ON THE DETECTION OF VIRTUAL MACHINE INTROSPECTION FROM INSIDE A GUEST VIRTUAL MACHINE

By

Brandon Ashlee Marken

RECOMMENDED:

[illegible]
Dr. Jon Genetti

[illegible]
Dr. Christopher Hartman

[illegible]
Dr. Ronald Barry

[illegible]
Dr. Channon Price

[illegible]
Dr. Orion Lawlor
Advisory Committee Chair

[illegible]
Dr. Jon Genetti Chair,
Department of Computer Science

APPROVED:

[illegible]
Dr. Douglas Goering,
Dean, College of Engineering and Mines

[illegible]
Dr. John Eichelberger
Dean of the Graduate School

7 December 2015
Date

ON THE DETECTION OF VIRTUAL MACHINE INTROSPECTION FROM INSIDE A GUEST VIRTUAL MACHINE

A

DISSERTATION

Presented to the Faculty

of the University of Alaska Fairbanks

in Partial Fulfillment of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

By

Brandon Ashlee Marken, B.S., M.S.

Fairbanks, Alaska

December 2015

Abstract

With the increased prevalence of virtualization in the modern computing environment, the security of that technology becomes of paramount importance. Virtual Machine Introspection (VMI) is one of the technologies that has emerged to provide security for virtual environments by examining and then interpreting the state of an active Virtual Machine (VM). VMI has seen use in systems administration, digital forensics, intrusion detection, and honeypots. As with any technology, VMI has both productive uses as well as harmful uses. The research presented in this dissertation aims to enable a guest VM to determine if it is under examination by an external VMI agent. To determine if a VM is under examination a series of statistical analyses are performed on timing data generated by the guest itself.

Table of Contents

List of Figures

List of Tables

Page

Acknowledgements

I would like to begin by thanking my committee for their contribution to this work. In particular I would like to give special thanks my original advisor Brian Hay for his guidance in the first years of my doctoral program. A special thanks as well to my advisor Orion Lawlor both for his guidance during my program and for taking me on as a graduate student with short notice. Thanks to Jon Genetti for helping me navigate some of the more difficult issues in the university bureaucracy.

In addition to them two teachers from high school played an important part in me choosing a career as a scientist. Paul Schwartz and Wade Roach from Dimond High School in Anchorage got me to focus on science as a career goal rather than simply choosing some path of least resistance. By giving me challenges as opposed to worksheets I was able to discover my love of the scientific process.

In my time as a graduate student at University of Alaska Fairbanks I had the pleasure of consulting with other graduate students both past and present. Using them as a sounding board helped me overcome problems that would have otherwise taken me far longer. In particular special thanks are due to Kevin Galloway, Mike Moss, Ann Tupek, and Christopher Granade for their assistance.

Finally I would like to thank my family and friends not already mentioned. In particular my wonderful wife Jennifer for her support in this endeavor. Without her support (and reminders to keep working on my dissertation rather than playing Fallout) this likely would not have come to fruition. I would also like to extend my gratitude to my in-laws for their support during these years. Finally, I must remember to thank my mother, who I inadvertently left out of my M.S. write up. This thanks is for her love and support as my mother and not simply because she has been threatening me with physical violence since 2010 lest I forget her in my doctoral dissertation.

Chapter 1

Introduction

Computers have become ubiquitous in our society over the past several decades. One of the negative consequences of this change is that these systems are vulnerable to a great number of threats. One method introduced in the past decade to combat such threats is Virtual Machine Introspection (VMI) which aims to analyze the state of a virtualized Operating System (OS) called a guest. While research into VMI has been extensive [1, 2, 3, 4, 5, 6, 7, 8] the security implications of VMI remain largely unexplored. This dissertation will concern itself with the detectability and reliability of VMI. This chapter will focus on the technical information and current literature necessary to put the rest of the dissertation into context.

1.1 Computer Security

The three pillars of computer security are integrity, availability, and confidentiality [9]. The integrity of a computer system is the property that it will behave as intended by both the user and the designer or programmer. The availability of a system is the property that a system can be used by a user at any time required. The confidentiality of a system is the property that it can only be accessed by authorized users. We consider the security of a system to be breached if any one of these is violated.

For an example let's consider Gmail [10]. A user expects that an email sent to their boss will go to their boss. The user also expects the email client will deliver the message exactly as written. These are examples of integrity. A user with an Internet connection can access Gmail at any time day or night. This is an example of availability. A login mechanism is used to enforce confidentiality. The user expects that no one else will be able to access their emails. This is an example of confidentiality of a system.

A VMI agent can breach the security of a computer's system by either attacking the integrity or the confidentiality of that system. The breach of confidentiality can occur when a VMI agent reads the pages of a target system and a breach in the integrity can occur if the VMI agent alters the pages of a target system.

1.2 Technical Background

In order to begin to understand the nature of VMI and, by extension, Virtualization, we must discuss some of the details of the operation of the 64-bit x86 processor (x86-64), some of the relevant design and functions of the Xen and KVM hypervisors, as well as the basics of VMI and the VMI tool suites we will be using for the remainder of this dissertation.

1.2.1 Privileges

In earlier versions of the x86 line (pre-80286) the processors existed in what we now call real mode. In real mode processes have unlimited access to physical memory as well as access to all peripherals. This means that processes can easily access the memory of other processes either accidentally or intentionally. This can cause a great deal of instability as well as security vulnerabilities.

To address this situation Intel introduced protected mode with the 80286 [11]. Protected mode is enabled by setting the PE flag in the CR0 register on the CPU and enables memory protection features such as paging and virtual memory. Protected mode is disabled at boot in order to ensure backwards compatibility and the PE bit must be set by the OS in order to enter protected mode. Once protected mode is enabled it cannot be disabled until the system is rebooted.

In the 32-bit x86 line of processors, introduced after the 80286, protected mode enables 4 separate privilege levels called rings. Ring 0 has the most privilege, Ring 1 has fewer privileges, Ring 2 fewer still, and Ring 3 the fewest privileges. While all privilege levels were intended to be used, only rings 0 and 3 were used in commodity operating systems such as Windows [12] and Linux [13]. When the x86-64 processors were released the privilege stack was reduced to two privilege levels (fig 1.1). In 2006 a third ring called Host Mode by Intel and Root Mode by AMD (which we will call Host/Root Mode for the duration of this dissertation and is colloquially referred to as Ring -1) was added to the x86-64 line of processors [14].

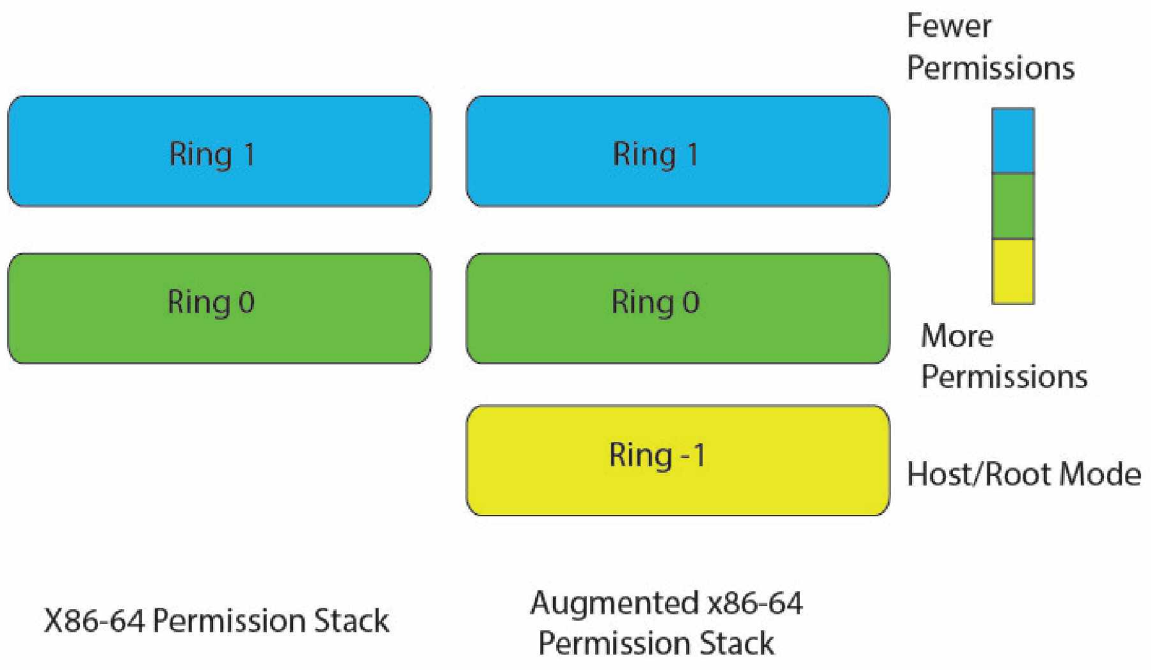

Figure 1.1: Augmented x86-64 privilege Stack

1.2.2 Virtualization

Virtualization itself is not a new concept. It began in the 1960s with the IBM System 360 [15]. This system, like most others for the next nearly four decades, used the trap and emulate model. In this model a virtual machine (VM) will proceed unaltered until it reaches an instruction which it cannot execute due to an insufficient privilege level [16]. The guest operating system then faults to the hypervisor which performs some set of instructions. This set of instructions will then perform an operation with an identical effect to the original instruction.

In 1974 Popek and Goldberg [16] formalized the conditions, which were sufficient to allow a CPU architecture to support virtualization. To begin they define a Virtual Machine Monitor (VMM) as a piece of software which provides a programming environment which is "essentially identical" [16] to the machine being virtualized (fidelity), only causes a minor performance decrease (efficiency), and is in complete control of the resources (resource control or safety).

Popek and Goldberg then separate CPU instructions into three different classifications. Privileged instructions are those which will cause a fault if run in user mode (such as the Intel *CLI* [17] instruction), control sensitive instructions which change the configuration of resources in a system (the Intel *CLI* instruction also falls into this category), and behavior sensitive instructions whose effects vary based on the configuration of resources (such as the *POPD* [17] instruction which varies based on privilege level). Control sensitive and behavior sensitive instructions are collectively referred to as sensitive instructions.

With these requirements and definitions acting somewhat as axioms, Popek and Goldberg give us two theorems [16]:

Theorem 1.2.1 (Popek Theorem 1) *"For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [16]*

Theorem 1.2.2 (Popek Theorem 2) *"A conventional third generation computer is recursively virtualizable if it is virtualizeable and a VMM without any timing dependencies can be constructed for it." [16]*

The proofs for these theorems can be found in the original work by Popek and Goldberg. The first theorem says that a VMM can only be constructed for an architecture if the sensitive instructions are a proper subset of the privileged instructions and the second says an architecture is not virtualizable if a VMM without timing dependencies cannot be constructed for it.

This model is not appropriate for x86 virtualization however as many of the x86 assembly instructions, such as *POPD* [17] (which pops an element off the floating point stack), are sensitive but not privileged. This violates the conditions required for VMM construction of Popek's first theorem and by extension x86 cannot be virtualized via the trap and emulate method.

In 1999 VMWare patented their techniques for binary translation, which they introduced in 1998 [18], allowing the x86 architecture to be virtualized. In binary translation the hypervisor runs one ring below the guest OS (in Host/Root mode on x86-64). The translator reads guest memory starting at the instruction pointer (eip/rip) and caches up to 12 instructions (fewer if a terminating instruction is reached) in a Translator Unit (TU). Unprivileged and non-sensitive instructions (such as *MOV* or *XOR*) are translated IDENT (identically) with no changes made.

Privileged and sensitive instructions however are translated producing Compile Code Fragments (CCF) using non-privileged instructions. Agesen et al. [19] use the *CLI* instruction as an example. The *CLI* instruction clears the interrupt flag on the physical CPU. Since the guest VM cannot (and should not) clear the interrupt flag on the physical CPU the interrupt flag is cleared on the VCPU (*vcpu.flag*) using the *AND* instruction. Once a TU is translated into a CCF it is then run on the CPU.

