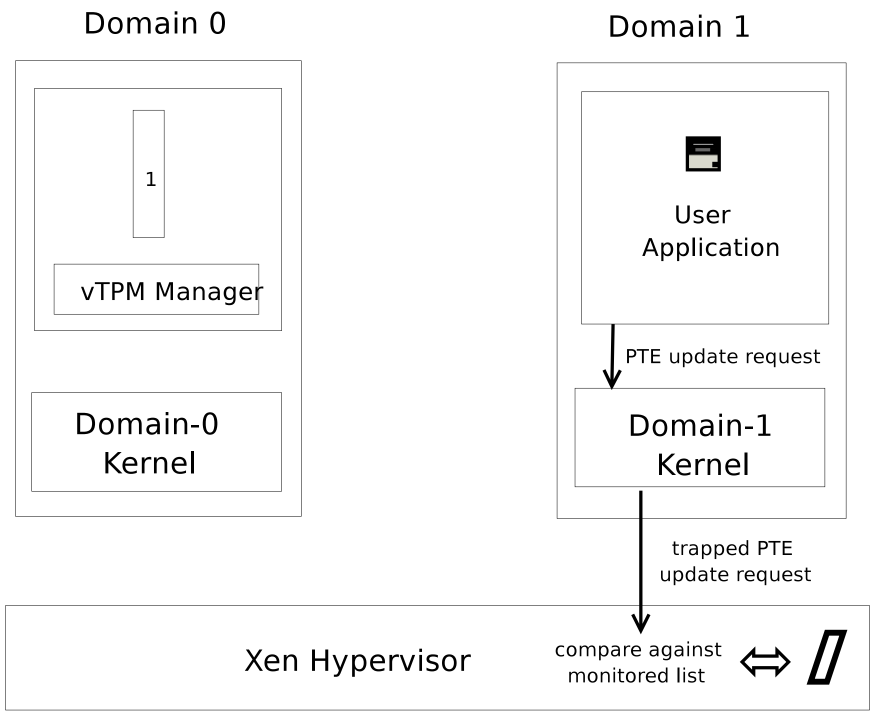

Figure 3.3: Monitoring the reported PTEs and frames for updation.

Writes to these physical memory addresses, or updates to these PTEs to make them map to a different subset of memory pages or make them into writable mappings, will be treated as tampering. The reason for this is that since we are monitoring the read-only code section of an application, neither of the above updates are legitimately required.

The most convenient and reliable method of detecting these types of updates is to 'hook' into Xen's page table updation code. As mentioned earlier, all page table updates in a Xen system go through the hypervisor. This enables us to put in code that can track specific addresses and PTEs.

The default mode of page table updates on our experimental setup is the Writable Page Table mode, as described in Section 2.3.3. In this mode, writes to page table pages are trapped and emulated by the hypervisor, using the `ptwr_emulated_update()` function.

Amongst other parameters, this function receives the address of the PTE that needs to be updated and the new value to be written into it. After doing a few sanity checks, it invokes Xen's `update_l1e()` function to do the actual update.

`update_l1e()` is the function that we instrument to detect tampering. Amongst other parameters, this function receives the old PTE value and the new PTE value that it needs to be updated to. To detect tampering, we perform the following checks:

- **For PTEs:** we check to see if the *old* PTE value passed in is part of our monitored list. If it is, it means that a 'trusted PTE' is being updated to either point to a different set of frames, or to make it writable. The alternate set of frames are considered as potentially malicious frames, and the updated writable permission leaves the corresponding trusted memory open for overwriting with malicious code. This scenario is described in more detail in Section 4.1.2 and Section 4.1.3.

- **For frames:** We first check to see if the *new* PTE value passed in has its writable bit set. If it does, we calculate the physical address of the frame it points to. We then inspect if this physical address is part of our monitored list. If it is, it means that a 'trusted frame' is being mapped writable by this new PTE. The writable mapping, created by this new PTE is interpreted as a means to overwrite the 'trusted frame' with potentially malicious code. This scenario is described in more detail in Section 4.1.1.

Once the tampering is detected in the hypervisor layer, we need to be able to indicate this fact to Domain-0. We do this by creating a new virtual interrupt, `VIRQ_TAMPER`, that a guest OS may receive from Xen. `VIRQ_TAMPER`, is a global virtual *Interrupt Request* (IRQ) , that can be allocated once per guest, and is used in our prototype to indicate tampering with trusted memory.

3.4.3 Invalidating

Once tampering of trusted memory is detected in the hypervisor layer, the Domain under test needs to have its integrity measurements updated. This is done by way of updating the Domain's platform configuration in its virtual TPM, as shown in Figure 3.4.

