1-1-2013

Attacking Disk Storage Using Hypervisor-Based Malware

Jaron W. Martin

Attacking disk storage using hypervisor-based malware

By

Jaron W. Martin

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2013

Attacking disk storage using hypervisor-based malware

By

Jaron W. Martin

Approved:

Yoginder S. Dandass Associate Professor of Computer Science and Engineering (Major Professor)	David A. Dampier Professor of Computer Science and Engineering (Committee Member)
Thomas H. Morris Assistant Professor of Electrical and Computer Engineering (Committee Member)	Edward B. Allen Associate Professor of Computer Science and Engineering (Graduate Coordinator)
Sarah A. Rajala Dean of the Bagley College of Engineering	

Name: Jaron W. Martin

Date of Degree: May 10, 2013

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Yoginder Dandass

Title of Study: Attacking disk storage using hypervisor-based malware

Pages in Study: 46

Candidate for Degree of Master of Science

Malware detection is typically performed using either software scanners running inside the operating system or external devices designed to validate the integrity of the kernel. This thesis proposes a hypervisor-based malware that compromises the system by targeting the hard disk drive and leaving the kernel unmodified. The hypervisor is able to issue read and write commands to the disk while actively hiding these actions from the operating system and any detection software therein. Additionally, the hypervisor's presence has minimal impact on the performance of the system. The ability to perform these commands compromises the confidentiality, integrity, and availability of the stored data. As a result, this thesis has widespread implications affecting personal, corporate, and government users alike.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

Computer security is a prominent area of interest that affects personal, industry, and government users alike. Traditional threats to security include viruses, trojans, worms, spyware, and rootkits. However, malicious hardware and malicious hypervisors can also pose substantial risk in computer systems. Hypervisors are designed to facilitate virtualization by handling context switching between multiple operating systems on the same hardware. While software typically runs at the user level (ring 3) or at the kernel level (ring 0), hypervisors run at ring -1, an even higher privilege level than the operating systems it controls. As such, these operating systems, called guests, are typically unaware that a hypervisor is even present on the system. This high privilege level and transparent execution have the potential to make malicious hypervisors a significant threat to security.

1.2 Virtualization

Virtualization is a method by which multiple guest operating systems can run simultaneously on the same hardware. Host software, also called the hypervisor, communicates directly with the hardware and presents virtualized hardware resources to each guest Operating System (OS). The guest OS assumes that it is running on its own

dedicated hardware while in reality the hardware it sees is being emulated or controlled by the host.

Hypervisors lie dormant while the guest OS executes, until one of a list of pre-defined wakeup condition occurs. The occurrence of a wakeup condition results in a *VM Exit* operation in the CPU. A VM Exit operation causes execution to switch from the guest OS to the hypervisor. The hypervisor saves the state of the guest OS and then performs any desired actions before restoring control to the same OS or giving control to a different OS.

There are two broad categories of hypervisors, labeled *Type 1* and *Type 2*. Type 1 hypervisors, also known as "bare-metal" hypervisors, run directly on the physical hardware. This category of hypervisors results in guests with near-native performance, where the execution overhead is largely limited to when the hypervisor is active. Type 2 hypervisors, also known as "hosted" hypervisors, run inside a host operating system. Type 2 hypervisors result in a greater performance degradation because the hypervisor must interact with the host OS in order to execute.

1.3 Hardware-Assisted Virtualization

The increase in adoption of virtualization has prompted hardware manufactures to incorporate virtualization technology at the processor level in order to increase performance and simplify hypervisor design. Two examples are Intel's VT-x [9] and AMD's AMD-V [1] technologies. Hardware assistance makes virtualization more efficient by reducing the complexity required to implement a hypervisor and providing native hardware instructions for common hypervisor tasks.

This hardware-assisted virtualization has proved to be an additional avenue for exploitation. Using hardware-assisted virtualization, the target operating system can be converted into a guest OS running atop a malicious hypervisor. This effectively allows the hypervisor to become a man-in-the-middle between the OS and the hardware without the OS knowing anything has changed. The hypervisor then has the ability to intercept and modify any access to the hardware from the guest OS (*e.g.*, access to memory, I/O devices, and timers) allowing the hypervisor to perturb normal execution of the OS. The hypervisor can use this capability to deny access to files and memory locations as well as modify the response to CPU timer queries that can be used to detect the hypervisor's presence.

1.4 Project Overview

The goal of this thesis is to utilize a hypervisor to discretely read and modify the contents of an attached hard disk drive (HDD) while preventing the target OS from detecting these actions. Additionally, this goal should be met with minimal stability and performance impact on the system in order to decrease the possibility of detection by a smart host bus adapter (HBA) or the host OS. This thesis implements a Type 1 (bare-metal) hypervisor which converts the OS running on the system into a guest OS running atop the hypervisor. The hypervisor has the capability of reading and modifying data on an attached HDD without the knowledge of the target OS. Control of the HDD is achieved by utilizing the HBA to issue commands using the Advanced Host Controller Interface (AHCI) protocol. These commands are hidden from the OS by using Extended Page Tables (EPT), a hardware-assisted virtualization technology, in order to modify the OS's view of the HBA's hardware registers. This capability violates the confidentiality,

integrity, and availability of the stored data by allowing data to be secretly read and modified by the attacker.

1.5 Hypothesis

The hypervisor-based disk drive attacking malware will be able to compromise the confidentiality, integrity, and availability of data on a system while concealing its activities from the OS, adding an overhead of less than 5%, and causing no disruptions (*e.g.*, timeouts and command collisions) in the OS's disk I/O operations.

CHAPTER II

RELATED WORK

2.1 Software Solutions

Numerous methods have been proposed to detect malicious software. While software solutions can reliably detect user-mode malware, these solutions become less reliable as the malware's privilege level increases. User-mode software solutions rely on the kernel to perform actions on their behalf, and kernel-level malware intercepts and modifies these requests. Kernel-mode software solutions, while running at a higher privilege level, are still vulnerable to modification by kernel-level malware.

2.2 Hardware Solutions

More reliable solutions utilize a co-processor and are often implemented as add-on PCI cards[14], secondary systems, or a combination of the two[10,3]. These devices use Direct Memory Access (DMA) to view the contents of main memory without using the OS, making them more reliable when dealing with an infected system.

2.2.1 Copilot

Copilot is a tool that utilizes a PCI card to access the system's memory using DMA[10]. It monitors two important sections of (Linux) kernel data memory commonly modified by rootkits: the kernel or LKM text section as well as tables containing function jump pointers. Copilot calculates the "known good" MD5 hash of each of these

structures in the clean system and periodically re-calculates the hash to ensure the structure has not changed.

Copilot was designed to assess versions 2.4 and 2.6 of the Linux kernel and relies on two specific characteristics: the kernel memory cannot be paged, and the kernel's virtual address space must be linearly mapped. Additionally, Copilot is unable to scan dynamically-loaded kernel modules [14] and, like all DMA solutions, is vulnerable to DMA redirection via the north bridge [11]. As Copilot is designed to verify kernel integrity, it does not detect the presence of a malicious hypervisor.

2.2.2 Hierarchical Trust Management

Hierarchical Trust Management is a solution similar to Copilot, except it is designed to verify all software running on the system rather than specific kernel structures[14]. A PCI card containing the 'SecCore' periodically verifies 'SecISR,' a kernel-level interrupt service routine. SecISR then verifies two additional software components, a kernel scanner and a software scanner. These scanners, in turn, will verify the kernel and any critical applications. The process then continues to form a hierarchical chain of trust that verifies the system.

A second component, 'SecIO', consists of an input device and a display device. This allows users with physical access to the system to approve modifications such as those made after a software update.

The verification process compares the current hash to the hash from a known good state. The known-good hash can either be calculated when the system is clean or can be signed and provided by the trusted software provider.

This solution assumes that any attacks on the system are network-based, and thus the attacker does not have physical system access. Additionally, it is assumed that a user will not approve a malicious modification. This solution is designed to scan Linux ELF files; therefore, modifications are required for use on Windows systems, and hypervisors are not detected.