This began the boom in x86 virtualization; in particular, the debut of Xen in 2003 [20], which introduced the paravirtualization method for x86 virtualization. In paravirtualization, like binary translation, the hypervisor runs at Ring 0, the guest OS runs at Ring 1, and user code runs at Ring 3. Paravirtualization works by using a modified kernel, replacing instructions which will require hypervisor support such as those involving memory management with hypercalls [20]. These hypercalls result in the hypervisor performing some operations which result in the state being presented to the VM which from its perspective appears as if it has been performed on physical hardware rather than on a VM. This allows the guest to run, without any modification to user space code, much like trap and emulate and binary translation. This method, however, traditionally requires that a specific kernel be used, which greatly increases the time between versions and makes the virtual environment sensitive to OS changes. As of Linux version 3.0 however the introduction of Paravirt Ops into the Linux kernel has added native paravirtualization support removing this limitation [21]. Due to the nature of paravirtualization we are limited in a practical but not theoretical sense to open source operating systems such as Linux [13] and BSD [22].

The 64-bit line of x86 processors was introduced by AMD in 2003 and this line only had 2 levels of privilege unlike the 4 in the 32 bit lines. As a result there was no longer room to run a hypervisor, guest OS, and user code each on their own privilege level. In 2006, Intel (VT-x [23]) and AMD (AMD-V [14]) both added hardware virtualization to their x86 line of processors. Hardware virtualization adds another layer in the privilege stack below Ring 0 called the Host Mode and Root Mode for Intel and AMD respectively. These are colloquially, though not formally, referred to as Ring -1. A structure called a Virtual Machine Control Block (VMCB) is defined in hardware. This structure holds the list of all instructions to be intercepted. Certain instructions are required to fault but others can be added to the VMCB by the hypervisor. This method allows the guest OS to run on Ring

0 of the hardware as it would normally expect to. The guest runs normally until it has to run an instruction which requires a fault (as defined in the VMCB). The instruction which caused the fault is then trapped and handled by the hypervisor. Any operating system can be virtualized in this manner, however it does require special hardware instructions (though those are now available on almost all Intel and AMD CPUs) and the shifts to the Host/Root mode are time consuming and need to be reduced to the smallest number possible to keep the process efficient.

Shortly after the advent of X86 hardware virtualization [23] [14] the hypervisor KVM (for Kernel Virtual Machine) was introduced by Kivity et al. [24] and was included as part of the main line Linux [13] kernel the same year. As part of the Linux kernel KVM is able to reduce some of the code base by incorporating certain aspects of the Linux kernel such as the scheduler in order to handle managing resources of virtual machines. Throughout this experiment we will be using either Xen or KVM as appropriate. Due to the differing architectures of the two hypervisors there will be some approaches that will work with one and not the other. Where applicable we will comment on why one was chosen for a specific experiment over the other.

1.2.3 X86-64 Memory Architecture

Modern commodity CPUs use an architecture known as the Von Neumann Architecture [25]. At the most basic level a computer based on the Von Neumann

Architecture consists of a CPU which processes data and instructions, memory, mass storage, and input/output devices. Due to limitations on the throughput between the different parts of a computer we encounter what's known as the Von Neumann Bottleneck [25]. Data on a CPU register is fast to access, RAM is slower, disk drives slower still, and network based storage the slowest available.

To address this problem CPU cache was introduced. Cache is a small amount of extremely fast RAM which exists on the CPU in order to alleviate, but not eliminate, the Von Neumann Bottleneck. In most modern X86-64 CPUs there are three levels of cache: L1, L2, and L3. L1 is the smallest and fastest with L2 being slightly larger but slower and L3 being even larger and slower. On the some of the newest cores such as the Haswell core a fourth level of cache is added [26]. This cache is shared between the CPU and the integrated Intel GPU.

On a modern CPU the process of accessing memory begins by checking the Translation Lookaside Buffer (TLB) (fig 1.2). The TLB is a small amount of cache which holds virtual address translations in the form of Page Table Entries (PTEs). These entries (discussed further in section 1.2.4) map virtual memory to physical memory. The Sandy Bridge and Haswell Cores have two levels of TLB. The L2 TLB maps both data and instructions whereas the L1 TLB is split into an instruction and a data TLB [27]. If the entry is in the TLB we have a TLB hit. If this occurs we look to see if our physical address is in the L1 cache. If this is the case we have an L1 cache hit and the data is loaded into the CPU. If this is not the case we have an L1 cache miss. If this occurs we look to see if our physical address is in the L2 cache. Again if it is we have an L2 cache hit and load our data to the CPU. If an L2 cache miss occurs we have to check the L3 cache. Unlike the L1 and L2 caches the L3 cache is shared by all cores on a CPU. If our physical address is in the L3 cache we have an L3 cache hit and our data is given to the CPU. Otherwise it requires looking to see if the data is in DRAM. In the event of a TLB miss a walk of the page table is necessary (1.2.4).

Each of the steps described before contributes to our Average Memory Access Time (AMAT). The formula for computing AMAT is described in equation 1.1, where H_i is the time per hit on the some memory element (e.g. cache or the TLB), AMP_i is the average penalty per miss on that element, and M_i is the probability that a miss will occur on that element. This is summed over all the elements of memory.

$$AMAT = \sum_{i=0}^{n} H_i + AMP_i \times M_i \tag{1.1}$$

1.2.4 Virtual Memory and Paging

In a modern programming environment memory is abstracted so that each process sees its address space as one contiguous region. This is a convenient abstraction made possible through virtual memory. In modern x86 and x86-64 virtual memory is provided through a system called paging. In paging memory is broken into

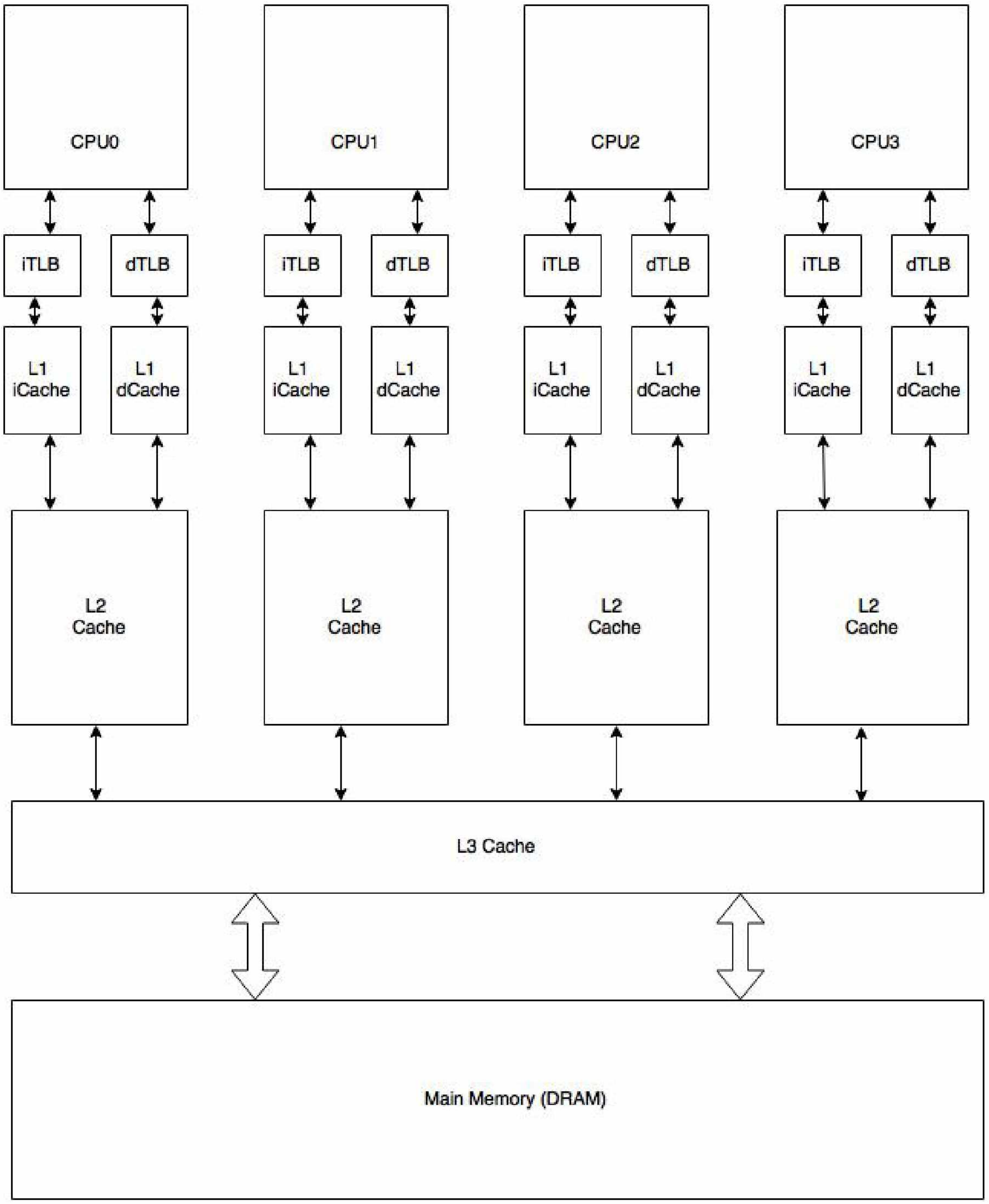

Figure 1.2: X86-64 Memory Hierarchy

segments called pages which are typically 4kb (though larger pages are supported up to 2MB and 1GB for 32-bit and 64-bit x86 respectively). A data structure called the page table is used to map virtual memory that the process sees to physical memory. In the x86 architecture this mapping is handled by hardware known as the Memory Management Unit (MMU).

The x86-64 processor uses a 4 level paging system to translate virtual addresses to physical addresses when 4kb pages are used and a 48-bit address space are used. The levels are organized in a tree structure. The CR3 holds location of the page directory for the process. The first 16 bits are unused and next 36 bits are broken into four segments of 9-bits each. The first is called the Page Map Level 4 page (PML4), the next is called the Page Pointers Directory page (PDP), the next is called the Page Directory page (PD), and the final segment is called the Page Table page (PT). The remaining 12 bits are for the page offset, which tells us where in the page the memory is located fig 1.3 [28].

1.3 Xen

Xen is a type-1 or "bare metal" hypervisor. This means that Xen runs directly on the physical hardware and is not itself hosted in another environment. This is in contrast to a type-2 hypervisor like VMWare Player which is hosted inside a Linux or Windows environment. Administration of Xen and its VMs is done via a special paravirtualized guest called the Dom0. All other guests in Xen can be either paravirtualized or hardware virtualized. For the remainder of this dissertation we

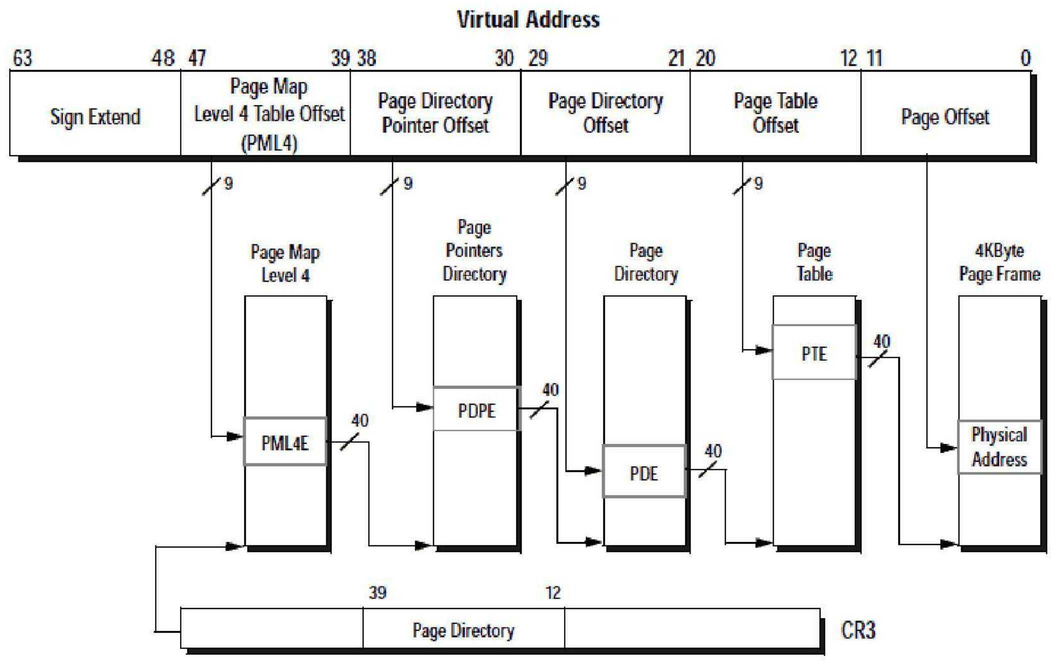

Figure 1.3: Page Table in x86-64 [29]

will assume all DomU guests (those guests which are not Dom0) are hardware virtualized unless otherwise specified.