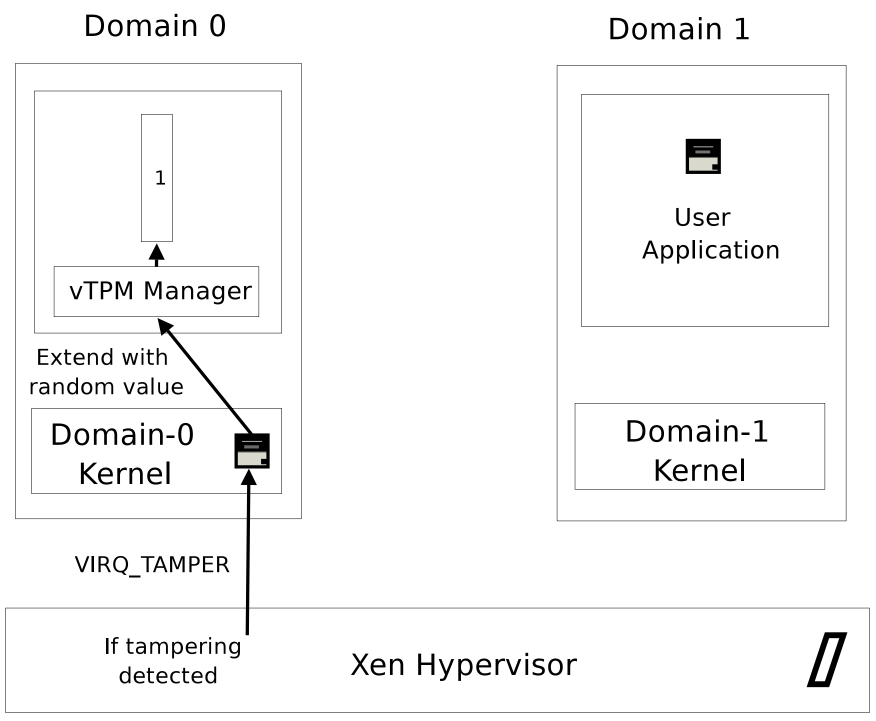

Figure 3.4: Updating PCR in vTPM of Domain-1 on tamper detection.

Our intention is to have the hardware (MMU) do this update. Considering that, in our prototype, the hypervisor together with Domain-0 are playing the role of the hardware, we need to have either of them perform this updation. However, as there are no device drivers present in the hypervisor layer, the hypervisor is unable to interface with the virtual TPM of Domain-1, and so this task is redirected to the privileged Domain-0.

The hypervisor will indicate tampering to Domain-0 by sending a specific virtual interrupt (`VIRQ_TAMPER`) to it. A Linux Kernel Module in Domain-0 will receive this interrupt, and will proceed to extend the concerned PCR in the virtual TPM of Domain-1 with a random value.

We have to make use of the virtual TPM Manager (`vtpm_managerd`) to talk to the virtual TPM of Domain-1. In its current implementation, the virtual TPM manager only delivers TPM commands from unprivileged Domains to the software TPM. Domain-0 is not allowed[4] to directly interface with the software TPM. However, for our prototype, we need Domain-0 to have this ability, and so we have to mislead the virtual TPM Manager into thinking that the TPM commands from Domain-0 are actually originating from Domain-1.

In Domain-0, we construct the required TPM I/O buffers and command sequences required for a `TPM_Extend` to a PCR of Domain-1. As described in Section 2.4, there is a unique instance number associated with each vTPM. To enable Domain-0 to access the vTPM instance of Domain-1, we prepend the above TPM command packets with the instance number associated with Domain-1. This effectively help us forge packets from Domain-1.

[4]Domain-0 is only allowed to access the actual hardware TPM or the software TPM, but not the vTPM instances of other unprivileged domains.

Chapter 4

Experiments and Evaluation

Our prototype on x86, runs on a Xen 3.0.3 virtual machine-based system. Xen's privileged and unprivileged domains run Linux Kernel 2.6.16.29. Our evaluation hardware consists of a 2 GHz Pentium processor with 1.5 GB of RAM. Virtual machines were allocated 128 MB of RAM in this environment. Our machine has an Atmel TPM 1.2 security chip.

4.1 Security Evaluation

In this section, we describe some software-based attacks on current TCG/TPM architecture, and show how our prototype successfully detects all of these attacks. We present three attack scenarios that we implemented, which can be used to subvert measured memory by exploiting the previously mentioned TOCTOU vulnerability.

These attacks, seek to change the `.text`[1] section of a loaded binary. The `.text` section is mapped read-only into memory, and so, is conventionally considered safe from tampering. However, with sufficient (`root`) privileges, an attacker can employ methods to modify the code stored in it by remapping or overwriting it.

The attacking process for all three attacks could be owned by the attacker, or could be

[1]The `.text` section holds the complied code of a program

an arbitrary process. In our scenarios, the attacker takes the form of a kernel module that is inserted into the kernel.

For these attacks the victim process we attacked is *not* the CA. To make things plausible we decided to attack smaller test programs. Nonetheless, the results we obtained would theoretically be the same if the victim process was the CA.

4.1.1 Attack and Defense 1

This attack scenario involves an attacker overwriting the trusted code of a victim process by creating writable page mappings to the victim process's trusted frames from another process, as shown in Figure 4.1.

We carried out this attack by modifying[2] a PTE in our malicious process to map to a physical frame in RAM that the victim process's trusted code was currently mapped to. We modified the PTE to hold the frame address of the victim process page that we wanted to overwrite. The PTE that we chose to update already had its writable bit set, so we did not need to update the permission bits. Using this illegitimate mapping we were able to overwrite a part of the trusted frame with arbitrary data.

It is interesting to note that this attack was possible without having to tamper with any of the victim process's data structures.