2.2.3 Gibraltar

Gibraltar uses a secure coprocessor (second system), called the *observer*, to monitor the physical memory of the target system via two PCI network cards, one installed in each system[3]. The network card on the target system is modified to perform DMA requests at the request of the observer. Gibraltar monitors kernel data structures starting with those at fixed locations determined by analyzing the source code (system.map). It then recursively follows pointers in these structures to find additional structures. The format of each type of structure is also determined by analyzing the source code.

Once the structures are found, Gibraltar utilizes a tool called Daikon in order to find invariants, or values that should not change during normal activity. Once these invariants are identified, Gibraltar periodically checks the invariants and alerts if a change is found.

While Gibraltar was able to detect all of the creator's sample rootkits, it also alerted 85 times during a 42-minute benign workload, meaning further research is needed to determine which invariants are valid. Also, in the situation where a structure is de-allocated prior to Gibraltar (asynchronously) accessing it, it is possible for stale pointers to be followed. System updates require Gibraltar to re-scan for invariants, kernel changes

would require re-analysis of the source code if data structures changed, and the requirement for access to kernel source code makes this solution infeasible for Windows systems. Also, as Gibraltar is designed to scan kernel structures, hypervisors remain undetected.

2.2.4 Kernel Patch Protection

Upon a kernel API call, the kernel performs a lookup in a table to determine the memory address of the function being called[8]. Kernel patching is the act of modifying this pointer in order to redirect execution of API calls, often called 'hooking'. This is legitimately used by virus scanners to scan a file prior to execution, but can also be utilized by malicious software. Kernel patching can result in system instability if the third-party code contains errors. Also, patching the kernel can slow the system, as additional code is executed prior to the API call. In an effort to improve system security and reliability, Microsoft began integrating Kernel Patch Protection (KPP) in versions of Windows to prevent kernel patching.

Microsoft's Kernel Patch Protection (KPP), also known as PatchGuard, is a feature of Microsoft's 64-bit operating systems that aims to detect third-party kernel modifications. Upon detection of a 3rd-party patch, KPP causes a Bug Check, which results in a system shutdown. KPP helps prevent malicious software from hooking the kernel; however, it does not protect against hypervisors.

2.2.5 Attacking Computer Security Using Peripheral Device Drivers

A thesis by M. King uses a Windows device driver (WDD) to emulate a malicious Network Interface Card (NIC)[7]. The idea is that the NIC can receive commands via the

network and execute those commands without the knowledge of the OS. The WDD is able to issue read/write commands to an attached Serial ATA (SATA) HDD configured to use the Advanced Host Controller Interface (AHCI) by modifying register values in the Host Bus Adapter (HBA).

When using AHCI, commands are issued by writing to a Command List that contains up to 32 entries. King observed that these entries are typically written in a round-robin ordering by the OS. Therefore, to minimize the probability of a collision, the WDD writes to a command slot previous to that used by most recently issued command.

Several issues were present in King's thesis. One issue is that during periods in which the OS rapidly issues disk requests log entries are created by the OS indicating disk timeouts. These log entries alert the user that that something is wrong on the system. The timeouts result from the WDD issuing a command prior to a rapid number of commands being issued by the OS causing the WDD's command to be overwritten. Second, the method used to issue commands results in abnormal patterns in HBA registers (PxCI and PxSACT) and the Command List, which can be detected either by intelligent HBA hardware or by a software scanner that watches for writes occurring in an unusual order, making detection relatively simple. Third, a delay introduced to reduce collisions significantly decreased data throughput of the WDD to the point where compromising 100GB of data takes in excess of three years.

2.2.6 Blue Pill

Blue Pill is a thin hypervisor-based malware presented by J. Rutkowska [12]. Blue Pill is designed to convert the running OS into a guest in order to take control of its execution. While the initial version of Blue Pill targets AMD systems, the concepts are

similar to Intel's VT-x. Blue Pill inspired many subsequent works describing hypervisor-based malware solutions.

CHAPTER III

APPROACH

3.1 Introduction

This thesis implements a hypervisor with the capability of compromising the data stored on a hard disk drive (HDD). The hypervisor is difficult to detect because it can hide the true values of important registers on the Host Bus Adapter (HBA) (*e.g.*, PxCI and PxSACT), as well as other critical disk device command data structures, from the OS and OS-level anomaly detection software. This prevents any software from scanning for unusual behavior regarding these values. The hypervisor also offers additional stability by preventing collisions of disk drive commands between itself and the OS and increased efficiency by only waking up when the hypervisor needs to issue commands or when the critical registers and command data structures are modified by the target OS.

This thesis targets a system running 64-bit Microsoft Windows 7 Professional on an Intel i7 processor with VT-x and Extended Page Table (EPT) capabilities. The target hard drive is a Serial ATA (SATA) disk that communicates with the system platform via the Advanced Host Controller Interface (AHCI) protocol. The disk controller (HBA) connects to the system platform via the PCI-e bus.

3.2 Implementation Details

This section provides descriptions of various technologies and how they are used in this thesis.

3.2.1 Hardware-Assisted Virtualization

Virtualization is a method by which multiple guest operating systems can run simultaneously on the same hardware. This is achieved by utilizing a hypervisor to manage each guest OS's access to the hardware resources. The hypervisor can either run directly on the physical hardware or inside a host OS. This thesis utilizes the Type 1 (native) hypervisor implementation in order to convert the target OS into its guest without relying on an additional underlying OS.

3.2.1.1 Hypervisor Implementation

This thesis implements and expands the hypervisor described by Y. Dandass, D. Dampier, and S. Shannon[4]. A Windows device driver (WDD) is used to load the hypervisor and convert the target OS into a guest by setting up the hypervisor control structures and invoking the *VM Launch* operation on the CPU. VM Launch is an instruction in the IA-32 architecture that transfers execution from the hypervisor to the guest virtual machine (VM). In addition to the features implemented by D. Dandass, *et al*., the hypervisor in this thesis adds support for EPT in order to control and alter the OS's view of the HBA registers used to issue disk operations.

3.2.1.2 Intel VT-x

VT-x is Intel's virtualization technology supported by modern IA-32 processors. This technology provides hardware extensions in order to improve the efficiency and

reduce the complexity of hypervisor implementation. Processors supporting VT-x have native support for suspending guest OSs and returning control to the hypervisor when specified actions occur such as a guest's interaction with hardware. VT-x also provides the capability to convert a host OS into a guest OS. This functionality allows a hypervisor to be installed underneath the running OS.

Converting the running OS to a guest involves several steps. First, memory pages for a Virtual Machine Control Structure (VMCS) and a VMXON region are allocated. Second, important CPU settings are copied into the VMCS in order to reflect the current OS setup. Third, the VM is launched, continuing execution of the OS as a guest. At this point the OS continues running, unaware that it now runs atop a hypervisor.

3.2.2 PCI Express

PCI Express (PCI-e) is the peripheral communications bus used in modern computing systems. This bus connects peripherals such as video cards and disk controllers (HBAs) to the CPU via the root complex. The HBA requests registers to be mapped into platform memory space. Software issues commands to the HBA by writing and reading these registers. The firmware (BIOS) on bootup determines the location where the registers appear in the platform's physical memory space. System software (*i.e.*, the device driver) reads PCI-e configuration registers to determine the address of the HBA registers. PCI-e also provides DMA capabilities to the HBA, allowing the HBA to access host memory in response to commands issued to the HBA by software.

3.2.3 SATA

The HBA communicates with the CPU using the PCI-e bus. The HDD is connected to the HBA using the Serial ATA (SATA) interface.

The SATA protocol utilizes packets to exchange information between the host and device [13]. The payload of these packets is called the Frame Information Structure (FIS). There are different types of FISs, each with a defined structure. In order to send a command to a SATA device, the system platform first constructs a Register Host-to-Device FIS and sends it to the device. Depending on the type of command, the device responds with one or more FISs. For a read operation data FISs are returned over the SATA bus, which the HBA transfers to host memory using DMA. For a write operation a status FIS is returned indicating that the device is ready to receive data. The HBA then transfers data to the device by using DMA transactions over the PCI-e bus to send host data.

3.2.4 AHCI

The Advanced Host Controller Interface (AHCI) is an implementation-independent specification to describe the interaction between a system platform and a SATA device. When the computer system boots, each SATA device is detected and a set of registers inside the HBA is mapped into memory for each SATA device. These registers consist of configuration registers and memory registers. Of interest to this thesis is a configuration register called the AHCI Base Address Register (ABAR). This register contains a pointer to a memory segment that consists of, among other things, 32 port registers. Figure 3.1 below shows ABAR and the memory segment it points to, including the Port registers [6].