1.3.1 Xen Virtual Memory Management

As Xen supports two different types of virtualization it also supports three different kinds of virtual memory management. Traditionally software virtualization uses a shadow paging scheme [20] which keeps an additional "shadow" page table. This provides an additional layer of abstraction between the guests and the physical hardware.

Since paravirtualized guests use hypercalls for sensitive instructions Xen is not required to keep a full shadow page table for its memory management, instead using a technique known as direct paging. In direct paging guests invoke a hypercall which directly maps their page table entries from virtual memory to physical memory as opposed to the extra paging layer provided in shadow page tables [20], essentially moving control of memory management from the OS to the hypervisor.

Since hardware virtualized (HVM) guests are not modified in the way that paravirtualized guests are, they do not have the option of this direct paging technique and instead have the option of either hardware assisted paging (HAP) or shadow paging. HAP uses a technique called Second Level Address Translation (SLAT; see fig 1.4) which is included in Intel processors since the Nehelem line and in AMD processors since the Barcelona line. Their technologies are called Extended Page

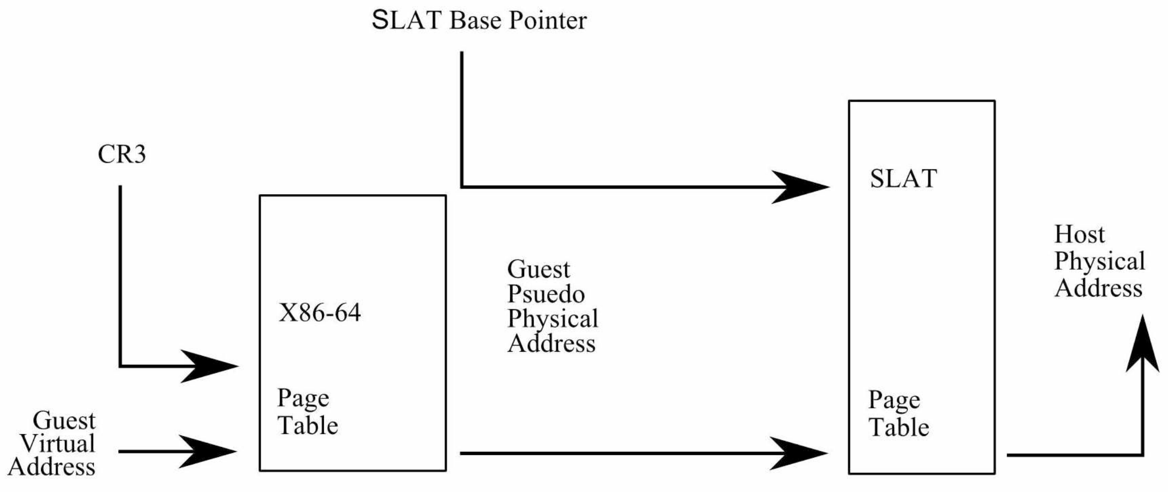

Figure 1.4: Diagram of Hardware Accelerated Paging using SLAT

Tables (EPT) and Rapid Virtualization Indexing (RVI) respectively. In SLAT the guest OS still maintains a logical page to physical page mapping (fig 1.4). However these physical pages are in pseudo pages and do not correspond directly to physical memory. Instead the hypervisor maintains the pseudo physical to machine page (or actual physical page) mapping.

HVM guests also have the option of using a shadow page table (SPT) similar to that used in VMWare [18]. Like HAP shadow paging works by adding another layer of abstraction to paging. In this model the processes use the software page tables provided by the OS just like they would normally. When a guest OS tries to update a page a shadow page is allocated. This shadow page can be then altered with no constraints. When the page is ready to be moved into the regular page table (i.e. permanent changes have been accepted) references are updated such that they point to the new page rather than the original.

1.4 KVM

Like Xen, KVM is a type-1 (i.e. Bare Metal) hypervisor. Unlike Xen, which is an independent hypervisor loading up a paravirtualized Linux VM as the administrative unit, KVM is itself integrated into the Linux kernel, which allows the use of a non-virtualized Linux environment for administration (as opposed to the paravirtualized Linux environment used in Xen). As a result of being part of the Linux kernel, KVM is able to use portions of the Linux kernel, for instance KVM uses the Completely Fair Scheduler [30] used in the kernel to schedule CPU time for VMs. Other functions such as memory management are also done using the internal Linux mechanisms.

1.4.1 KVM Page Merging

In version 2.6.2 of the Linux Kernel a scheme called Kernel Samepage Merging [31] was introduced. In this scheme pages which are identical among processes are merged in order to save memory. To accomplish this processes which may be candidates for merging register themselves with the kernel. The kernel then scans the registered areas of virtual memory by taking the hash of those pages. Pages which have unique hashes are skipped for the remainder of the scan, while pages which have the same hashes are then checked byte by byte to ensure they are identical and avoid hash collision. Identical pages are then merged by first marking all page table entries where page occurs as unwriteable. Next all page table entries which refer to that page are updated so they point to only one instance of the page. The remaining unmerged copies of the merged page are then freed and a final direct memory comparison is made to ensure that the pages have not changed. These merged pages can only remain merged so long as the processes or VMs using them are only reading from them. To address this KVM uses a Copy-On-Write (COW) scheme. With this technique pages remain merged until one guests attempts to write to it. When that occurs a copy of the page is made for that guest and the remaining guests continue to use the merged page.

1.5 Virtual Machine Introspection

Garfinkle and Rosenblum [7] introduced Virtual Machine Introspection (VMI) in 2003. In VMI the state of a running VM is interpreted by some external entity, usually either another VM or by the host system. To accomplish this memory is mapped or copied from the target VM to the VMI program. Memory is then interpreted to determine some portion of the internal state of the target VM. While the original work was used for intrusion detection a number of other applications have emerged in the years following.

Memory outside of a given context has no intrinsic meaning, only inside a given context or process does it have meaning. As such memory in a guest OS has no meaning outside of that guest. This problem is known as the semantic gap. The typical way to solve the problem [7], [8] is to use a template approach for each individual OS. In this approach certain areas of memory are marked as being where certain structures in kernel memory are located. That way a VMI agent can look for the relevant areas in memory and interpret them at run time. The process of determining the location of these kernel structures can be time consuming, error prone, and must be repeated for each version of a running kernel. Work is being done to address the problem of the semantic gap and is addressed later in this dissertation.

At its most fundamental level VMI represents a mapping of memory from the guest to an area outside of the guest (be it another VM as in the case of Xen or an administrative Linux OS as in the case of KVM). In this case we can study one method for VMI without losing generality. For our experiments we will use VMI-Tools [32]. It is an open source framework to allow easy development of VMI applications, supports Linux and KVM, and has appeared in a number of works on VMI (though in some of these works it appeared under its former title XenAccess) [3, 33, 34, 35, 36].

1.6 Virtual Machine Introspection Literature

Gu et al. [5] introduced a scheme for active VMI. In this scheme for VMI a process is injected from the hypervisor to the guest VM. This process is hidden inside an innocuous process already running inside the VM. These processes are called the implanted process and the victim process respectively and are both chosen by the host administrator at runtime. When the victim process is scheduled to be run, the hypervisor captures the context switch. The implanted process the replaces the victim process. To accomplish this the relevant instruction pointers are changed as well as the relevant stack pointers etc. When the OS resumes to continue the context switch the implanted process is run in place of the victim process. Once the implanted process has completed or the hypervisor determines it is time to switch the context back the contexts are switched and the victim process runs are executed normally.

In the initial experiments ltrace, a program to scan the libraries called by a process, is implemented as the implanted process. This allows them to implant ltrace into a running VM and are able to trace the library calls of selected processes. While this does accomplish some of the goals of conventional VMI it can also be unusable for applications such as digital forensics in which a VM must remain unaltered in any capacity in order to be of use in a courtroom setting.

Dolann-Gavitt et al. introduce a system called Virtuoso [4] which is aimed at bridging the semantic gap. The execution of Virtuoso is broken into three phases: The training phase, the analysis phase, and the runtime phase.

In the training phase Virtuoso attempts to gain information about the guest. Inside the guest a program must be written to access the information desired. For instance if one were trying to write a VMI program to access the Process Identifiers (PIDs) then one would write an in guest program which would access the PIDs. This program would then be run repeatedly and each time it was run a syscall trace (such as with Linux strace) would be taken of that program. After running this program repeatedly an extensive list of execution paths would be generated. Using this combination of execution paths the analysis phase can begin.

The collection of syscall traces, while it does contain the necessary execution paths, also contains a great deal of extraneous noise. This is because the system trace follows the entire execution through the system not just the relevant information. Parts of the trace, which are known a priori to be unnecessary such as hardware interrupts or memory management, are thrown out immediately. Then a dynamic data slice [37] is then done. The slices are merged into a unified program which can be turned from an in guest program to an out-of-guest program.

The translated code cannot be run directly on the host. So Virtuoso creates a runtime environment for the translated code. The run time environment is installed on the host machine and has the ability to run the translated code. This gives the code created by Virtuoso an appropriate context from which to access the VM resources (e.g. CPU registers or main memory) in a read-only manner.

While Virtuoso does make significant progress in bridging the semantic gap it does not change the fundamental nature of introspection. The tools generated by Virtuoso still simply read and interpret memory from the guest, which means that our study will still apply to Virtuoso without having to directly address Virtuoso.

Like Virtuoso, Fu and Lin [6] make an attempt at bridging the semantic gap and automatically generating VMI utilities. The process begins with an untrusted target VM (called the product-VM) and a secure trusted VM (called the secure-

VM). Three techniques are then used to extract information from the product VM. These are syscall execution context identification, redirectable data identification, and kernel data re-direction.

The context being executed is identified using a stack to keep track of times *IRET* and int are called. Global kernel data is then identified using an adapted form of taint analysis. With the relevant information located and the contexts identified they are then able to redirect the kernel information between the product-VM and the secure-VM. This allows native system monitoring utilities to be run on the secure-VM as VMI targeting the product-VM unaltered.

When certain native Linux utilities such as lsmod [38] were run on product-VM and the secure-VM, identical results were produced. However certain utilities like date [39] and ps [40] produce similar results on the product-VM and secure-VM, however these results were not identical. This was found to be due to the timing of the snapshots they were using for the analysis as certain programs are quite time sensitive. While this has approached an automatic bridge for the semantic gap this work is still reliant on semantic information about system calling conventions and can be altered if they are changed between kernels. This seems like a promising tool to use for VMI however as of this publication the source code has not yet been released for use by the public, so it will remain unstudied in this work.

1.6.1 Uses for Virtual Machine Introspection

In this section we will discuss current uses for VMI especially as they related to information assurance and security.

In Crawford and Peterson [41] VMI is leveraged to address the insider threat problem. The insider threat problem is the situation that occurs when current or past members of an organization have both malicious intent and legitimate access to a system or systems [42]. To accomplish their goal they break their approach into four steps: Development of a taxonomy of malicious insider behavior, development of a taxonomy of VMI observables, malicious insider detection, and data validation.

To develop a taxonomy they began by setting up a number of possible high-level uses cases. The activities identified in this taxonomy are printing activity, disabling defensive tools (e.g. anti-virus), abusing removable media (e.g. putting sensitive information onto a flash drive), sudden change in employee behavior, use of remote access, and strange clipboard activity. Once they determine major uses cases they wish to look for they decompose each scenario into individual observable attacks and each attack is broken down into seven areas of analysis. These areas are the attacker (who is doing something or can do something), which tools are used, which vulnerabilities are used, what actions are taken in order to achieve the desired effect, which systems are targeted, what the result of this unauthorized attack is, and what the objective of that result is. Using this taxonomy they can break many insider threat problems into simple terms.