The above update to the malicious process's PTs goes through the hypervisor and in effect through our monitoring code. The monitoring code detects that a writable mapping is being created to a subset of the physical frames that it is monitoring. It evaluates this as being a tampering attempt, and raises an alarm.

In the case of the CA, the attacker could overwrite its trusted memory to gain unauthorized use of its private key, such as to sign bogus certificates.

[2]The attack could also be carried out by creating a new PTE that maps to the victim process's frames

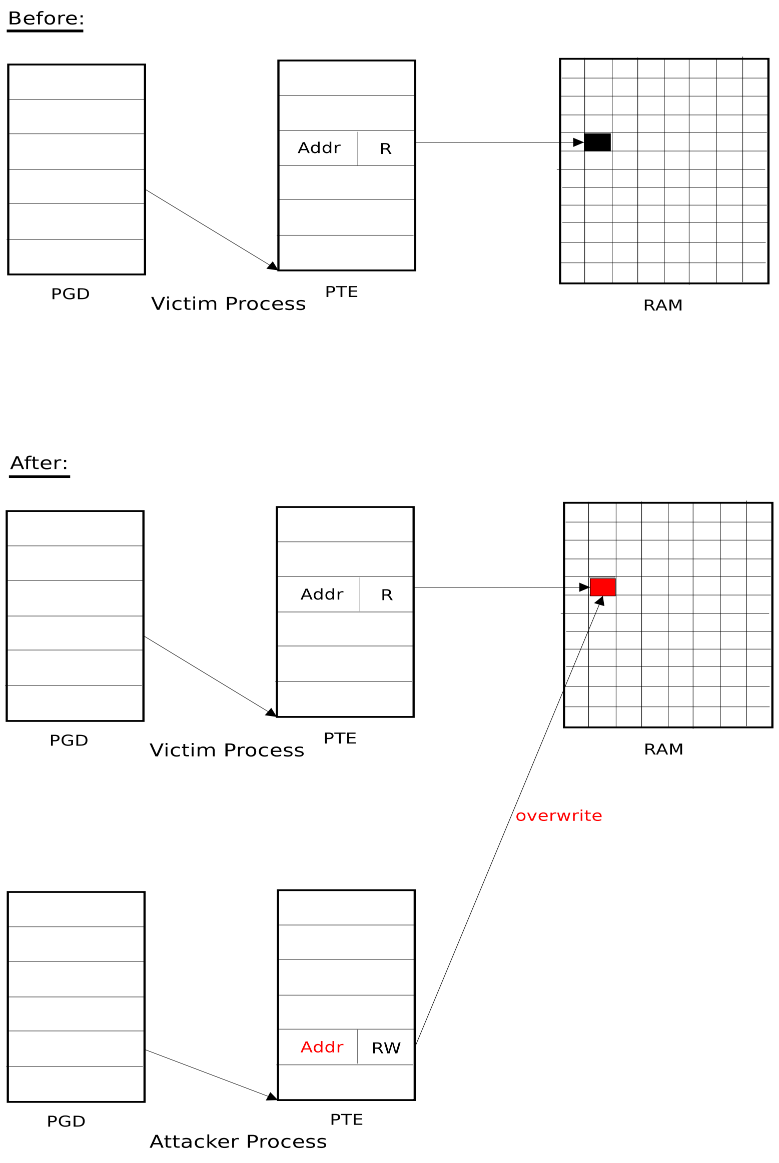

Figure 4.1: Attacker manipulates PTE(s) of his process to map to trusted frames of victim process. Overwrites memory in RAM.

4.1.2 Attack and Defense 2

This attack scenario requires an attacker to modify the trusted code of a victim process by updating the mappings of its `.text` section to point to rogue frames in RAM, as shown in Figure 4.2.

We carried out this attack by using our malicious process to update the address portion of a PTE in the victim process that was mapping its code section. The updated address in the PTE mapped to rogue physical frames in RAM that were part of our malicious process. Due to these updated mappings, the victim process's trusted code was now substituted with the content of our rogue frame.

The above update to the victim process's PTs goes through the hypervisor and in effect through our monitoring code. The monitoring code detects that a subset of its monitored PTEs are being updated to point to different portions of RAM. It evaluates this as being a tampering attempt, and raises an alarm.

In the case of the CA, the attacker could modify its trusted memory to point to rogue frames, which allowed for unauthorized use of its private key, such as to sign bogus certificates.