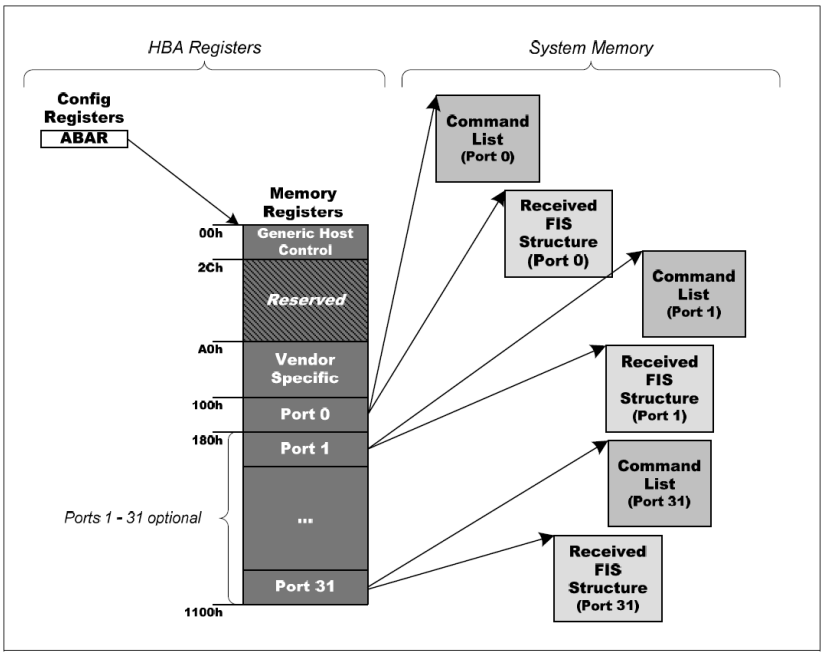

Figure 3.1 HBA Port Registers

Each port register represents a SATA device. Only the port corresponding to the target HDD is important for this thesis. Each port register points to two structures: a Received FIS structure and a Command List structure. Figure 3.2 shows the Received FIS structure, while Figure 3.3 and Figure 3.4 show the Command List structure [6].

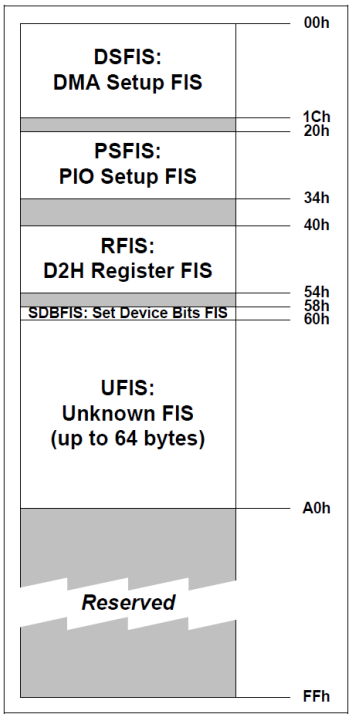

Figure 3.2 Received FIS Structure

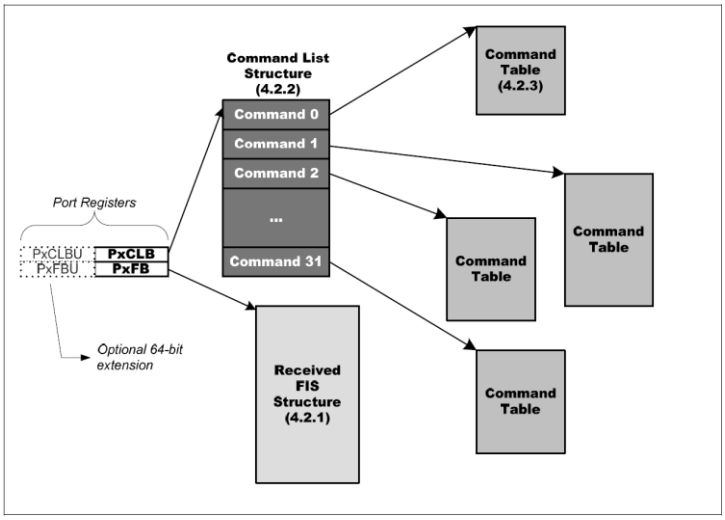

Figure 3.3 Command List Structure

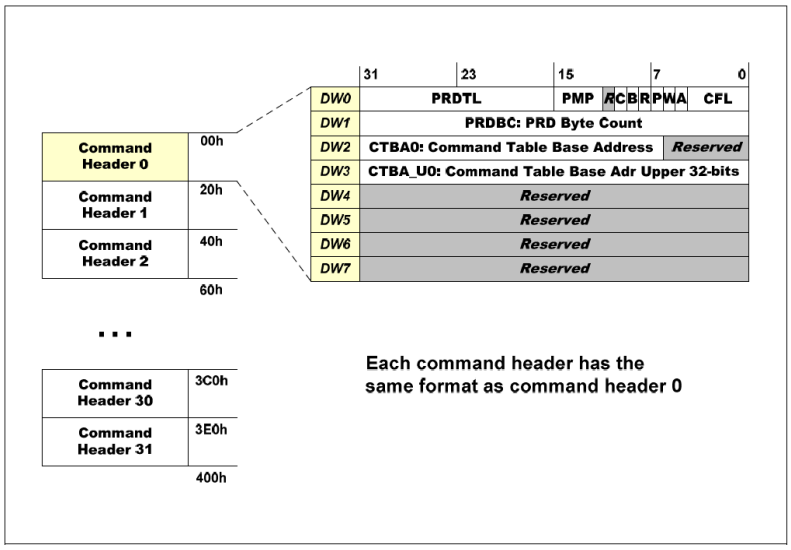

Figure 3.4 Command List Header

The Received FIS structure stores FISs that have been received that correspond to the particular port. This includes a copy of the DMA Setup FIS (DSFIS), PIO Setup FIS (PSFIS), D2H Register FIS (RFIS), and the Set Device Bits FIS (SDBFIS).

The Command List structure is an array that contains 32 command headers (numbered 0 through 31). Each command header points to a Command Table structure, shown in Figure 3.5 below [6].

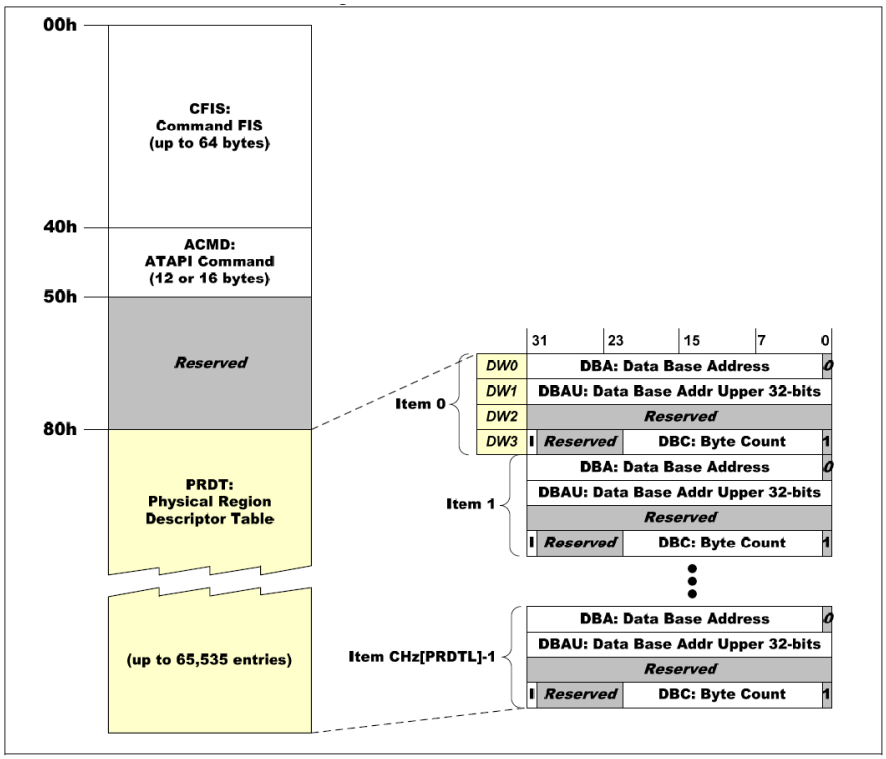

Figure 3.5 Command Table Structure

The Command Table structure represents a command to be sent to the HDD. It contains a Command FIS (CFIS), ATAPI Command (ACMD), and a Physical Region Descriptor Table (PRDT). The CFIS is the command to be sent to the HDD, while the ACMD is used for devices such as CD Drives, and is thus cleared to '0's in this thesis. The PRDT consists of up to 65,535 entries, each of which points to a data buffer in memory. For a read operation, data from the HDD is placed in these buffers. For a write operation, data contained in these buffers is written to the HDD.