The next step is to determine which parts of a system can be observed by VMI. The observables consist of "registry information, hexadecimal patterns, and clipboard information." Each of the potential malicious activities is performed while the observables are being monitored by VMI. If those observable areas create signature patterns then they can be used to identify the insider threat actions from VMI. The relevant observables are provided in the work from Crawford and Peterson [41].

The third step is essentially the experiment portion of the paper. During the experiment VMI is used as well as Windows event logs. VMI and Windows event logs are analyzed while certain potentially malicious operations described earlier were performed. The experiment recorded which actions set showed as being malicious and which ones did not.

The final step is the analysis phase. For each attack in the scope of the research they perform it manually several times to determine if their tools developed from the third phase are capable of detecting it. Their results showed that they were able to detect 18 different types of insider attacks. They report a high false positive rate though they don't give the specific rate of either detection or false positives. The authors indicate that while this work shows potential more work needs to be done on determining which observables correlate to which observables can indicate attacks in order to increase the accuracy of their detection.

Harrison et al. [43] have proposed using VMI combined with the related yet distinct field of Forensic Memory Analysis (FMA). Harrison's goal is to put together an entirely out-of-band passive sensor to monitor for malicious software such as kernel rootkits. Like all current VMI approaches FMA is adversely affected by the semantic gap. In order to address this situation a file system was built using FUSE [44], which translated the page table such that the memory of the VM was able to be read as one contiguous "file." The volatility framework for FMA is then used in order to analyze the contents of the memory. A rule learning algorithm called IREP++ [45] is used as a classifier in order determine if any intrusions are made into the system (such as kernel rootkits) were made and the results are logged into a postreSQL database. This is an interesting approach to side-stepping rather than attempting to directly solve the semantic gap. Volatility however like many VMI approaches still relies on a priori knowledge of the location of data structures inside the kernel which may leave this approach vulnerable to attacks which manipulate those structures.

Hay and Nance [46] developed a method for using VMI to read the plaintext for encrypted data with the VIX tool suite [8]. While direct attacks on modern cryptographic systems such as AES is generally computationally intractable three basic facts are noted: Before being encrypted cipher text exists in an unencrypted form, after being decrypted cipher text exists in an unencrypted form, and encryption requires that somewhere on the system cryptographic keys exist. To take advantage of the first two it is a simple matter of observing the state of VM while the plaintext is in memory. The third requires two steps; recover the key while it exists in memory and use that key as to recover the plaintext using the appropriate decryption algorithm. This use of VMI is an instance of security software which has great use for law enforcement and intelligence agencies as well as great potential to be misused as well.

1.7 Information Leakage and Side Channel Attacks

In this section we discuss the current state of research into information leakage and side channel attacks.

1.7.1 Side Channel Attacks

While a number of side-channel attacks have been explored in the past [47, 48, 49] a formal model of the information leakage due to these attacks was first introduced by Demme et al. [50]. Demme introduces the Side-channel Vulnerability Factor (SVF) in order to determine exactly how vulnerable a certain cross-channel attack makes a system.

The SFV begins with an oracle, which contains the truth about the execution of the victim. The side-channel produces the information that an attacker is able to measure. An example of an oracle could be number of accesses to a certain page and the side-channel could be the average power consumption on the host. A perfect side-channel would be able to trace the oracle trace directly. The SVF also requires a distance metric. The distance metric could vary from problem to problem. For instance if the data were represented by vectors the Euclidean or Manhattan distances could be used.

Next they establish a similarity matrix for both the oracle and side-channel traces. These similarity matrices are necessary since oracle and side-channel are by their very nature measuring different things (such as pages accesses versus cache access time).

Next they establish a similarity matrix for both the oracle and side-channel traces. These similarity matrices are necessary since oracle and side-channel are by their very nature measuring different things (such as pages accesses versus cache access time). The similarity matrix M is defined as being of length S and size $|S|^2$. Each element of M is defined as

$$M(i,j) = \begin{cases} D(S_i, S_j) & \textit{if } i > j \\ 0 & \textit{otherwise} \end{cases} \tag{1.2}$$

where D is our distance function. This creates a triangular matrix with no diagonal. The matrices are compared element wise and the Pearson correlation coefficient between the two is computed. The further from 0 the Pearson Correlation Coefficient is the more information is leaked through a side channel. A coefficient of 1 will mean the channel is perfectly transparent and a coefficient of 0 will mean that the channel is totally opaque.

This SFV will be able to provide us with a measure to tell how our different approaches to information leakage will be relative to each other. As well as a measure of how much information leaks from VMI relative to other types of information.

1.7.2 Determining Co-Residency

Ristenpart et al. [48] introduced a scheme for determining co-residence by measuring the load on the cache. In their paper they did a variant on the cache-probe technique which relies on the architecture of the x86 cache. It begins by allocating a buffer B of size b bytes, where s is the size of the cache line. Their initial attack is broken into three pieces

1. Prime: Read B at s byte offsets. (Ensuring that B is in cache)
2. Trigger: Wait until the number of CPU cycles passed jumps by some large value (to determine if the VM has been interrupted by the Xen credit scheduler)
3. Probe: Measure how long it takes to read B.

In step 3 B is accessed in pseudorandom order in order to prevent the hardware pre-fetcher from hiding the latency. These latencies correlate strongly with use of cache during the trigger step. However due to Xen's scheduling algorithm this is not quite enough to measure the cache latency. For that they expand the prime-probe attack even further to the following.

1. Allocate B contiguous bytes.
2. Sleep briefly (to build up credits with Xen's scheduler)
3. Prime: Read all of B to make sure it's fully cached
4. Trigger: Busy loop until the number of CPU cycles jumps by a large value
5. Probe: Measure how long it takes to read B

Through this load measurement and comparison to VMs running known services they are able to determine which VMs are co-resident.

Zhang et al. [49] put forth another scheme for determining co-residency of VMs on the Amazon EC2 cloud [51] by measuring the load on the cache. In this attack we consider two entities U and V each of which share a common cache. Then a similar prime-probe method as above is used though it is modified to function as follows

1. Prime - U fills a cache set by reading a region from its own memory
2. Idle - U waits for a specified interval during which the cache is used by V
3. Probe - U times the reading of the same cache set in order to learn of V's activities

In their initial trials the VM represented by U attempted to determine if V was co-resident by running one prime-probe trial and averaging the time across all cache sets. If this time was below a certain threshold a foe-absent classification was issued and if it was below that threshold a foe-present classification was issued. This proved to be extremely unreliable for two primary reasons: The Xen scheduler balances load by shifting VMs to different cores which may not share physical caches and because friendly VM activity on other cores, especially IO activity, will cause activity in the Dom0 which will introduce significant noise into the cache. In order to deal with this high level of noise a multi-probe classifier was implemented. In this classifier they looked at the results of 2000 prime-probe trials and noticed a pattern appearing to be two overlapping normal distributions. They then use these statistics to determine the classification.

In Owens and Wang [52] a scheme using the memory de-duplication techniques provided by commodity hypervisors (specifically ESXi [53] in this case). They begin by assuming that an attacker can instantiate VMs in the same environment as the targeted VM and that the attacker has root control over any VMs it instantiates. Further they assume the standard 4kb pages size. To begin their procedure they first determine which pages are unique to a specific OS version and are present in all versions of that OS. They must also determine which of those pages are relevant to their analysis, for example a memory dump of Windows XP

SP3 contains 59,238 pages of zeros [52]. These are not useful to the analysis. They are then able to determine which pages are signature pages for each OS.

At this point they take advantage of the de-duplication mechanism in ESXi. When ESXi writes a non de-duplicated page a simple write is performed. However when writing to a de-duplicated page a copy is made of that page and then it can be written to. The memory allocation and copy of the new page takes a measurable level of overhead. While there is overhead from the copy it can also be easily confused with the computational overhead from reading a page which has been swapped out having not been used recently. To handle this a read operation is performed on the signature page immediately before attempting an OS fingerprinting operation. They are then able to plot the timing results and can demonstrate which OS is being run on the host.

Xiao et al. [54] explored the implications of memory de-duplication on security in a more general sense by putting together both a reliable attack using this technique and a reliable defense against similar techniques. For their offensive approach Xiao shows that one can transmit data between two co-located VMs using the memory de-duplication.

They begin with two co-located VMs with a hypervisor which uses memory de-duplication. Each VM creates a number of identical pages. This can be easily accomplished by reading an identical file into memory. Then the sender encodes the information which it wants to transmit to the receiver. It does so by writing to certain pages and not to others. The sender and the receiver then sleep for some amount of time to allow the pages to be merged by the hypervisor's memory de-duplication mechanism. The receiver then writes to each of the pages and times the amount of time it takes to make these changes. Because of the Copy-On-Write (COW) nature of these systems it takes slightly longer to write to a de-duplicated pages as it makes a copy before writing to it. In this way memory can be encoded into bits. A page which has been merged representing a 0 and a page which has not representing a 1. So if both VMs allocate 8 pages in this manner and wishes to transmit 10011001 the sender would alter pages 1, 4, 5, and 8 so they would not merge.

In addition to transmission they were also able to determine that one was in a virtualized environment using the artifacts of memory de-duplication. A file is read into memory and then all pages in memory are written to and the write access times are recorded. Then the file is read in its entirety into two different memory locations. Since every page of the file has an exact duplicate in memory the hypervisor's de-duplication mechanism (if present and virtualized) is likely to merge the pages after an appropriate amount of time. Then one of the identical pages can be written to, if the environment is virtualized with memory de-duplication then the write access time will be significantly higher than the original write access times due to COW.

As a defensive mechanism against these attacks Xiao suggests a kernel runtime integrity scheme. There several hypervisor based mechanisms for determining the presence of kernel rootkits running in guests [55, 56]. These all require extensive knowledge of the kernel in order to bridge the semantic gap. Xiao [54] proposes examining the kernel for the sections of data, such as the syscall table, which are meant to be read-only. Inside the kernel binary read only data is designated by the .rodata section. This data is copied and read by a C program inside which holds an exact copy of the read only kernel data in memory.

A statistic gathered by Linux called the Proportional Set Size (PSS) is then used in order to determine whether kernel integrity has been threatened. The PSS value is the number of unique pages and the weighted number of the duplicated pages a process has. For instance a process with 100 unique pages and 100 duplicated pages would have a PSS of 150 [54]. This value is stored in the file /proc/$pid/smaps. A simple shell script can then determine if the PSS value has increased significantly to determine if kernel integrity has been violated.

These side-channels are directly applicable to determining whether or not a VM is being monitored by VMI. Linux uses a technique called Kernel Same Page Merging (KSM) to reduce memory commitment from processes in a manner similar to how ESXi de-duplicates pages for VMs. The KVM hypervisor treats all VMs as if they are Linux processes. As a result it is possible that identical pages between a VM and some Linux process can be merged thus causing a measurable delay when these pages are written to. If the VMI program happens to hold a page identical to one in a guest VM then it's possible that the VMI process can be detected through this delay.

Zhang et al. [49] introduced a scheme for reading cryptographic keys used by another VM sharing the same physical hardware and hypervisor. A similar prime-probe technique to was used, but this time specifically used on the instruction cache (icache) as opposed to the data cache (dcache). It does so with the standard icache technique introduced by Aciicmez [57]. A icache line is loaded with *NOP*s and then filled again with *NOP*s and the time difference between the two is noted. Further steps are needed however, as information from another VM is being sought the Xen scheduler has to be taken into account. To handle this inter-processor interrupts (IPIs) are used. In symmetric multiprocessing (SMP) systems processors are allowed to interrupt each other or even themselves through an IPI. To make sure the attacking VCPU has precedence another VCPU (called the interrupting VCPU or IVCPU) runs in a continual loop sending IPIs to the attacking VCPU.