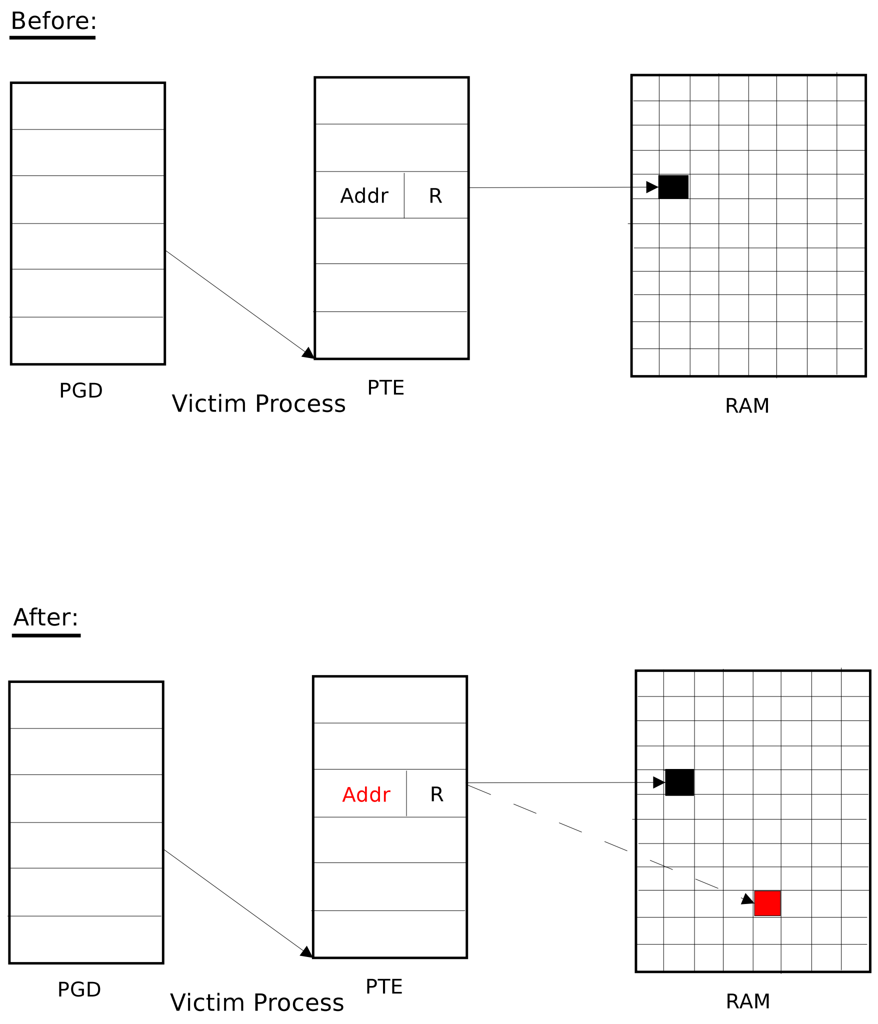

Figure 4.2: Attacker manipulates PTE (address portion) of victim process to map to rogue frames in RAM.

4.1.3 Attack and Defense 3

This attack scenario entails an attacker overwriting the trusted code of a victim process by updating the permission bits of its `.text` section to make them writable, as shown in Figure 4.3.

We carried out this attack by using our malicious process to update the permission bits of a PTE in the victim process that was mapping its code section. We updated the permission bits to set the writable bit making the corresponding mapped frame writable. We used this writable mapping to modify the trusted code in the victim process with arbitrary data.

The above update to the victim process's PTs goes through the hypervisor and in effect through our monitoring code. The monitoring code detects that a subset of its monitored PTEs are being updated to make them writable It evaluates this as being a tampering attempt, and raises an alarm.

In the case of the CA, the attacker could overwrite its trusted memory to gain unauthorized use of its private key, such as to sign bogus certificates.