Port x Command Issue (PxCI) and Port x Serial ATA Active (PxSACT) are two additional important registers inside the HBA. Each of these registers is a bit field in which each bit corresponds to a command slot. A set (*i.e.*, '1') bit in PxSACT signifies

that the corresponding command utilizes the Native Command Queuing (NCQ) protocol. NCQ a feature which allows the HDD to determine the optimal order to perform outstanding commands in order to minimize the required seek time. Two NCQ commands are of interest to this thesis, READ FPDMA QUEUED and WRITE FPDMA QUEUED. A set (*i.e.*, '1') bit in the PxCI register signifies to the HBA that the corresponding command table has been built by software and is ready to be issued to the disk. Once the command issued by the HBA has completed, an interrupt is triggered to alert the OS.

This thesis exploits the Command List structure and the PxCI and PxSACT registers corresponding to the target HDD. By taking control of these structures, the malicious hypervisor is capable of issuing arbitrary commands to the HDD without interacting with the OS. Additionally, the hypervisor maintains a shadow copy of these registers which contain the values expected by the OS. The hypervisor intercepts any attempts by the OS to read or modify the HBA's registers by using EPT virtualization technology and updates or responds with the value stored in the shadow copy. This ensures that the OS cannot detect the modifications to the command registers and to the command structures made by the hypervisor.

3.2.5 Extended Page Tables

An extension of Intel's VT-x technology, Extended Page Tables (EPTs), or more generically Second Level Address Translation (SLAT), improve the efficiency of memory address translation by providing hardware-optimized page table management for virtualization environments. EPT adds an additional layer of address translation by

converting the guest-physical addresses seen by the OS to physical addresses on the system.

In this thesis, EPT is utilized in order to create a wake-up condition when the OS tries to access the memory addresses mapped to the PxCI, PxSACT, and Port x Command List Base Address (PxCLB) registers. This allows the hypervisor to hide its modifications from the OS.

If EPT is enabled for the guest OS, physical addresses inside the guest are interpreted as guest-physical addresses [5]. Any attempt by the guest to access memory using a guest-physical address results in the use of a set of EPT paging structures to determine the actual physical address accessed. There are four levels of paging structures, and the number used determines the size of the page being controlled. Two levels provide control of 1-GByte pages, three levels provide control of 2-MByte pages, and four levels provide control for 4-KByte pages.

Each level allows for the specification of read, write, and execute permissions. In the event that a guest attempts to access a page to which it lacks appropriate permissions an EPT Violation occurs, resulting in a VM Exit which returns control to the hypervisor. This functionality provides a mechanism for the hypervisor in this thesis to awaken when the guest OS attempts to access any address guarded by the hypervisor. This is achieved by disabling the read, write, and execute access to the memory location of the PxCI, PxSACT, and PxCLB registers using the EPT.

3.3 Guarding Address Segments

By utilizing EPT, the hypervisor is able to intercept access by guest software in order to guard the PxCI, PxSACT, and PxCLB registers as well as any other desired

memory addresses. In order to do this, the hypervisor sets up entries in each of the four levels of the EPT paging structure.

As each entry in the PML4 table controls 512-GByte of memory, only the first entry in this table is used. This entry allows read, write, and execute permissions and points to an EPT Page Directory Pointer Table (PDPT). The remaining 511 entries in the PML4 table indicate that the entry is not present.

Each entry in the PDPT corresponds to 1-GByte of memory. The number of entries in this table depends on amount of memory mapped to the guest, and each additional entry is marked as not present. As this thesis assumes bit 7 of the Page Directory Pointer Table Entry (PDPTE) must be '0', each PDPTE points to a Page Directory (PD). All entries in the PDPT allow read, write, and execute permissions.

If the system contains more than 1-GByte of memory multiple PDs are required. Each Page Directory Entry (PDE) corresponds to a 2-MByte memory segment. The entries corresponding to the addresses being guarded each point to a Page Table (PT) while the remaining entries point directly to the corresponding 2-MByte page in physical memory. All entries in the Page Directory allow read, write, and execute permissions.

As with the previous level, there may be multiple Page Tables. Each Page Table Entry (PTE) corresponds to a 4-KByte memory segment. Each PTE points directly to a corresponding 4-KByte page in physical memory. The PTEs that correspond to guarded memory locations disallow read, write, and execute permissions in order to cause an EPT Violation on access. This EPT Violation results in a VM Exit and wakes the hypervisor. All other entries allow read, write, and execute permissions.

In addition to permission settings, the lowest-level EPT tables contain memory caching information that specifies the type of caching used for the memory location. These possible cache types are: Uncacheable, Write Combining, Write-through, Write-protected, and Writeback [5]. Typically, the memory cache type is calculated using a combination of the values in the Memory Type Range Registers (MTRRs) and the value specified in the Page Address Translation tables created by the OS. The MTRR values are initialized by the BIOS at boot time. When EPT is enabled the MTRR values are replaced with the values in the lowest-level EPT entries for the corresponding memory location. In order to ensure the memory caching types are set up correctly the hypervisor must assign the EPT cache types to reflect the MTRR values.

EPT violations occur on access anywhere inside the 4-KByte page with denied permissions. As the memory segments being guarded may be smaller than 4-KBytes, EPT Violations can occur on access to data not of interest to the hypervisor. Therefore, each time a VM Exit occurs due to an EPT violation the hypervisor must check to see if the address being accessed is of interest. If not, the hypervisor emulates the instruction requested by the OS. If the location is being guarded then the hypervisor takes appropriate action. A read attempt by the OS to read the contents of a guarded memory location causes the hypervisor to respond with the contents of the appropriate shadow. An attempt by the OS to write to a guarded memory address causes the hypervisor to update the appropriate shadow. In addition, an attempt by the OS to write to the HBA's PxCI register causes the hypervisor to perform additional steps in order to issue the command on behalf of the OS.

3.4 CPU Instruction Emulation

In order for the hypervisor to guard a memory location it denies access permissions within the EPT tables. The smallest granularity available using EPT is 4-KByte. An attempt by the guest to access any memory location within this 4-KByte segment causes an EPT Violation. Each EPT Violation results in a VM Exit and the guest's CPU instruction is prevented from executing. As a result, the hypervisor must decode the instruction and take appropriate action. If the instruction attempted to access a memory address that is not being guarded, the hypervisor emulates the instruction on behalf of the OS. If the address is being guarded the hypervisor takes appropriate action such as modifying the shadow copies.

The hypervisor first reads the instruction by dereferencing the guest's instruction pointer. The hypervisor then parses the instruction to determine what the guest OS is attempting to do. The instruction includes an optional REX Prefix byte, opcode (1-3 bytes), optional ModR/M byte, optional SIB byte, optional Displacement (1-4 bytes), and optional Immediate (1-4 bytes) [5].

Once the instruction is parsed the hypervisor determines the source and destination of the instruction, which can either be registers or memory locations. The base address of a memory location is determined by the addressing mode of the instruction. If a SIB byte is present in the instruction the base address is scaled using this byte. If a displacement is included in the instruction it is then added to the address.

Once the source and destination are calculated, the operand size, which specifies the number of bytes the source and destination reference, is determined. This is done using a combination of the REX Prefix (if present) and the opcode.