This attack searches for instructions which are a square, a multiply, or a modulus-reduce instruction. In order to do so they use multiclass support vector machine [58] to pick out when these instructions are being used. While this is fairly good at picking out instructions the SVM is subject to the hardware and software noise introduced by the system (things such as TLB misses, or context switches.) To handle

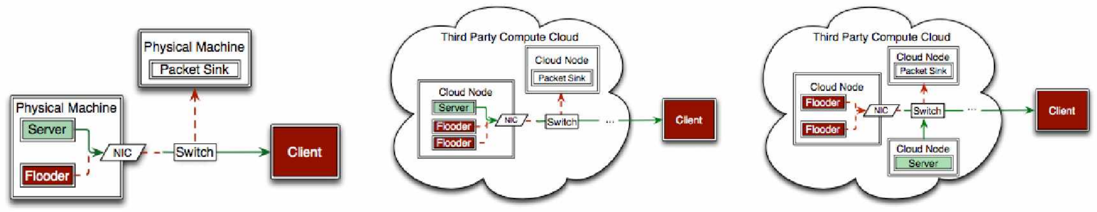

Figure 1.5: Topologies used in active traffic colocation test from Bates et al. [60] (left) local test system (center) successful co-location (right) failed co-location

this hidden Markov models [59] are used to filter the noise. This allows them to determine the cryptographic key in as little as 40,000-50,000 brute force attempts. While this may seem like a great deal we keep in mind that 50,000 brute force attempts will take less than 1 s on commodity hardware making this a reasonably effective attack.

These previous works have been directed almost entirely at exploring side-channel attacks aimed at the hypervisor layer. Bates et al. [60] take a lower level approach and investigate whether co-residency of one or more VMs on a hypervisor can be determined via active traffic analysis techniques. They begin by assuming that they are on a normal cloud instance such as EC2 [51] that has been patched against previous co-residency attacks [61, 48].

Their attack (fig 1.5) then creates a large number of instances on that cloud service, which they term flooders. Each flooder announces its presence to an external machine called the client. Another instance created in the cloud is called the server.

Upon receiving a signal from the server the client sends signals out to the flooders, at which point the flooders begin to send outbound traffic on their machines physical interface to a packet sink which is not the client. This outbound traffic causes a delay in the server flow.

These delays in server flow form a watermark in the signal from the client and the server. The network flow between the client and server can be given by T and divided into n segments of t_i. Watermarks of this kind require two different levels of packet delay to encode their signal represented by $\pm d$. Since negative delays are not possible in this environment they take no delay to represent $-d$ and a delay to represent $+d$. Using this scheme they are able to encode bit values using $-d$ as a 0 and $+d$ as a 1.

Due to the nature of virtualization and network traffic in general a certain amount of noise can affect the signal. These can come from sources such as network congestion or hypervisor scheduling (as described above in the Xen Credit Scheduler for instance.) When the client receives the signal a Kolmogorov-Smirnov (KS) [62] test is done for independence. If a signal is embedded in the traffic flow it will demonstrate a different discrete distribution from one without a signal. With the KS test they are able to determine a signal's independence with a 95% confidence.

This attack is interesting in that it operates at the hardware layer rather than at the hypervisor layer. It is possible that VMI targeted at the network stack of a VM (such as the VIX ifconfig [8]) could cause a change in the delay, which could be detected in the traffic. However this delay is unlikely to be substantial or consistent enough to useful to detect information leakage from VMI.

1.7.3 Defense Against Attacks

Martin et al. [63] introduced a scheme to protect against micro-architectural attacks by obscuring the way the *RDTSC* instruction works. Micro-architectural attacks such as the cache timing attacks described above [49, 48, 61] rely on precise timing of micro-architectural events in order to gather their information. Martin proposes a scheme by which they add two delays to the *RDTSC* counter. One, called the real offset, delays the execution of the *RDTSC* instruction. The other, called the apparent offset, which adds a random delay to the end of the *RDTSC* instruction. These delays can turned off in the OS so that system critical portions of the OS such as the scheduler are not interfered with. While this can be quite effective against certain types of attacks like those listed above it can be easily worked around if an attacker can gain kernel access (for example through a kernel module) and gives their malicious process rights to use the unaltered scheduler. While this may pose a challenge for some side-channel attacks their threat model differs from ours significantly enough that it poses no hindrance.

Chapter 2

Experiment Design and Analysis Techniques

2.1 Motivation and Goals

With the increase in the use of VMI [32, 8, 64, 3, 46] in research settings as well as the migration of VMI to the commercial sphere [65] the study of the security of this technique has taken on paramount importance. With the use of VMI to extract cryptographic keys from live memory [46] the dangers of misuse of VMI have gone from the theoretical to the practical.

In this dissertation our goal is to detect the use of VMI on a guest VM from within that VM. Our threshold for success will the answer to a simple yes or no question: "Can the guest VM detect that it is being monitored by some VMI agent?" Any results which exceed this threshold will also be taken as confirmation of detection of VMI.

2.2 Threat Model

We begin by defining the Trusted Computing Base (TCB) as the set of all hardware and software which is essential to the security of a computer system [42]. Vulnerabilities in the TCB will be considered vulnerabilities in the whole of the system. Components outside the TCB should not be able to elevate privileges that they are granted by the OS or hypervisor.

For the following experiments we will assume that the hypervisor as well as all associated interfaces such as libvirt or xencntrl are part of the TCB. All VMI agents will also be assumed to be part of the TCB. The guest VM will be outside of the TCB and therefore all malicious code must be executed on the guest VM. We further assume that the malicious VM is isolated from all other VMs.

The attacker on the malicious VM will be assumed to have root access to the VM and therefore will be allowed to install malicious kernel modules as well as run malicious user code.

2.3 Experimental Setup

The hosts in our experiments will use version 4.2 of Xen [20] and version 3.2.0 of KVM [24] along with version 1.6.2 of QEMU [66] (the userspace component to KVM). Guests are Ubuntu Server VMs with Linux kernel version 3.11.0-12-generic [13]. Guests are allocated 1GB of RAM and 1VCPU. Unless otherwise stated all VMs are clones of the original VM.

For experiments concerning the detection of VMI a simple system will be used where one guest VM is run on a physical host system (fig 2.1).

2.4 VMI Agents

For the detection of VMI all experiments will be done using VMITools [32]. Since the primary means of doing VMI is to copy a page from memory and interpret it [32, 8, 64, 3], any toolsuite which does this can be used for this experiment without the loss of generality. There will be three VMI programs used for these experiments: map-addr, process-list, and module-list.

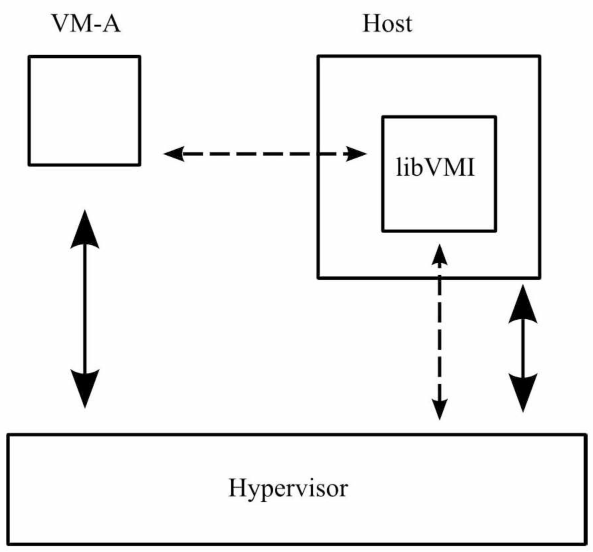

Figure 2.1: Experimental Setup

The map-addr program simply maps an address from the guest's memory to the memory of the VMI agent. The process-list command maps the processes currently running on a VM. In Linux the list of running processes is stored in the task-list [67]. The task list is a doubly linked list where each node in the list represents a process being handled by the OS. The process-list program begins by mapping the head node of the task list from the guest to the VMI agent. The first task struct is then decoded. The desired information, such as pids and process names, is displayed to the user and the location of the next task in the list is recorded. The next node in the task list is processed in the same way. This continues until the list comes back to the head of the task list, indicating that the entire list has been traversed (alg. 1). The module list works in much the same way (alg. 2).

```
Result: A list of the processes running on a target VM
current_task = Domain.task_list_head;
repeat
    adr=current_task.nextTask.adr ;
    map Task Struct at adr to host ;
    translate nextTaskStruct ;
    current_task = taskStruct at adr ;
until current_task.adr==Domain.task_list_head.adr;
```

Algorithm 1: The Process-List Program

```
Result: A list of the modules running on a target VM
current_module = Domain.module_list_head;
repeat
    adr=current_task.nextModule.adr ;
    map Module Struct at adr to host ;
    translate nextModuleStruct ;
    current_Module = ModuleStruct at adr ;
until current_module.adr==Domain.module_list_head.adr;
```

Algorithm 2: The Module-List Program

2.5 Experiments

In this dissertation we propose four experiments to determine if the guest VM has been monitored by some VMI agent.

For our first experiment we wish to analyze data produced by the Linux utility Sysstat [68]. Sysstat measures over 200 fields used by Linux such as page faults per second, context switches per second, and percentage of swap space used. Since the host, guest, and VMI agent all use the same physical resources it is possible that patterns can emerge in those values which can identify the use of VMI on the guest. This experiment also allows us to use an existing tool, which would be convenient for administrators who already run Sysstat.

For our second experiment we propose to look at main memory as a shared resource between the host and the guest. To determine whether this is viable we propose to analyze the time it takes to map and unmap a page in memory. With main memory, CPU cache, and the page table being shared between VMs, it's possible that the use of a VMI agent on the guest will cause a distinguishable difference in the time taken to do the mapping and unmapping of a page. It is hoped that this difference will allow us determine whether VMI has been used on a target VM.

Our third experiment is similar to the first experiment in that in-memory timing is used to determine if a guest is being monitored by VMI. In this case we are directly examining the CPU cache. If they are on the same socket a guest and a VMI agent will share the same L3 cache. If they are on the same core they will share the same L2 and L1 cache as well. For our experiment we propose measuring how long it takes to write an element in memory. Between each measurement we flush all three caches. We hypothesize there will be a small dip in the time required to access memory after it has been flushed from cache if a VMI agent has been accessing memory in that program.

Our fourth experiment takes advantage of the fact that KVM uses several features of the Linux kernel: in this case the scheduler and the KSM mechanism. KVM uses the Linux scheduler, in this case the Completely Fair Scheduler (CFS) [30], to manage VMs essentially as Linux processes. As a result of being treated as Linux processes, KVM VMs are subject to memory de-duplication using KSM [24]. We hypothesize that the memory de-duplication can be measured as in [54, 52] if a VMI agent copies a page from the guest's memory to the host's memory.

2.6 On Timing

In this chapter we will be discussing the analysis of a number of different timings. For these timings we used the C++11 chrono object [69]. The C++ chrono object offers a high-resolution timer, the resolution of which is dependent on the system. In Linux the high-resolution timer offers nanosecond resolution. In order to determine the resolution and any computational overhead for the chrono object we take a sample inside the guest VM. In this sample we simply take two time measurements one after the other and log the difference between them. We take 1,000,000 measurements like this and plot them; see fig 2.2. What we see is that

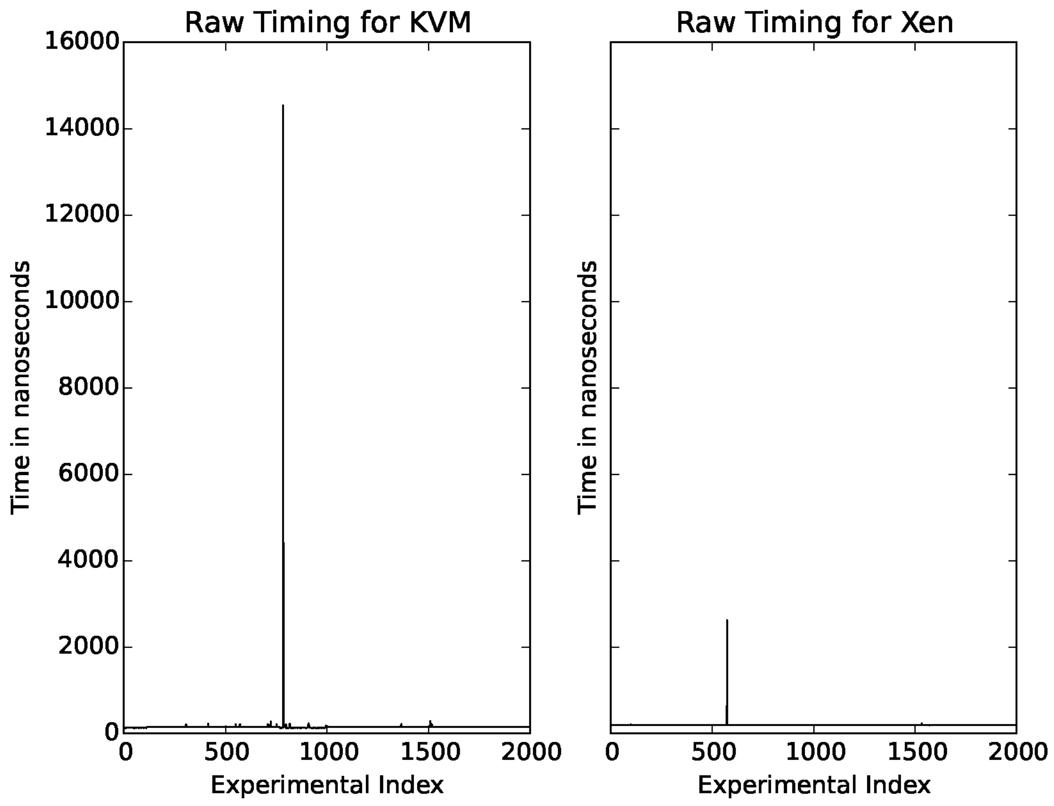

Figure 2.2: Raw Timing for Xen and KVM

while we have nanosecond resolution there is a significant amount of overhead involved in the start of a timing measurement. In order to compensate for this we take the minimum value of our measurements and subtract it from all subsequent measurements.