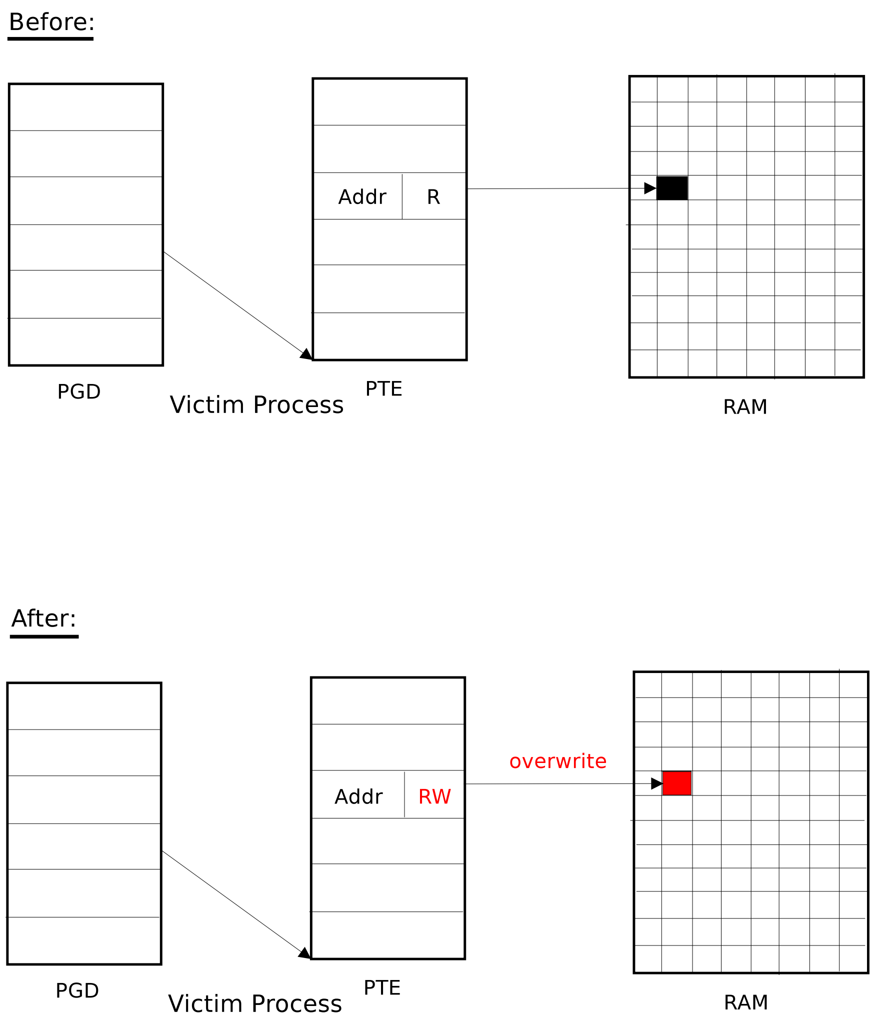

Figure 4.3: Attacker manipulates PTE (permission bits) of victim process to make frames writable. Overwrites memory in RAM.

	Current TPM Implementation		Prototype TPM Implementation	
Attack 1	undetected	allows for unauthorized use of CA's private key	detected, reported & TPM state updated	no unauthorized use of CA's private key
Attack 2	undetected	allows for unauthorized use of CA's private key	detected, reported & TPM state updated	no unauthorized use of CA's private key
Attack 3	undetected	allows for unauthorized use of CA's private key	detected, reported & TPM state updated	no unauthorized use of CA's private key

Table 4.1: Comparison of TPM implementations

4.2 Protecting Data and Secrets

As described in Section 2.1.2, the TPM can be used to generate RSA key-pairs, where the usage of the private key is bound to a particular platform configuration. The platform configuration is defined by the set (or subset) of values contained in the PCRs.

We can use this feature of the TPM to verify the integrity of data, such as a configuration file, associated with an application. Using the CA as an example we want to ensure that the CA will use its private key only when it is running the legitimate CA binary, as well as is correctly configured to be the CA. The initial settings of the CA are setup using a configuration file. Therefore, at CA start-up, it is essential to ensure that we are reading the correct untampered CA configuration file.

To achieve this, in an initial test run, we do both – hash the CA binary, as well as the CA configuration file, and extend both to specific distinct PCRs. We then generate the CA's signing key pair, as a wrapped key pair that is bound to the two aforementioned PCRs.

In production run, when we start-up the CA, we do the same, i.e., have its binary and configuration file extended to the same specific PCRs as before.

<u>Note:</u> If the production run is performed after a system reboot of the test run, the PCRs would have automatically been *zeroized*, and we don't need to do anything special. If, however, the production run is being performed without a system reboot of the test run,

we have to ensure that the CA specific PCRs [3] are zeroized using the `TPM_PCR_Reset` command.

When the CA's encrypted private key is requested to be used for signing, it will only be able to be decrypted and used within the TPM if the following conditions hold:

- The PCR containing the hash of the binary is unchanged.
- The PCR containing the hash of the configuration file is as expected.

These checks guarantee two things. First, the CA process image in memory has not been tampered with since start-up. If it had, our prototype would have detected the tampering and updated the PCR. Second, the CA was started-up with the correct configuration file, and so, is set-up and running as expected.

To prove our hypothesis, we tested out the above ideas using small test programs and found them to be safe and effective.

In a similar vein, we can use the TPM sealing functionality to allow an application to encrypt data, and ensure that the data will only be decrypted when the application is set-up and running as expected.

4.3 Binding Secrets and Data to Processes

In their current specification and implementation, the sealing/wrapping and signing facilities do not bind secrets and data to their 'owner' process. By 'owner' we refer to the application that either encrypted/signed a piece of data or generated a key.

This lack of binding could have security implications. Any running application on the system could potentially unseal the data or unwrap the key, and use it for unauthorized purposes. The TCG specification does have a provision to guard against this – specifying

[3] In this case, the CA specific PCRs chosen will have to be a subset of the resettable PCRs.

a password [4] that will be checked against at the time of unsealing or unwrapping, in conjunction with checking the value in the PCRs. However, if no password is specified, or if it is easily guessable, it leaves open the possibility for unauthorized use.

One way of resolving this problem would be to have a dedicated resettable PCR on the TPM that would be extended with the hash of the binary of the currently executing process on the system. This PCR would be included in the set of PCRs that are used for sealing/wrapping against. As a consequence, the 'owner' process would be required to be currently active on the processor for unsealing/unwrapping to be successful. Every time there is a context-switch (indicated by a change of value in the `CR3` register), the above-mentioned PCR would first be reset, and then be extended with the relevant hash value. This mechanism would prevent a process that is not the 'owner' of a particular sensitive artifact from accessing it.

On systems where the context switching rate is high, having the reset and extend TPM commands carried out on every context switch, might be infeasible and inefficient. We suggest and implement a more feasible resolution to this problem. Note, however, that this will work only for a 'session', i.e., if the application is not re-started. This steps are as follows:

- We implemented new `Create_Key` and `Seal` commands as part of the TSS that include the current `CR3` register value as part of the binding information, along with the specified PCRs. This would cause the sealed/wrapped data to be associated with the currently active process, and help establish an 'owner'.