The hypervisor then determines if the address being accessed is guarded. This is done by comparing the physical address that resulted in the EPT Violation to a list of guarded locations. If the address is not guarded the hypervisor then emulates the instruction by performing the appropriate action (*i.e.*, Move or Bit Test). If the address is guarded the hypervisor performs the necessary actions, which depend on the memory location being accessed.

3.5 Issuing Commands from the Hypervisor

The technologies described above provide the necessary functionality to intercept, modify, and issue OS commands as well as issue arbitrary commands. This section demonstrates the steps necessary to perform these actions. Figure 3.6 below provides an illustration of the interaction between the OS, hypervisor, and HBA.

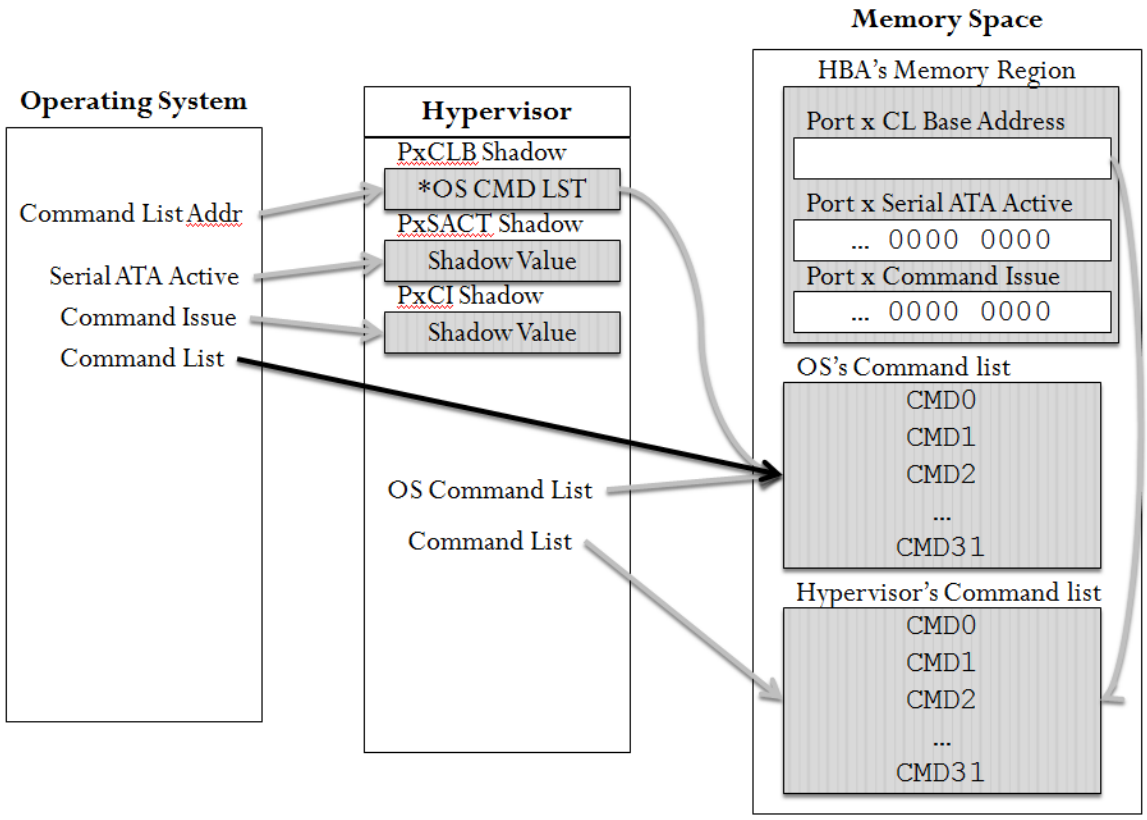

Figure 3.6 Operating System, Hypervisor, and HBA Interaction

3.5.2 PxCLB

The HBA stores a pointer to the Command List for each port (PxCLB), labeled "Port x CL Base Address" in Figure 3.6, which is typically created and maintained by the OS. The hypervisor stores the original contents of this register in a shadow copy, labeled "PxCLB Shadow." This value is used by the hypervisor to gain access to the OS's Command List. In order to issue commands, the hypervisor creates a Command List under its control, labeled "Hypervisor's Command List," and modifies the HBA's PxCLB register in order to instruct the HBA to utilize the hypervisor's Command List when issuing commands.

Access to this register by the OS is intercepted by the hypervisor. A write to this register by the OS results in the hypervisor updating its shadow value, and a read of this location results in the hypervisor returning the value stored in the shadow.

3.5.3 Port x Command List

A Command List is an array of 32 Command Headers located in memory, as shown in Figure 3.3 [6]. The hypervisor does not restrict access to the OS's Command List. Instead, it allows the OS to directly maintain its own list. The hypervisor creates a Command List under its control and updates it whenever necessary. The HBA's PxCLB register is modified to point to the hypervisor's Command List.

In order to issue a command on behalf of the OS, the hypervisor copies the Command Header from the desired slot in the OS's Command List into the next available slot in the hypervisor's Command List, following the expected round-robin ordering.

3.5.4 Command Headers

Each Command Header is 32 bytes. The Command Header structure is shown in Figure 3.4 [6]. When issuing commands on behalf of the OS no changes need to be made in this structure. The corresponding Command Header is copied from the OS's Command List to the hypervisor's Command List in the appropriate entry.

In order to issue an arbitrary command the Command Header must be set up by the hypervisor. The Command FIS Length is set to 5, the number of DWords in a Register-Host to Device FIS. The Write bit is either set or cleared, depending on the command being issued. The Physical Region Descriptor Table Length (PRDTL) is set to 1, indicating that the PRDT contains one entry. The Command Table Base Address is set

to the physical address of the hypervisor's Command Table. All other entries are cleared to '0'.

3.5.5 Command Tables

Each Command Table contains a Command FIS, ATAPI Command (not used in this thesis), and a variable-length Physical Region Descriptor Table. The structure is shown in Figure 3.5 [6] and each component is discussed below.

3.5.5.1 Command FIS

The Command FIS is located at offset 0 inside the Command Table [6]. It contains the command being issued, first sector being accessed, number of sectors being accessed, and a TAG indicating the entry in the Command Table where the command is located, among other values.

When issuing a command on behalf of the OS the hypervisor must modify the TAG value to correspond to the command slot in the hypervisor's Command List.

In order to issue an arbitrary command the Command FIS must be set up by the hypervisor. The FIS Type is set to 0x27 (Register Host-to-Device FIS), C is set to '1', Command is set to 0x60 (READ) or 0x61 (WRITE), Sector Count and LBA are set appropriately, and the TAG is set to indicate the correct entry in the Command List. All other values are cleared to '0' except Byte 7:Bit 7, which is defined as '1'.

3.5.5.2 Physical Region Descriptor Table

The Physical Region Descriptor Table (PRDT) is located at offset 0x80 inside the Command Table and contains between 0 and 65,535 entries, each containing an address of a data buffer and a byte count indicating the size of the buffer [6].

When issuing a command on behalf of the OS no actions must be taken with regard to this structure.

In order to issue an arbitrary command the PRDT must be set up by the hypervisor. In this implementation only one entry is used. The Data Base Address contains the physical address of a buffer controlled by the hypervisor. The Data Byte Count (DBC) represents the number of bytes being read/written by the command. The DBC is 0-based, meaning its value is the number of bytes minus 1.

3.5.6 PxSACT

The Port x Serial ATA Active (PxSACT) register is used to alert the HBA that a Native Command Queuing (NCQ) Command has been constructed [6]. This field is bit-significant and is written prior to a write to PxCI if the command is a NCQ Command.

An attempt by the OS to write to PxSACT results in an update of the PxSACT Shadow, and a read of this register returns the value stored in the shadow.

In order to issue an arbitrary command the hypervisor sets the bit in PxSACT that corresponds to the command entry containing the command to be issued.

The PxSACT register can only be cleared by the HBA hardware. The HBA clears the appropriate bit in PxSACT once the corresponding command has completed. After receiving an interrupt software reads this register to determine which of the outstanding commands have completed.

3.5.7 PxCI

The Port x Command Issue (PxCI) register is used to instruct the HBA that the corresponding command is ready to execute [6]. A write to this register causes the

hypervisor to perform the processing necessary to issue the command on behalf of the OS.