The minimum value is quite close in both KVM and Xen at 119 ns and 539 ns, respectively. These measurements represent the bare minimum amount of time it takes to perform a timing measurement so we can subtract it as overhead on subsequent measurements.

2.7 Analysis Tools Definitions and Terms

We begin our discussion of our common analysis techniques by defining the terms population and sample. A population is the complete set of data that have one property in common that is the subject of some statistical analysis. A sample is a subset selected from a larger population [70]. For example, we might consider a population as all people over 1.8m tall and a sample could be 1,000 randomly chosen people over 1.8m tall.

Next let us consider a sample X drawn from some population. Further let us define an individual element in X as x_i where $\{x_1, x_2, ..., x_{n-1}, x_n\} \subseteq X$. Next we assume that each x_i has the probability p_i. Further we assume that all values in X are independent and identically distributed (iid). That is, no element in X depends on another and all elements are pulled from the same distribution. We define the sample mean of X as [71]

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2.1}$$

This represents the "average" value of the dataset. We further define the standard deviation of the dataset [71] by equ. 2.2. The standard deviation gives a measure of how dispersed the sample is from the average. Finally we define the variance of X which is simply s^2.

$$s = \sqrt{\sum_{i=1}^{n} p_i (x_i - \overline{X})^2} \tag{2.2}$$

2.8 Hypothesis Testing

In this set of experiments we will be using statistical hypothesis testing in order to analyze our data. Hypothesis testing is the process of using a statistical test to determine whether a hypothesis about some model is false. We begin by defining the null hypothesis H_0 as some claim that we initially assume to be true [70]. The alternate hypothesis H_a is the claim that we assume to be true if we reject H_0.

When discussing hypothesis testing we will be comparing two random samples. The first sample is X where x_i has a probability of p_{xi} and $\{x_1, x_2, ..., x_{n_x-1}, x_{n_x}\} \in X$. The second sample is Y where y_i has a probability of p_{yi} and $\{y_1, y_2, ..., y_{n_y-1}, y_{n_y}\} \in Y$.

Using an appropriate statistical test (discussed in sections 2.8.1 and 2.8.2) we will be able to compute our test statistic. A test statistic is the result of a function of our data which gives us one number upon which we can base our rejection of H_0. Given the test statistic we can then compute our p-value. The p-value is the probability of obtaining a test statistic t, which is at least as contradictory to H_0 as the value obtained, under the assumption that H_0 is true [70]. That is, a p-value gives us the probability that our test statistic would have been produced if H_0 were true. The smaller the p-value, the more the data contradicts H_0. Note this is not the same as saying the p-value is the probability that H_0 is true nor is it the error associated with our test. Next we have the rejection region which is the set of values for which we reject H_0. If the p-value falls in our rejection region we reject H_0.

When we discuss hypothesis testing we must also discuss the types of errors associated with it. A false positive, also called a type I error, is when one asserts that H_0 is true when it is in fact false. A false negative, also called a type II error, is when the null is false but is not rejected. For brevity we will call the number of true positives generated by some process as TP, the number of true negatives as TN, the number of false positives as FP, and the number of false negatives as FN.

We further define the term accuracy as

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.3}$$

Accuracy gives us a measure of how close to the "correct" result the predicted results are. This will serve our metric for determining how good a classification is. It should be noted that accuracy gives equal weight to both types of errors. While this may be inappropriate for some purposes, such as those in judicial proceedings, it will be fine for the purposes of this dissertation.

2.8.1 Welch's t-test

The first hypothesis which will be used is Welch's t-test [72]. In Welch's t-test we look at the two samples and calculate a t-statistic. This is more appropriate for our purposes than the more common student t-test as it does not assume both samples

X and Y have the same variance. Based on this statistic we can determine whether or not to reject the null hypothesis.

We begin by computing the t-statistic to test the null hypothesis that the two populations X and Y share a mean via the following formula:

$$t = \frac{\overline{X} - \overline{Y}}{\frac{s_x^2}{n_x} + \frac{s_y^2}{n_y}} \tag{2.4}$$

Now that we have the t-statistic we can begin to compute the p-value. First however we need to determine the degrees of freedom ν for our test. For each of our random samples X and Y the degrees of freedom are given by $\nu_x = n_x - 1$ and $\nu_y = n_y - 1$. We can then approximate the degrees of freedom for Welch's t-test using the Welch-Satterthwaite equation [73, 72]

$$\nu \approx \frac{(\frac{s_x^2}{n_x^2} + \frac{s_y^2}{n_y^2})^2}{\frac{s_x^4}{n_x^2 \nu_x} + \frac{s_y^4}{n_y^2 \nu_y}} \tag{2.5}$$

With the degrees of freedom for the test we can compute the p-value by using the probability density function (pdf) for student's t-distribution.

$$f(t) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}(1 + \frac{t^2}{2})^{\frac{-\nu+1}{2}} \tag{2.6}$$

where $\Gamma(a)$, the gamma function, is defined as

$$\Gamma(a) = \int_0^\infty x^{a-1} e^x dx \tag{2.7}$$

We now obtain the p-value by integrating the pdf of student's t-distribution from t to ∞.

$$p = \int_t^\infty f(t)dt \tag{2.8}$$

With the p-value in hand we can determine whether or not to reject the null hypothesis by selecting a critical value. If the p-value is smaller than that critical value then we can reject the null hypothesis that the two distributions share a mean. For all t-tests in this dissertation we will assume a critical value of 10^{-6}. We will be performing all of our t-tests with the scipy library [74]. It should be noted that many of the p-values obtained in this dissertation are 0. For a finite dataset it is impossible to obtain a p-value of 0; a vanishing value of p is a limitation of the software and should be taken as less than 10^{-6}. We chose 10^{-6} as our critical value as it is an order of magnitude higher than the highest p-value we obtained experimentally. While these values may initially appear suspicious it should be noted they can easily be verified by performing the integral numerically using a different function in scipy [74]. It can also be intuitively verified if one looks at the student-t distribution for one degree of freedom (fig 2.3). We see that as x increases P(x) asymptotically approaches 0.

Finally we note that Welch's t-test assumes that the data being tested follows a normal distribution. However it was noted in [75] that the t-test is robust with

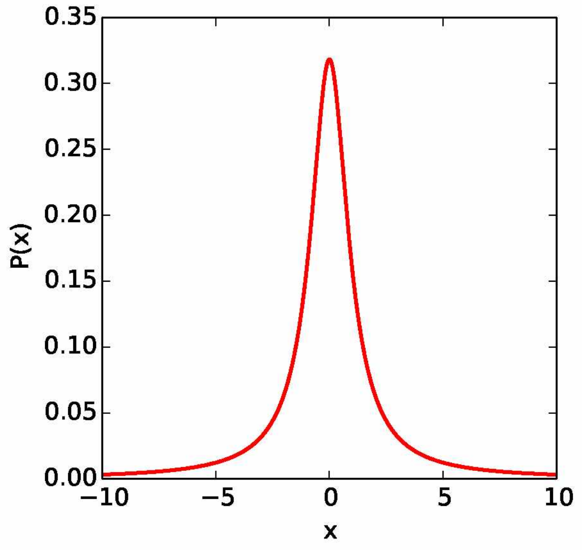

Figure 2.3: t-distribution for one degree of freedom

non-normal data. As such even though our data is not normal it can still be applicable.

2.8.2 Mann-Whitney U Test

The Mann Whitney U-test [71], also called the Wilcoxon rank-sum test, tests the null hypothesis that one sample tends to have larger values than another. We begin the Mann Whitney U-test using the same populations, means, and standard deviations as in Welch's *t*-test. Then we check to see whether or not the following four assumptions are satisfied.

1. X and Y are independent of each other

2. All observations are ordinal (i.e. it can be distinguished that one observation is greater than another)

3. Under the null hypothesis the distributions of both populations are equal

4. Under the alternate hypothesis the probability of an observation from one population exceeding an observation from the other is not 0.5.

In order to perform the test we compute the U-statistic. There are two methods for computing the U-statistic: one for small data sets (less than 20 elements) and one for larger data sets. As all of the datasets used in this dissertation are on the order of 10^6 elements we will only be discussing the latter.

The first step is to rank all the observations. The smallest observation is assigned the rank 1, the next smallest value is assigned the rank 2, and so on. Ties are equal to the midpoint of the assigned rankings. So in the set $\{2,4,4,7\}$ the ranks $\{1,2.5,2.5,4\}$ would be assigned.

The next step is to sum the ranks of all observations taken over X and call it R_1. The sum of the ranks from observations taken over Y is given by R_2. We can then then compute the U-statistics

$$U_1 = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1 \tag{2.9}$$

$$U_2 = n_1 n_2 + \frac{n_2(n_2+1)}{2} - R_2 \tag{2.10}$$

where n_1 and n_2 are the number of elements in R_1 and R_2, respectively. When adding equ. 2.9 and 2.10 together we get the sum as

$$U_1 + U_2 = 2n_1n_2 + \frac{n_1(n_1+1)}{2} + \frac{n_2(n_2+1)}{2} - R_1 - R_2 \tag{2.11}$$

Since we know that $R_1 + R_2 = \frac{N(N+1)}{2}$ where $N = n_1 + n_2$ we can expand equ. 2.11 to

$$U_1 + U_2 = 2n_1n_2 + \frac{n_1(n_1+1)}{2} + \frac{n_2(n_2+1)}{2} - \frac{(n_1+n_2)(n_1+n_2+1)}{2} \tag{2.12}$$

Equ. 2.12 simplifies to

$$U_1 + U_2 = 2n_1n_2 + \frac{n_1(n_1+1)}{2} + \frac{n_2(n_2+1)}{2} - (\frac{n_1(n_1+1)}{2} + \frac{n_2(n_2+1)}{2} + n_1n_2) \tag{2.13}$$

Simplifying see that $U_1 + U_2 = n_1n_2$. This allows us with a bit of algebra to simplify equ. 2.9 and 2.10 slightly to

$$U_1 = R_1 - \frac{n_1(n_1+1)}{2} \tag{2.14}$$

$$U_2 = R_2 - \frac{n_2(n_2+1)}{2} \tag{2.15}$$

The smaller of U_1 and U_2 is chosen for computing the p-value which are obtained through a table of critical values [71]. As with the t-test we will be using the scipy implementation [74] for our Mann Whitney U-tests.

If we assume the null hypothesis that the samples taken when the VM is monitored by VMI and when it is not monitored by VMI have the same mean, we can determine via these two tests whether or not to reject this hypothesis.

2.9 Mutual Information

Mutual information is a quantity which measures the amount of certainty gained about a population X when measuring some random variable Y. In particular let $Y \in 0, 1$ represent whether or not the VM is monitored by some VMI agent and let $X \in \mathbb{R}^n$ represent the measured data from a given experiment. The mutual information [76] then specifies how many bits of information are gained about Y when sampling the random variable X. Given a joint probability distribution $P(Y = y, X = x)$, when the mutual information is given by

$$I(Y : X) = \sum_{x \in X} \sum_{y \in Y} P(y, x) log_2 \left(\frac{P(y, x)}{P(x)P(y)} \right) \tag{2.16}$$

To estimate the mutual information, we use the implementation provided by scikit-learn [77] where X is approximated by histograms over a large number of sample measurements. Mutual information will allow us to determine how much samples are needed in order to make a classification between two samples.