- Similarly, we implemented new `Unseal` and `Sign` commands that compare the previously stored `CR3` register value against the current value, as well as checks against the PCR values. If they match, it implies that it is the 'owner' of the data that

[4]The password is stored in a wrappedKey data structure associated with the corresponding asymmetric key.

is trying to access it. Hence, the unsealing/signing would be allowed, else it would be disallowed.

4.4 Performance Evaluation

In this section, we evaluate the performance of our prototype system. We measure the additional overhead incurred by our system vs. a native XEN/vTPM system. The results show that our prototype offers enhanced security features with a minimal performance overhead.

Table 4.2 shows the overhead incurred in the Linux Kernel for reporting trusted PTEs/frames to the hypervisor. As can be seen for the first run the overhead is much greater than for subsequent runs. The reason being that for the first run none of the pages of the code section have been mapped into memory as yet. For subsequent runs, if the pages have not been swapped out reporting is much faster. On the average, for the first run it takes between 8-14 microseconds for the reporting of a PTE-frame pair from the Guest OS to the hypervisor.

Binary Name	Size of Binary (Kb)	Number of PTEs/Frames to monitor	Overhead on first run (μs)	Average overhead on subsequent runs (μs)
openssl	392	93	774	23
perl	1036	248	3558	687
aptitude	2248	559	7963	1977
gdb	2312	609	8501	2219

Table 4.2: Reporting Overhead

Table 4.3 shows the overhead incurred in the hypervisor for monitoring trusted PTEs/frames passed to it from the Guest OS. We performed two sets of calculations. The first shows the overhead incurred on loading a binary when the hypervisor's monitored list is empty. The other shows the overhead incurred on loading a binary when there are a thousand PTEs/frames being monitored.

Binary Name	Size of Binary (Kb)	Overhead when monitored list is empty (μs)	Overhead when 1000 PTEs/Frames are being monitored (μs)
openssl	392	289	2799
perl	1036	680	3164
aptitude	2248	1462	3952
gdb	2312	1588	4072

Table 4.3: Monitoring Overhead

These results show that the overhead of our prototype system is almost negligible, making it a very usable and deployable.

Chapter 5

Discussion and Future Work

In this chapter, several issues that deserve more careful discussion and consideration are listed. We also point out potential avenues for further research. Implementation of these ideas will help make this prototype more complete and robust.

Protection: We only protect against physical memory accesses that are resolved by traversing the page tables maintained by the MMU. *Direct Memory Access* (DMA) allows certain hardware subsystems to access system memory independently of the CPU will not be protected against.

Paging to Disk: Pages can be swapped in and out of physical memory and onto the disk. If a page gets swapped out, its present bit will be cleared. On being read back from disk, this page may be allocated into a different frame than it was previously stored in. If a page mapped by a subset of the physical addresses that we are monitoring, gets swapped out, we need to remove those addresses from our monitored list, and update it with a new set of addresses when the page gets swapped in again. Also, when a page gets swapped out, its PTE is updated to record the disk location (swap space) at which the content of the page can be found. We will need to make note of this address, to check that the page that gets swapped back in is the same that was swapped out.

In our initial prototype, we have not implemented any functionality related to swapping, and leave this as future work.

Context Switches: Since we are monitoring physical addresses, which remain unique across context switches between applications (or virtual machines), no special processing is required for this case.

Memory to Monitor: Ideally, besides critical code, it would be beneficial to monitor static read-only data and memory at which important kernel and application data structures, such as interrupt descriptor tables, system call tables and function pointer tables (*Global Offset Table* (GOT) and *Procedure Linkage Table* (PLT)) are stored.

Dynamic TPM: Implementing the dynamic TPM, as described in Section 3.3.1, would be a radical step forward in the way TPMs currently operate. It would enable the TPM to hold the run-time memory configuration of a process, and hence allow for more accurate trust judgments.

Chapter 6

Related Work

6.1 IMA

IBM designed and implemented a TPM-based *Integrity Measurement Architecture* (IMA) to measure the integrity of a Linux system. Their implementation [15] was able to extend the TCG trust measurement architecture from the BIOS all the way up into the application layer.

Integrity measurements are taken as soon as executable content is loaded into the system, but before it is executed. An ordered list of measurements is maintained within the kernel, and the TPM is used to protect the integrity of this list. Remote parties can verify what software stack is loaded by viewing the list, and using the TPM state to ensure that the list has not been tampered with.

6.2 Bear/Enforcer

The Bear/Enforcer [11, 10] project from Dartmouth College developed a *Linux Security Module* (LSM) to help improve integrity of a Linux system.

The Enforcer is a Linux Security Module that calculates the hash of each protected file as it is opened, and compares it to a previously stored value. If a file is found to be modified, Enforcer does some combination of the following: denies access to the file, writes an entry in the system log, panics the system or locks the TCG hardware.

6.3 Copilot

Copilot [9] is a run-time kernel integrity monitor, that uses a separate bus-mastering PCI add-in card to make checks on system memory.

The Copilot monitor, routinely recomputes hashes of the kernel's text, modules, and other critical data structures, and compares them against known good values to detect for any corruption.

6.4 Pioneer

Pioneer [16] provides software-based run-time code attestation.