First, the hypervisor determines the next available location in its Command List, maintaining the expected round-robin ordering. It then copies the Command Header from the OS's Command List into the hypervisor's Command List at the determined location. Next, the hypervisor updates the TAG value in the corresponding Command Table to reflect the modified command slot. If the PxSACT shadow has the corresponding bit set, the hypervisor sets the appropriate bit in the HBA's PxSACT register. The hypervisor then sets the appropriate bit in the HBA's PxCI register to instruct the HBA to issue the command.

If the corresponding bit the PxSACT shadow was not set, this means that the command is not a NCQ command. The AHCI Specification [6] states that a Non-NCQ command cannot be issued if an NCQ command is outstanding. Since the hypervisor may have issued an NCQ command it must check to see if an NCQ command is outstanding. If not, it issues the Non-NCQ command by setting the corresponding bit in PxCI. If an NCQ command is outstanding, it sets an internal flag and does not issue the command immediately. Instead, it waits until an AHCI interrupt occurs that results in no outstanding NCQ commands (PxSACT == 0). At this time it issues the pending Non-NCQ command by setting the appropriate bit in the HBA's PxCI register.

In order to issue an arbitrary command the hypervisor sets the bit in PxCI that corresponds to the command entry containing the command to be issued.

The PxCI register can only be cleared by the HBA. The HBA clears the appropriate bit in PxCI when it issues the corresponding command to the HDD.

3.5.8 Event Injection

Once an issued command has completed, an interrupt is generated by the HBA to alert the OS. As the interrupt may correspond to a command issued or modified by the hypervisor, the hypervisor must determine the origin of the command. The hypervisor reads PxSACT to determine which commands are outstanding and compares these to an internal list of issued commands. Any commands not indicated to be outstanding by PxSACT have completed. If the command originated from the hypervisor then the hypervisor performs any action required in response to the interrupt including removing the command from its list of issued commands. Additionally, if the command was a READ command any processing of the retrieved data can be performed at this time. For example, the hypervisor can transmit a copy of the data over the network or copy the content of the file to another device.

If the command originated from the OS the hypervisor must update the OS's shadow structures appropriately by clearing the corresponding bits in the shadow PxSACT and PxCI copies. Additionally, the hypervisor must restore the TAG value inside the Command Table to the value originally written by the OS.

Once all fields are updated the hypervisor must forward the interrupt back to the OS. Since the OS may have interrupts disabled, the hypervisor first adds the interrupt to a queue and enables interrupt windowing in the VM Execution Controls. This results in the hypervisor being awoken when the OS is ready to accept interrupts. When this occurs any interrupts in the queue are injected by populating the event injection fields in the VMCS.

3.6 VMX Preemption Timer

The VMX Preemption Timer is a timer that can be utilized in order to wake the hypervisor after a predefined period of time. This is achieved by writing a value to the Preemption Timer entry in the VMCS. The timer value is decremented by the CPU at a rate proportional to the CPU clock [5]. By writing a value to this field and enabling the VMX Preemption Timer in the Pinbased Execution Controls field of the VMCS the hypervisor can trigger a VM Exit event to wake itself when the timer value reaches 0.

This functionality is utilized in order to wake the hypervisor at periodic intervals. When the hypervisor is awoken using this mechanism it resets the timer value and then checks the value of PxSACT to determine if any outstanding SATA commands are present. If no SATA commands are outstanding the hypervisor issues its own command. The VMX Preemption Timer is used in order to increase the consistency of issuing commands rather than relying on other events (*i.e.*, writes to HBA registers by the OS or interrupts generated by the HBA) to wake the hypervisor.

CHAPTER IV

EXPERIMENTS AND RESULTS

4.1 Introduction

The performance testing of this code developed as part of this thesis closely follows the test plan used by M. King in his thesis and consists of two phases [7]. The initial phase tests the functionality of the implementation, verifying that the hypervisor is able to compromise the confidentiality, integrity, and availability of the data contained on the hard disk drive (HDD). These tests consist of read operations to breach confidentiality, write operations aimed at the contents of specific files to breach data integrity, and larger write operations aimed to overwrite data or OS structures to breach data availability.

The second phase tests the effect the hypervisor has on the target system's performance and stability. This is achieved by performing timing analysis to measure HDD throughput and CPU performance before and after the installation of the hypervisor.

4.2 Functionality Experiments

This section describes the process used to verify the hypervisor functionality with regard to its ability to compromise the confidentiality, integrity, and availability of stored information.

4.2.1 Confidentiality

In order to demonstrate the ability to breach data confidentiality the hypervisor performs a read operation on the target HDD. A file containing a known pattern is created on the HDD and the physical addresses of the sectors containing the file are determined. The hypervisor is then instructed to read the first sector of this file. This is achieved by setting up the Command Table and Command Header, inserting the Command Header into the Command List, and setting the corresponding bit in PxSACT and PxCI registers. These operations are performed using the technique described in Section 3.5 such that the OS's disk access operations are not disturbed.

Upon completion of the command the Host Bus Adapter (HBA) issues an interrupt, which wakes up the hypervisor. Verifying that the hypervisor's buffer contains the known pattern demonstrates that the read operation succeeded, and thus data confidentiality has been breached.

4.2.2 Integrity

In order to demonstrate the ability to breach data integrity the hypervisor performs a specific write operation on the target HDD. A file containing a known pattern is created on the HDD and the physical addresses of the sectors containing the file are determined. The hypervisor is then instructed to write to the first sector of this file. This is achieved by setting up the Command Table and Command Header, inserting the Command Header into the Command List, populating the hypervisor's data buffer, and setting the corresponding bit in PxSACT and PxCI. These operations are performed using the technique described in Section 3.5 such that the OS's disk access operations are not disturbed.

Upon completion of the command the HBA issues an interrupt, which wakes the hypervisor. Verifying that the file's contents have changed demonstrates that the write operation succeeded, and thus data integrity has been breached.

Since the verification is being performed using the OS, it is important to note that if the file has been opened prior to the hypervisor's write operation the OS may have cached the file's contents. Therefore, the modifications may not appear until the cache is cleared or a system restart is performed.

4.2.3 Availability

In order to demonstrate the ability to breach data availability the hypervisor performs multiple write operations on the target HDD. A file containing a known pattern is created on the HDD and the physical addresses of the sectors containing the file are determined. The hypervisor is then instructed to write to each sector of this file. This is achieved by setting up the Command Table and Command Header, inserting the Command Header into the Command List, populating the hypervisor's data buffer, and setting the corresponding bit in PxSACT and PxCI. These operations are performed using the technique described in Section 3.5 such that the OS's disk access operations are not disturbed.

Upon completion of the command the HBA issues an interrupt, which wakes the hypervisor. The hypervisor then issues a command to write to the next sector of the file. Once all sectors have been written to, verifying that the file's contents have been overwritten demonstrates that the write operations succeeded, and thus data availability has been breached.

As with the integrity test, it is important to note that if the file has been opened prior to the hypervisor's write operations the OS may have cached the file's contents. Therefore, the modifications may not appear until the cache is cleared or a system restart is performed.

4.3 Timing Experiments

In order to ensure that the hypervisor does not significantly affect HDD or CPU performance, timing analysis is performed to measure the effect the hypervisor has on these operations. Disk performance is measured by timing the copy of a set of large files, and CPU performance is measured by using *IntelBurn*. IntelBurn uses the linpack library to utilize the floating point unit on the CPU. In this thesis the difference in execution times of the IntelBurn software when the hypervisor is not running and is running is used to measure the impact the hypervisor has on CPU performance[2].

4.3.1 Disk Performance

In order to test the impact on HDD read and write operations, large file copy operations are timed before and after the installation of the hypervisor. Five files with randomly-generated contents are created on the target HDD (named fa, fb, fc, fd, and fe), each 26GB in size to ensure that no entire file could be cached into the 24GB of system RAM. A batch file performs copy commands and records the time taken by each.

For each test case 70 file copy operations are performed. The test copies E:\fa to E:\dup\fa, then E:\fb to E:\dup\fb, etc. until all five files are copied. This process occurs 14 times (overwriting the previous duplicate files) resulting in 70 copy operations.