In this dissertation we will be using the mutual information to determine the fewest number of measurements we can take in order to correctly classify whether or not a guest has been monitored by some VMI agent.

Chapter 3

Sysstat Experiment

3.1 Introduction and Motivation

In our first chapter investigating methods of detecting VMI we first ask "What are the shared resources we need to investigate?" and "How can we get the information we need from these shared resources?" Since all physical hardware is shared between the host and the guest we have an abundance to choose from such as memory, CPU, disk, and the network. We can get this information directly from the OS itself as a fair amount of information is recorded by the OS. In this experiment we will take that information provided by the OS and analyze it to determine if it can yield information about the use of VMI on that guest.

3.2 Sysstat

Modern OS's log a large amount of information for performance and monitoring purposes such as the page fault rate, CPU frequency, and disk IO rates. In the Linux OS a program called Sysstat [68] makes this information easily available to the user. In this experiment we attempt to analyze the data produced by Sysstat in order to determine if a VM is being monitored by a VMI agent. We hope the use of an extant tool like Sysstat will allow easy monitoring of this field by a system administrator and will require minimal setup on their part.

3.3 Experiment

We begin our experiment, which we call the SAR Experiment, with the same apparatus as described in section 2.3. For our first step we synchronize the clocks on the host and guest. We do this using the NTP protocol [78] available in Linux. Synchronizing the clock on the host and guest allows us to compare measurements taken by Sysstat on the guest to times when a VMI agent was used by the host.

On the guest Sysstat is run to gather all of the data it's capable of gathering. The interval is set to 1s as it is the smallest measurement Sysstat can take. One hour of data was taken. The command used to gather the data is

```
sar -A 1 3600
```

On the host we run a script called *collectData.py*. This script runs a VMI program specified by the user and at a time interval also specified by the user. For our experiment we run the VMI programs *process-list*, *module-list*, and *map-addr*. We do our measurements at intervals of 100 s, 50 s, 25 s, 10 s, and 5 s. Each time a measurement is made the time stamp is noted.

3.4 Data and PreProcessing

The data in Sysstat is recorded in a difficult to read binary format. However if one captures the live output of Sysstat (as we did in our experiments) it becomes human readable and far easier to parse. After being parsed it still requires a fair amount of preprocessing. To begin with many of the fields are non-numeric such as the network interface being monitored or which CPU is being monitored. We can immediately discard these fields as non-numeric fields will be significantly more difficult for us to use machine learning techniques on, in addition since we only have one of each interface (one CPU and one network adapter for instance) it is unnecessary to keep this information.

The next portion of our preprocessing involves removing of all fields which are uniformly zero. These are removed as they do not contribute any information to our classification, yet they can still add to the computation time required to perform our classification. The question is why are some fields uniformly zero? For instance major faults per second (fig 3.1)(those which require going to disk for a page) is uniformly zero yet page faults per second (total page faults including major faults) is not (fig 3.2). The fields which are uniformly zero are not necessarily the same for Xen and KVM. These fields are also not necessarily zero when the measurements are taken on a non-virtualized OS. This implies that the fault to the hypervisor caused in some cases (such as in the case of a major fault) the measurement to be reported as a 0 to the OS.

Further complicating matters is that the majority of the measurements taken by Sysstat are not of the same unit. This poses two problems: you cannot directly compare measurements of different units and that different measurements are often of different scales. For instance you cannot compare amperes and meters as one measures electrical current and one measures distance. Further an every day

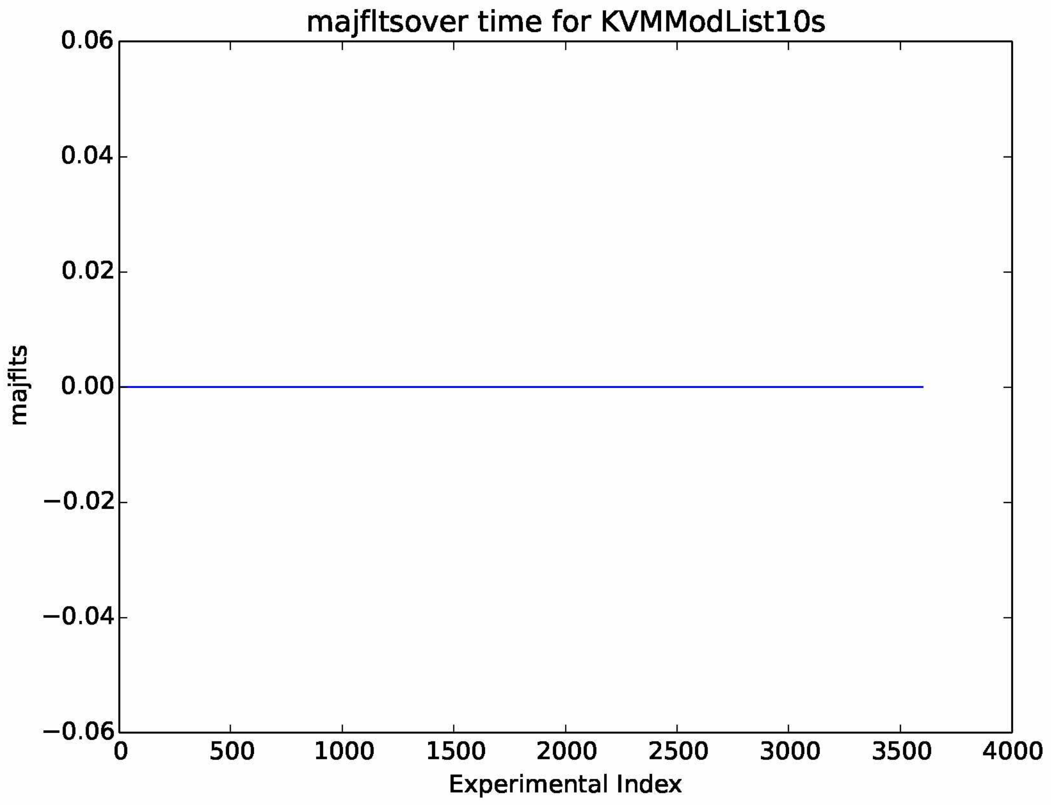

Figure 3.1: A graph of major faults per second over time for KVM while monitored by VMI every 10 s

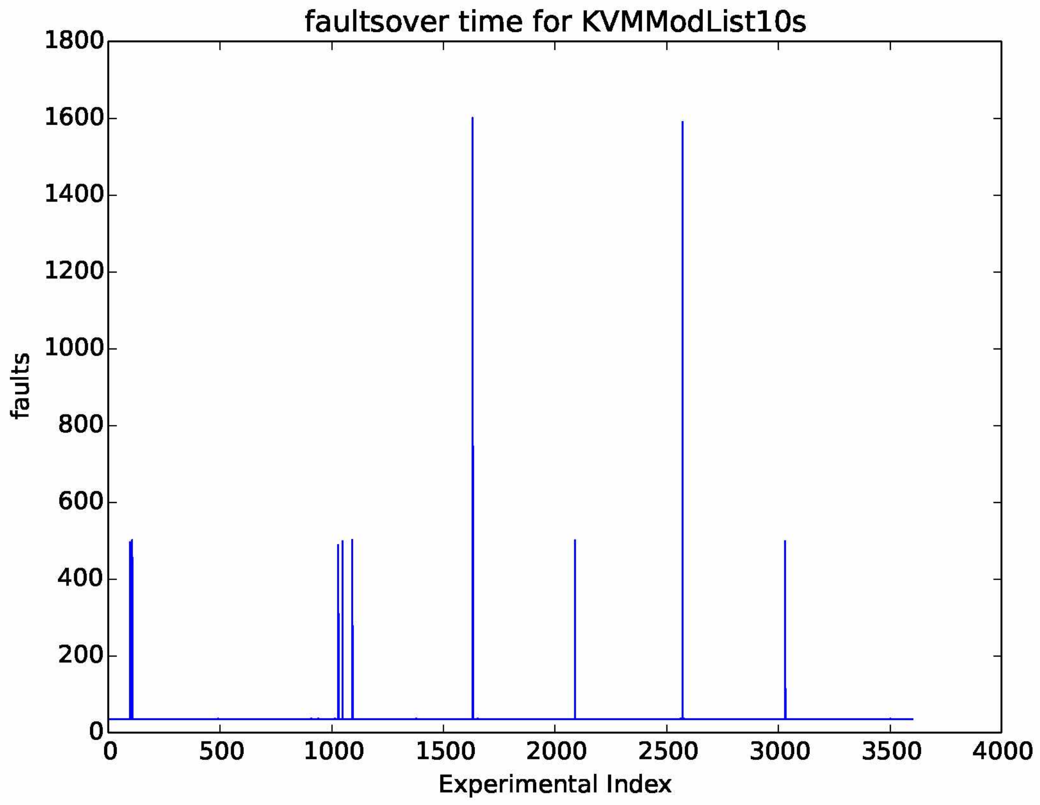

Figure 3.2: A graph of page faults per second over time for KVM while monitored by VMI every 10 s

measurement of current may be on the order of 10^{-3} A but measurements of distance might be on the order of 1 m. So while a change of 10^{-3} might be insignificant for a measurement of distance it might be very significant for measurement of current. To address this problem a common technique called standardization (which yields a Z-score) is used. To compute the Z-score of a data set we first split the data set into features. A feature is a type of measurement in our data set such as page faults per second. Since our data is conveniently divided into fields we will use each field as one feature. For each feature we compute the mean ($\overline{X}$) and the standard deviation (s). Then for each datum x in the feature we compute and replace it with Z as defined in equ. 3.1.

$$Z = \frac{x - \overline{X}}{s^2} \tag{3.1}$$

After the features are scored we must classify them. To do so we compare the time stamps taken by Sysstat on the guest and the time stamps taken when VMI was run on the host. Data points from the guest that are within 0.5 s of a time stamp noted on the host are marked as being monitored which we denote with a 1. All other points are marked as unmonitored denoted by a 0.

After processing the data we still had more than 150 features available each with 3600 measurements which can be quite a large dataset when using machine learning algorithms which suffer from the so-called "curse of dimensionality" [79].

So we try to remove more of the features which may not be of interest to us or which may be redundant.

3.5 Analysis

3.5.1 Information Gains

Suppose we have a dataset S with s_i samples of class i and m classes total (in our case two for monitored and unmonitored). The amount of information needed to classify a sample is given by

$$I(s_1, \ldots, s_m) = -\sum_{i=1}^{m} \frac{s_i}{\|s\|} log_2 \frac{s_i}{\|s\|} \tag{3.2}$$

Now let us denote a feature by F. A feature F is made up of ν subsets $\{s_1, s_2, \ldots, s_\nu\}$ where s_j is the subset of F with the value f_ν. Now we let s_j contain s_{ij} samples of class i. Classes are broadly any way we can classify data; though for our purposes the classes will be whether a VM was monitored or not by VMI. We can then compute the entropy of the feature with equ. 3.3

$$E(F) = \sum_{j=1}^{\nu} \frac{s_{1j} + s_{2j} + \ldots + s_{mj}}{\|s\|} I(s_1, \ldots, s_m) \tag{3.3}$$

The information gain is then computed as

$$Gain(F) = I(s_1, \ldots, s_m) - E(F) \tag{3.4}$$

Table 3.1: Features and information gains for our top 10 features

feature	info gain
membuffer	0.2002
memcache	0.1475
wtps	0.1351
memfree	0.1350
memusedpercent	0.1281
pgfree/s	0.0531
sys	0.0328
swapusedpercent	0.0324
idle	0.0312
pgscank/s	0.0271

Using the information gain we are able to select the features which contribute the most information to classifying the datum. We select the features which have the 10 highest information gains and use those for our analysis. These features are all CPU or memory related.