A trusted entity known as the *verifier* can verify the software stack running on an untrusted platform, by sending a challenge to a self-checking verification function on that platform. The check-sum, returned as the response to the verifier, is checked for correctness as well as if it is returned within the expected time or not. If an adversary tries to manipulate the check-sum computation, the computation time will noticeably increase. This helps the verifier determine if a dynamic root of trust exists on the untrusted platform. The dynamic root of trust is then used to measure further executables that run in an untampered execution environment.

6.5 Bind

BIND [17] is a service that performs fine-grained attestation for establishing a trusted environment for distributed systems.

Rather than attesting to the entire contents of memory, BIND attests only to a critical piece of code, that is about to execute. It narrows the gap between time-of-attestation and time-of-use, by measuring code immediately before it is executed, and protects the execution of the attested code by using a sand-boxing mechanism. It also binds the code attestation with the data that it produces. It requires programmer annotations, and runs within a Secure Kernel that is available in the new *LaGrande Technology* (LT) -style CPUs.

6.6 Terra

Terra [7] is a virtual machine-based platform for trusted computing.

The Terra Trusted VMM (TVMM), partitions a single platform into multiple isolated virtual machines. Using a TVMM, existing OSes and applications can run in an "open-box VM" or a "closed-box VM." The privacy and integrity of the contents of a closed-box VM are protected by the TVMM. The TVMM also allows applications to attest their software stack to remote parties. Attestation is done by decomposing attestable entities into fixed-sized blocks, and computing a separate hash over each block.

Chapter 7

Summary and Conclusion

7.1 Summary

In this research, we described the design and implementation of a prototype system, that will serve as a defense toward the TOCTOU limitation of the TCG/TPM architecture. Once deployed, the integrity measurements of a TPM-protected system, will more reliably be able to be trusted. The TPM will be capable of indicating the tampering of memory containing trusted data.

7.2 Conclusion

The goal of this thesis was, to demonstrate flaws and limitations in the current TCG/TPM architecture. In the course of this research, we made the following contributions:

- We pointed out that current assumptions made about measured memory at run-time is flawed. Specifically, that previously measured memory can be modified at run-time, in a way that is undetectable by the TPM.

- We demonstrated a few software-based TOCTOU attacks on measured memory, and exhibited ways to detect such attacks – by monitoring the relevant PTEs and physical frames of RAM.

- We applied our prototype to verify the integrity of data associated with an application. We also suggested and implemented ways to associate data with the process that owns it, so as to prevent unauthorized use of it.
- Our recommendation is to have a closer binding between the MMU and the TPM, so that the above detection that is currently done by Xen, can be done by the MMU.

Appendix A

Technical Details

This appendix contains various technical details relating to our prototype.

A.1 Virtual machine configuration file

Shown below is the configuration file for virtual machine Domain-1. The Domain is named ”vm01”, and is used as the test machine for our prototype.

```
name="vm01"
kernel="/boot/vmlinuz-2.6.16.29-xen"
root="/dev/sda1"
memory=128
disk=[’file:/vserver/images/vm01.img,sda1,w’,
      ’file:/vserver/images/vm01-swap.img,sda2,w’]
vtpm = [’instance=1, backend=0’]
# network
vif=[ ’mac=00:16:41:AE:50:E4’ ]
dhcp="dhcp"
extra="3"
```

A.2 Attack Overview

Listed here are the steps carried out for each of the attacks and defenses described in Section 4.1:

- Ensure that the virtual TPM backend driver is available in Domain-0. If not statically compiled into the Kernel, the module can be loaded using the command:

  ```
  modprobe tpmbk
  ```

  This will make available a character device:

  ```
  /dev/vtpm
  ```

  which is where the vTPM listens for requests.

- Start the virtual TPM manager daemon in Domain-0. The command to do that is:

  ```
  vtpm_managerd
  ```

- Launch the virtual machine that will function as the test machine. The command for that is:

  ```
  xm create -c /etc/xen/vm01-config.sxp
  ```

  This virtual machine should be TPM-enabled, i.e., the TPM frontend driver must be compiled for its Kernel, and in its configuration file it should specify that it would like to be associated with a vTPM instance using the command line:

  ```
  vtpm = ['instance=<instance number>, backend=<domain id>']
  ```

  as shown in the configuration file above.

- Once the guest machine is started, we need the TPM frontend driver to be activated. If the driver is not compiled into the kernel, it must be loaded using the following command:

```
modprobe tpm_xenu
```

This will make available a character device:

```
/dev/tpm0
```

- In Domain-0, we load a process that waits for an interrupt (`VIRQ_TAMPER`) from the hypervisor. The interrupt indicates tampering of trusted memory.
- We start the victim process in Domain-1. On being loaded into memory, the PTEs/frames mapping its code section are reported to the hypervisor
- We attack the above process by inserting a Kernel module into Domain-1, that manipulates PTEs/frames of the process. This tampering is detected by the hypervisor.
- The hypervisor upcalls into Domain-0 to indicate this tampering. On receiving this upcall, i.e., the interrupt, our 'invalidating process' updates a specific PCR in the virtual TPM of Domain-1 with a random value.
- In Domain-1, the updated PCR values can be seen using the following command:

```
cat /sys/devices/xen/vtpm-0/pcrs
```

A.3 Important files modified

For hooking into the page-table update code

```
xen-3.0.3_0-src/xen/arch/x86/mm.c
```