4.3.2 CPU Performance

In order to test the impact on CPU operations the burn test/benchmark tool IntelBurn is used to measure the performance of the floating point unit on the CPU. The test parameters were set to 35 iterations, 1024MB of memory utilization, and 64-bit operation. It is important to note that, in order to simplify implementation, the system was restricted to utilizing only one active processor core.

4.4 Results

This section provides the results from the experiments outlined above.

4.4.1 Functionality Experiments

The functionality experiments consist of verifying the hypervisor's ability to compromise the confidentiality, integrity, and availability of the data stored on the target HDD while maintaining system stability in order to avoid detection.

In order to compromise the confidentiality of the data the hypervisor issued a read operation to the target HDD and verified that the command completed successfully. A text file containing the string "HeLlO, ThIs Is A TeSt!" repeated 65 times, long enough to span two sectors and into the third, was created on the disk and its sector number was provided to the hypervisor. The hypervisor then issued a one-sector read command requesting the first sector of the file. Upon receiving an interrupt corresponding to the issued command the hypervisor verified that its buffer contained the file contents. This test demonstrates the hypervisor's ability to compromise data confidentiality.

In order to compromise the integrity of the data the hypervisor issued a write operation to the target HDD and verified that the command completed successfully. A

text file containing the string “HeLlO, ThIs Is A TeSt!” repeated 65 times was created on the disk and its sector number was provided to the hypervisor. In addition, the hypervisor populated its buffer with the character “A” followed by the character “a” repeated 511 times. The hypervisor then issued a one-sector write command specifying the first sector of the file. Upon receiving an interrupt corresponding to the issued command the file was opened using Notepad to verify that the first 512 characters had changed. This test demonstrates the hypervisor’s ability to compromise data integrity.

In order to compromise the availability of the data the hypervisor issued multiple write operations to the target HDD and verified that the commands completed successfully. A text file containing the string “HeLlO, ThIs Is A TeSt!” repeated 65 times was created on the disk and its sector number and size (number of sectors) was provided to the hypervisor. In addition, the hypervisor populated its buffer with the character “A” followed by the character “a” repeated 511 times. The hypervisor then sequentially issued three one-sector write commands, each specifying a different sector of the file. Upon receiving an interrupt corresponding to the last issued command the file was opened using Notepad to verify that all file contents had been overwritten. This test demonstrates the hypervisor’s ability to compromise data availability.

Each test completed successfully without affecting the stability of the target system, verifying the functionality of the hypervisor.

4.4.2 Timing Experiments

These timing experiments illustrate the performance impact of the malicious hypervisor on both HDD operations and CPU operations.

4.4.2.1 Disk Performance

Disk performance is measured by timing a series of large file copies before the hypervisor is installed as well as while the hypervisor is active given a variety of hypervisor configurations. Various tests are performed with the hypervisor's periodic wake timer operating at 1Hz with the hypervisor issuing read commands requesting 1, 4, 8, 4096, and 8192 512-byte sectors per operation. In addition, a test is run requesting 8192 sectors per read operation with the hypervisor's wake timer operating at 8Hz. Table 4.1 below lists the tests performed.

Table 4.1 Hypervisor Disk Performance Tests

Read Size (Sectors)	Preemption Frequency (Hz)
1	1
4	1
8	1
4096	1
8192	1
8192	8

The results of each test are then compared to the base case to determine the percentage impact of the hypervisor. In addition, the runtime of the longest running test case (*i.e.*, 8192 sector, 8Hz) is compared to the base case using a two-sample z-test in order to determine the probability that the mean time for the worst case hypervisor test is within the acceptable range. The results of these tests are given in Figure 4.1 below.

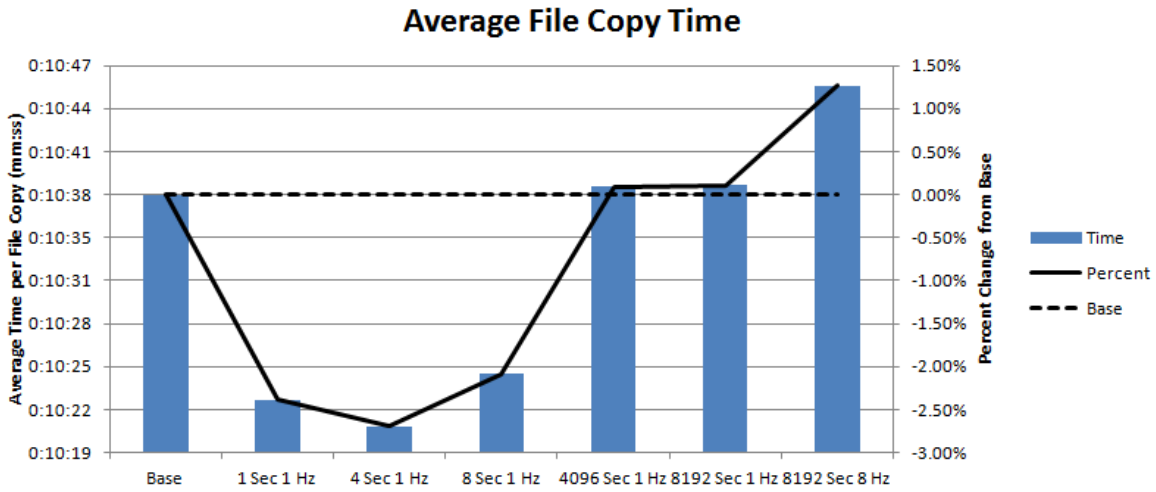

Figure 4.1 Disk Performance Test Results

The results show that the performance impact tends to increase with the number of sectors read as well as with the frequency of the interrupt timer. The results also indicate that the presence of the hypervisor improves the performance when the hypervisor issues small read requests. This may be a result of the hypervisor shadow copies, which provide a similar effect as caching in that requests by the OS to read guarded HBA registers no longer require relatively high overhead communication with the HBA over the PCI-e bus.

The test resulting in the most significant performance impact is the test issuing 8192-sector read operations with a wake frequency of 8Hz. A two-sample z test comparing this test to the base case shows that there is a 95% probability that the hypervisor caused less than 2.06% performance impact on the system. Additionally, there is a 99.5% probability that the hypervisor caused less than 2.25% performance impact on the system. Therefore, with a high degree of certainty, the hypervisor's impact on HDD performance is within the range stated in the hypothesis.

4.4.2.2 CPU Performance

The IntelBurn tool is used to measure the floating point performance of the CPU before the hypervisor is installed and for a series of tests while the hypervisor is active. Tests are performed for 1-sector and 8192-sector read operations with the VMX preemption timer ranging from 1Hz to 8Hz. Table 4.2 below lists the tests performed.

Table 4.2 CPU Performance Tests

Read Size (Sectors)	Preemption Frequency (Hz)
1	1
1	2
1	4
1	8
8192	1
8192	2
8192	4
8192	8

The results of each test are then compared to the base case to determine the percentage impact of the hypervisor. In addition, the runtime of the longest running test case (*i.e.*, 8192 sector, 8Hz) is compared to the base case using a two-sample z-test in order to determine the probability that the mean time for the worst case hypervisor test is within the acceptable range. The results of these tests are given in Figure 4.2 and Figure 4.3 below.