For adding new hypercalls

```
Xen side:
xen-3.0.3_0-src/xen/arch/x86/x86_32/entry.S
xen-3.0.3_0-src/xen/include/asm-x86/hypercall.h
xen-3.0.3_0-src/xen/arch/x86/mm.c

Linux side:
xen-3.0.3_0-src/linux-2.6.16.29-xen/include/asm-i386/mach-xen/asm/hypercall.h
xen-3.0.3_0-src/linux-2.6.16.29-xen/include/xen/interface/xen.h
```

A.4 Binding secrets and data to processes

Current Implementation

```
1. CreateKey [options] <keyname> <parent keyhandle>

                  ↓

            Encrypted key blob
         (containing private key)


2. Loadkey <parent keyhandle> <encrypted key blob file>

                  ↓

         keyhandle to loaded key


3. Signfile [options] <keyhandle> <input file> <output file>

                  ↓

             signed file
```

New Implementation

```
   1. CreateKey [options] <keyname> <parent keyhandle> <CR3 value>

                              ↓

                     Encrypted key blob
                  (containing private key)
                             +
                            CR3

                              ↓

Sealfile [options] <keyhandle> <input file> <outputfile> <CR3 value>

                              ↓

            New encrypted blob(key blob +  CR3)



  2. LoadSignKey <parent keyhandle> <new encrypted blob> <input file>
               <output file> <CR3 value>

                              ↓

Unsealfile <keyhandle>  <new encrypted blob> <outputfile> <CR3 value>

                              ↓

   <outputfile>( encrypted blob  + stored CR3 ) == <CR3> passed in

         ↓                                  ↓

  If No -- Abort                      If Yes -- Load

                                              ↓

                                             Sign

                                              ↓

                                            EvictKey
```

A.5 Sample source code

Below is the listing of `virt_to_phys()`, which is a function to walk the page-tables in software.

```
static unsigned long virt_to_phys(struct mm_struct *mm, unsigned long start_addr,
       unsigned long end_addr, unsigned long protected_frames[])
{
      pgd_t *pgd;
      pmd_t *pmd;
      pte_t *ptep, pte;

      unsigned pgd_index, pte_index;
      unsigned long ret = 0UL;
      unsigned long addr = start_addr;

      int i = 0;
      int garbage;

      /*iterate through pages*/
      for(addr = start_addr; addr <= end_addr; addr += 4096)
      {
              pgd  = pgd_offset(mm, addr);
              pmd  = pmd_offset(pgd, addr);
              ptep = pte_offset_map(pmd, addr);

              //access an address on each page
              garbage = *((int*)addr);

              if(pte_present(*ptep)) {
                 pte = *ptep;
                 ret =   pte.pte_low & ((ulong) 0xFFFFF000);

                 protected_frames[i++] = ret;
              }
    }
    /* print this to prevent the compiler from optimizing it out*/
    printk("%c\n",garbage);
    return protected_frames;
}
```

Bibliography

[1] AMD. Virtualization Solutions. http://www.amdboard.com/pacifica.html.

[2] Applied Data Security at University of Bochum. Trustedgrub. http://www.prosec.rub.de/trusted_grub.html.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM Press.

[4] Stefan Berger, Ramn Cceres, Kenneth Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM – Virtualizing the Trusted Platform Module. In *15th Usenix Security Symposium*, pages 305–320, 2006.

[5] Bochs. IA-32 Emulator Project. http://bochs.sourceforge.net/.

[6] Mark Franklin, Kevin Mitcham, Sean W. Smith, Joshua Stabiner, and Omen Wild. CA-in-a-Box. In *EuroPKI*. Springer.

[7] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[8] Intel. Virtualization Technology (VT). http://www.intel.com/technology/computing/vptech/.

[9] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th USENIX Security Symposium*, pages 179–194, 2004.

[10] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love The Bear. Technical Report TR2003-476, Dartmouth College, Computer Science, Hanover, NH, December 2003.

[11] John Marchesini, Sean W. Smith, Omen Wild, Joshua Stabiner, and Alex Barsamian. Open-Source Applications of TCPA Hardware. In *ACSAC*, pages 294–303, 2004.

[12] OpenCA Project. The OpenCA PKI Development Project. http://www.openca.org/.

[13] OpenSSL Project. The Open Source toolkit for SSL/TLS. http://www.openssl.org.

[14] QEMU. Open Source Processor Emulator. http://www.qemu.com/.

[15] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, pages 223–238, 2004.

[16] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2005. ACM Press.

[17] Elaine Shi, Adrian Perrig, and Leendert van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *IEEE Symposium on Security and Privacy*, pages 154–168, 2005.

[18] Mario Strasser. Software-based TPM Emulator for Linux. *Department of Computer Science. Swiss Federal Institute of Technology Zurich*, 2004.

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

[20] Trusted Computing Group. TPM Main Specification Part 1: Design Principles. Feb 2005.

[21] Trusted Computing Platform Alliance (TCPA). Main Specification, Version 1.1b. https://www.trustedcomputinggroup.org/specs/TPM/TCPA_Main_TCG_Architecture_v1_1b.pdf, Feb 2002.

[22] Xen. Virtual Machine Monitor. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

Index