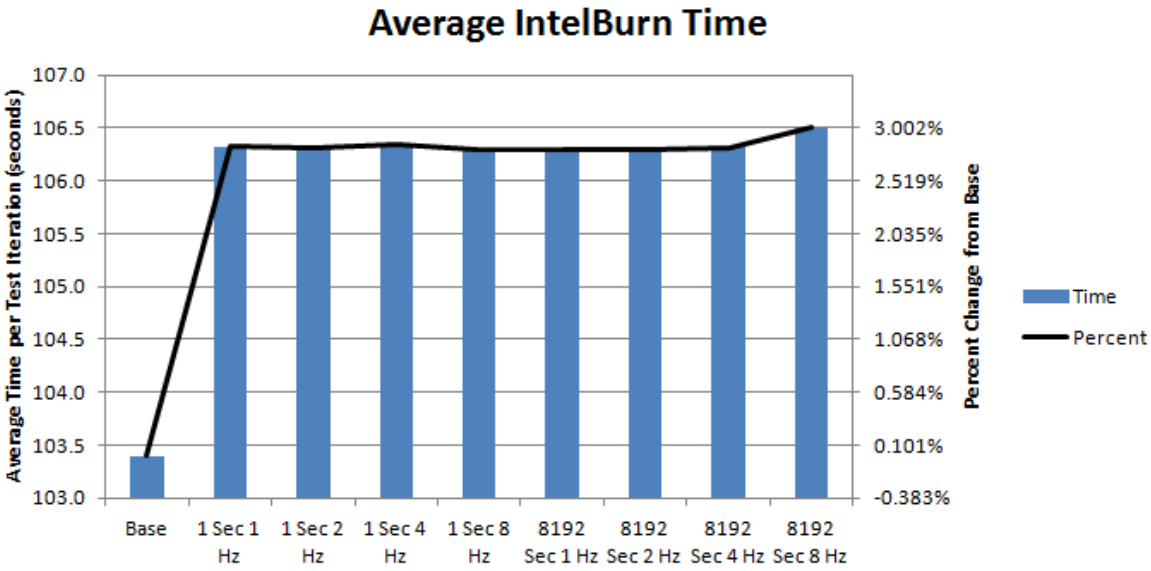

Figure 4.2 IntelBurn Results Including the Base Case

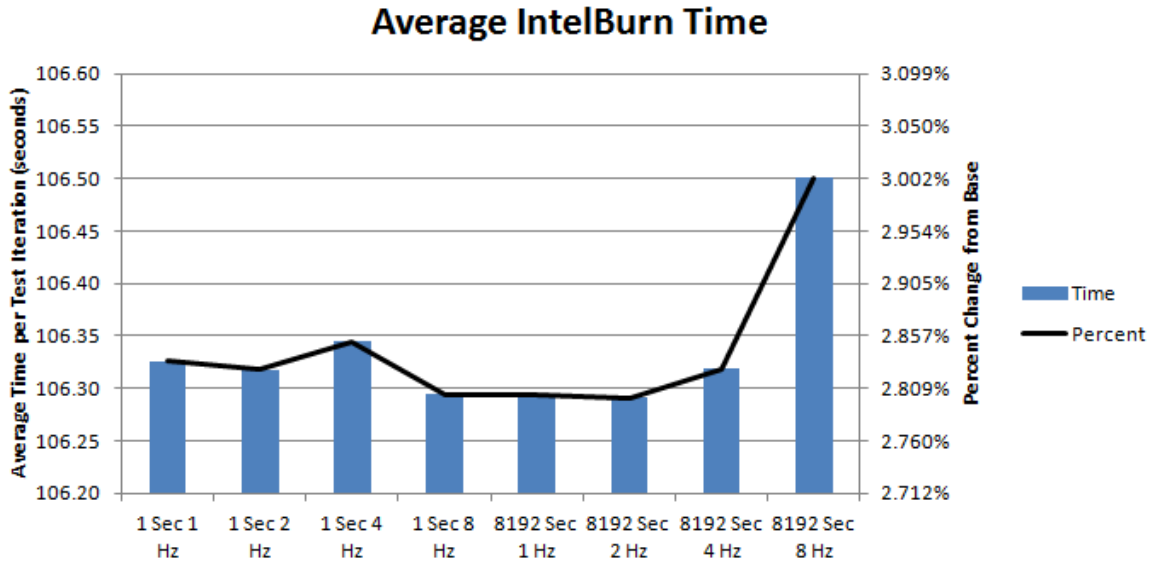

Figure 4.3 IntelBurn Results Without the Base Case

The results show that the hypervisor parameters are less significant than the presence of the hypervisor itself. The fastest hypervisor test (*i.e.*, 2Hz, 8192 Sectors)

resulted in a 2.80% increase in time taken, while the longest test (*i.e.*, 8Hz, 8192 Sectors) resulted in a 3.00% increase.

The test resulting in the most significant performance impact is the test issuing 8192-sector read operations with a wake frequency of 8Hz. A two-sample z test comparing this test to the base case shows that there is a 95% probability that the hypervisor caused less than 3.29% performance impact on the system. Additionally, there is a 99.6% probability that the hypervisor caused less than 3.4% performance impact on the system. Therefore, with a high degree of certainty, the hypervisor's impact on CPU performance is within the range stated in the hypothesis.

CHAPTER V

CONCLUSION AND FUTURE WORK

This thesis demonstrates the ability of a malicious hypervisor-based malware to compromise the confidentiality, integrity, and availability of data on a system while concealing its activities from the OS. This is achieved by using Intel's VT-x technology to convert the operating system into a guest under the hypervisor's control as well as Extended Page Tables to guard important memory segments from being directly accessed by the operating system.

The hypervisor implemented in this thesis performs the above functions while causing less than 5% overhead to disk and CPU operations and no disruption to the operating system's execution, thereby confirming the claims made in the hypothesis.

While this hypervisor conceals its disk operations from the operating system, it does not attempt to hide its own memory footprint. Memory for the hypervisor is allocated by the Windows device driver used to load the hypervisor, and is therefore detectible by software solutions. In order for this hypervisor perform on modern systems, support for multiple processing cores must be added. In addition, it is currently unknown how the hypervisor may react to Trim commands used in modern operating systems with solid-state disk drives installed. Further research is necessary in order to combine this work with a stealthy hypervisor and stealthy installation mechanism in order to prevent

detection and increase its utility by supporting multi-core systems and systems with solid-state disk drives installed.

REFERENCES

[1] Advanced Micro Devices, Inc., *AMD64 Architecture Programmer's Manual*, vol. 2, rev. 3.22, http://support.amd.com/us/Processor_TechDocs/24593_APM_v2.pdf (current Mar. 11, 2013).

[2] AgentGOD, "[RELEASE] IntelBurnTest v2.54," *Xtreme Gaming Studio*, http://www.xgamingstudio.com/forum/showthread.php?9-RELEASE-IntelBurnTest-v2-54 (current Mar. 11, 2013).

[3] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting Kernel-Level Rootkits Using Data Structure Invariants," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, IEEE CS Press, Los Alamitos, CA, Oct. 2011, pp. 670 – 684.

[4] Y. S. Dandass, S. T. Shannon, and D. A. Dampier, "Teaching Hypervisor Design, Implementation, and Control to Undergraduate Computer Science and Computer Engineering Students," *45th Hawaii International Conference on System Sciences*, IEEE Computer Society, Maui, Hawaii, 2012, pp. 5613 –5622.

[5] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer Manuals*, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (current Mar. 11, 2013).

[6] Intel Corporation, *Serial ATA AHCI: Specification, Rev 13*, http://www.intel.com/content/www/us/en/io/serial-ata/serial-ata-ahci-spec-rev1_3.html (current Mar. 11, 2013).

[7] M. A. King, *Attacking computer security using peripheral device drivers*, master's thesis, Department of Computer Science and Engineering, Mississippi State University, Starkville, Mississippi, 2010.

[8] Microsoft Corporation, *Kernel Patch Protection: Frequently Asked Questions*, http://msdn.microsoft.com/en-us/library/windows/hardware/gg487353.aspx (current Mar. 11, 2013).

[9] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization," *Intel Technology Journal*, issue 3, vol. 10, 2006, pp. 167-178, http://noggin.intel.com/content/intel-virtualization-technology-hardware-support-for-efficient-processor-virtualization (current Mar. 11, 2013).

[10] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," *Proceedings of the 13th conference on USENIX Security Symposium*, USENIX, San Diego, California, Aug. 2004, pp. 13–13.

[11] J. Rutkowska, "Beyond The CPU: Defeating Hardware Based RAM Acquisition (part I: AMD case)," *Black Hat DC*, Washington, DC, Feb. 2007, https://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf (current Mar. 11, 2013).

[12] J. Rutkowska, "Subverting Vista Kernel for Fun and Profit," *Black Hat USA*, Las Vegas, Nevada, 2006, http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf (current Mar. 11, 2013).

[13] Serial ATA International Organization, *Serial ATA Specification*, rev. 3.0 Gold, Jun. 2009.

[14] L. Wang and P. Dasgupta, "Coprocessor-based hierarchical trust management for software integrity and digital identity protection," *Journal of Computer Security*, vol. 16, no. 3, Aug. 2008, pp. 311–339.