Table 3.2: Accuracy Rating for Xen monitored by the process-list command

times	100 s	50 s	25 s	10 s	5 s
Linear SVM	0.991	0.9833	0.958	0.963	0.802
SVM	0.990	0.983	0.958	0.963	0.802
SGD	0.990	0.016	0.958	0.963	0.802
KNN	0.990	0.991	0.973	0.984	0.892
Nearest Centroid	0.990	0.991	0.973	0.985	0.892
Running Tree	0.988	0.984	0.973	0.963	0.884
Gradient Boosting	0.988	0.990	0.966	0.973	0.833

3.5.2 Classificaton

Next we split our data into testing and training sets. To do this we use the function *train_test_split* provided by *scikit-learn* [80]. We split our data into 60% for our training data set and 40% for our testing data set.

With our testing and training set randomly chosen we can begin the classification of our data. We begin by looking at the available classifiers provided by scikit-learn. We ran tests using the Linear Support Vector Machine Classifier, Regular SVM classifier, GD classifier, KNN classifier, Nearest Centroid, Tree Classifier, and Gradient Boosting Classifier. The results are shown in table 3.2. Results from KVM are not shown but are analogous.

Table 3.3: Raw scores for Xen with the linear support vector machine classifier

times	100 s	50 s	25 s	10 s	5 s
True Positives	0	0	0	0	0
False Positives	0	0	0	0	0
True Negatives	1427	1416	1380	1386	1156
False Negatives	13	24	60	54	284

While it does appear that we get extremely high accuracy with each of our classifiers (as high as 99.1% accuracy) the accuracy decreases significantly the more frequently VMI is used on the target VM. When we look at the raw scores (table 3.3) we see a slightly different picture.

We see that there are no actual positives marked by our classifier. The classifiers are marking each of the points as negative and thus decreasing the accuracy when more positive points are included. The implication of this technique is that it will not be useful for determining whether or not we have been monitored by VMI. Why is this likely the case? Measurements taken by Sysstat are at their finest grain 1 s, the measurements taken by our VMI tools happen on the order of 10^{-6} s. It therefore appears that a finer grained measurement will be needed to determine if our guest VM has been monitored by VMI.

Chapter 4

Using Page Allocation Timings to Detect VMI

4.1 Motivation

In the previous chapter we were unable to detect VMI use with Sysstat as it proved to be too coarse grained. In this chapter we look at the shared resource of main memory, which is shared by both hosts and guests across the entire system. In addition events in memory occur on the order of 10^{-7} s which would be fine enough grained to pick up VMI. In the x86-64 processor main memory is handled by the Memory Management Unit (MMU), which uses a process called paging to control which data and instructions are held in main memory.

When a VMI program is used on a guest, memory is mapped from the memory space of the guest to the memory space of the host (which, for brevity, we will refer to as guest-space and host-space respectively). We believe this mapping from guest-space to host-space will affect which pages are in memory to a degree that is measurable in the time required to map a page in memory.

4.2 Experimental Design

For our initial experiment, which we call the MMap Experiment, we aim to determine whether or not a guest VM can detect that it is being monitored by a VMI process. We begin by using the same experimental hardware described in chapter 2. For our experimental setup we use a KVM and a Xen host. Each host runs a single VM of Ubuntu 14.04 as described in chapter 2.

On the host the VMI agent was run continuously. We did three trials: one where the process-list command was run, one where the module-list command was run, and one where the map-addr command was run. In each case the VMI program is continually run on the guest VM.

On the guest side a probe is set up. This probe uses the C++11 chrono object [69] which gives us nanosecond resolution. For each iteration the time stamp is recorded, a page is mapped and unmapped from memory using the mmap function [81], and then the timestamp is again recorded. The difference between the second time stamp and the first time stamp are taken and this result is recorded as the time taken to map and unmap a page. As mentioned earlier however a small correction is made and the overhead time in the previous section is subtracted from the result to give our final result. A control sample where no VMI agent is being run on the guest is also taken.

1. Mark Timestamp t_0
2. Map page in memory
3. Unmap page in memory
4. Mark Timestamp t_1
5. Record t_1-t_0

4.3 Results and Analysis

The first step of our analysis is to compare the histograms of the control data with data where VMI is used. It can be seen immediately (figs 4.1 and 4.2) that not only are the samples with VMI different from the control sample but they are also different from each other as well. One should note that these histograms are zoomed in to give a better insight into the data. There are still small pockets of data after the 10,000 ms bin but these are so small as to be not evident in the histograms. The next step is to determine whether or not these datasets are statistically different from the data in the control sample. To do this we use two statistical tests, which determine whether or not two populations share the same mean. The first is Welch's t-test [72] which determines whether the mean of two populations is the same. The second is the Mann Whitney U-test which again measures whether the two means of the population are the same.

In both cases we begin with the null hypothesis that the mean of a sample not being monitored by VMI is the same as the mean of a sample which was being monitored by some form of VMI. The results of these tests are shown in table 4.1. As we can see the results of the likelihood of the means of the two populations being the same is extremely low. It should be noted here that while all of the p-values are 0 this is strictly speaking not possible for finite populations. Instead this is a limitation of IEEE floating point arithmetic and a value of 0 should be taken as a value of less than 10^{-6}.

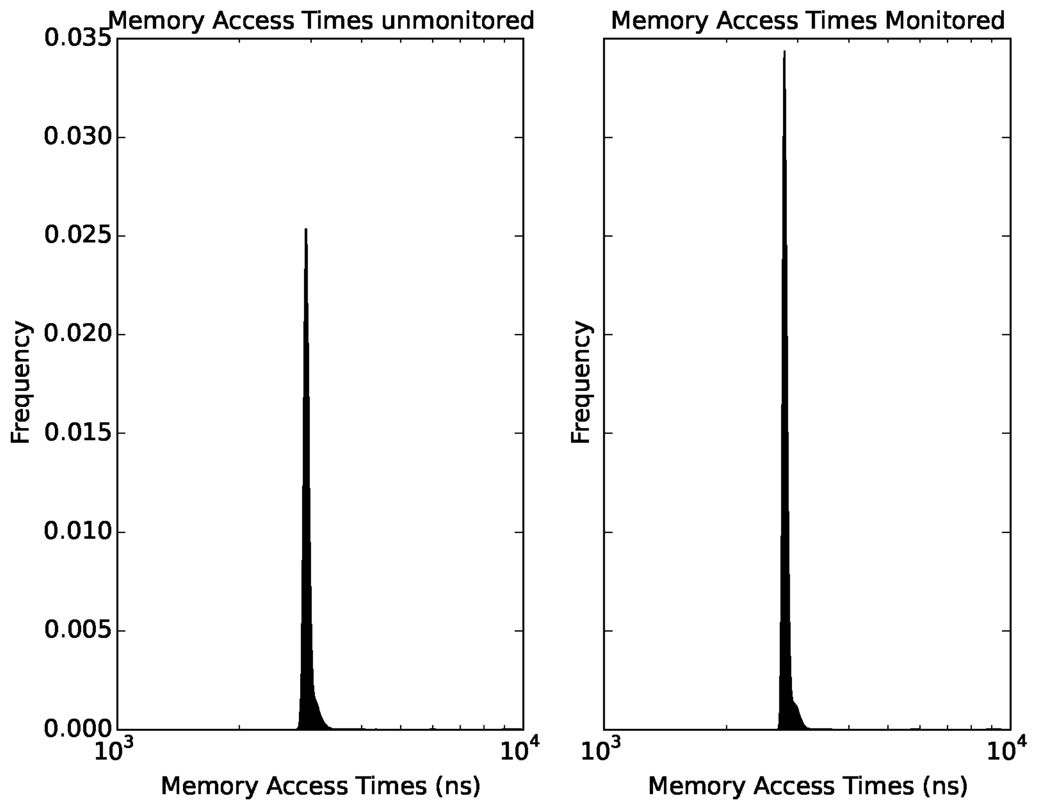

Figure 4.1: Histograms of the mmap time when a Xen VM is not observed by VMI (left) and observed by the process-list command (right)

Table 4.1: *t*-stats for Xen and KVM compared to the null hypothesis that no VMI is being used

Hypervisor	mapPage	modList	procList
Xen	-4361	-5678	-1691
KVM	-903	-1000	-632

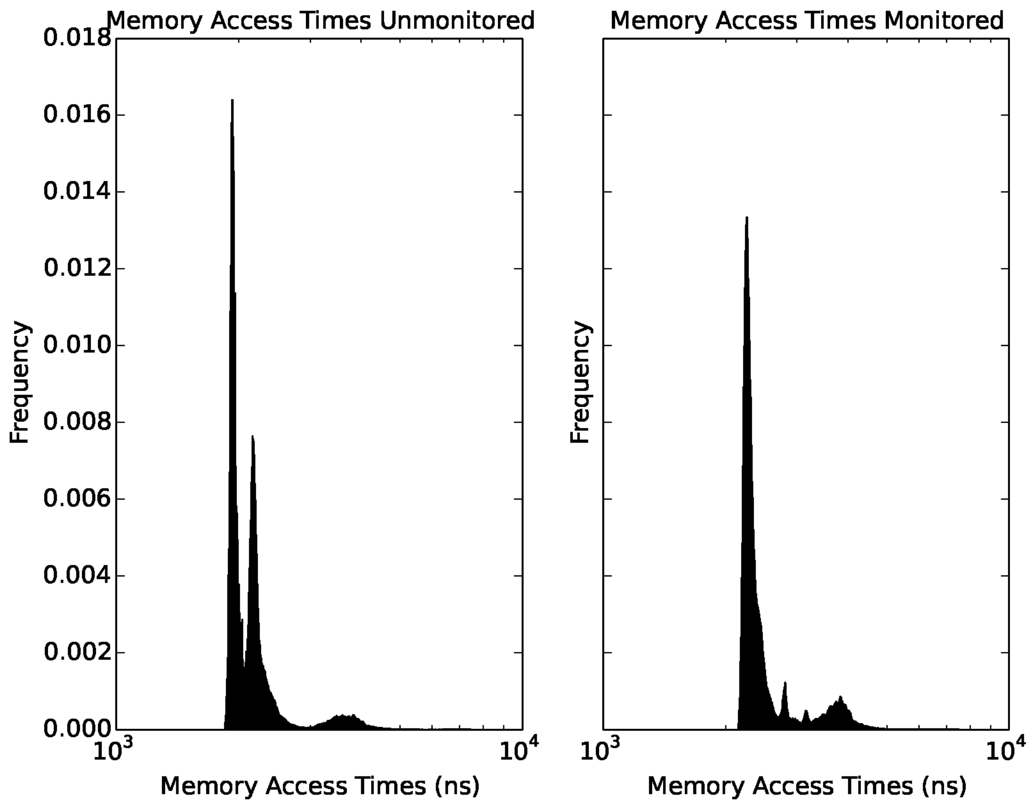

Figure 4.2: Histograms of the mmap time when a KVM VM is not observed by VMI (left) and observed by the process-list command (right)

Table 4.2: U-stats for Xen and KVM compared to the null hypothesis that no VMI is being used

Hypervisor	mapPage	modList	procList
Xen	$3.73 \cdot 10^8$	$8.71 \cdot 10^8$	$7.43 \cdot 10^8$
KVM	$8.34 \cdot 10^8$	$1.02 \cdot 10^9$	$3.26 \cdot 10^8$

The next question to arise is this: are these patterns unique to VMI or can they be reproduced by other means? To answer this we first consider which factors can impact time taken to map a page are page faults and cache misses [82]. At the scale of time being dealt with in this experiment, cache misses will likely not be a significant factor given that they tend to be in the 100 ns range. As a result we will instead focus on page faults.

4.4 Elimination Experiments

For this portion of the experiment we will test whether a VM with more memory will cause similar patterns in the time it takes to map a page to those caused when a VM is being monitored by a VMI agent. We begin this portion of the experiment by cloning our initial VM but changing configuration so the VM has 4GB of RAM instead of the 1GB of RAM it had earlier. The probe process as described in section 4.2 is repeated. The results are compared to the null hypothesis that the target VM is being monitored by the process-list program.

We can see that these histograms are distinctly different (fig 4.3). We then compare the two samples using the Mann-Whitney U test and the t-test. We can see that the p-values are well outside our critical range and thus we reject the null hypothesis that two samples are the same.

Next we test whether a VM which has more VCPUs can be confused with the signal of a VMI agent. We begin this portion of the experiment by cloning our initial VM but changing configuration so the VM has 3 VCPUs instead of the one

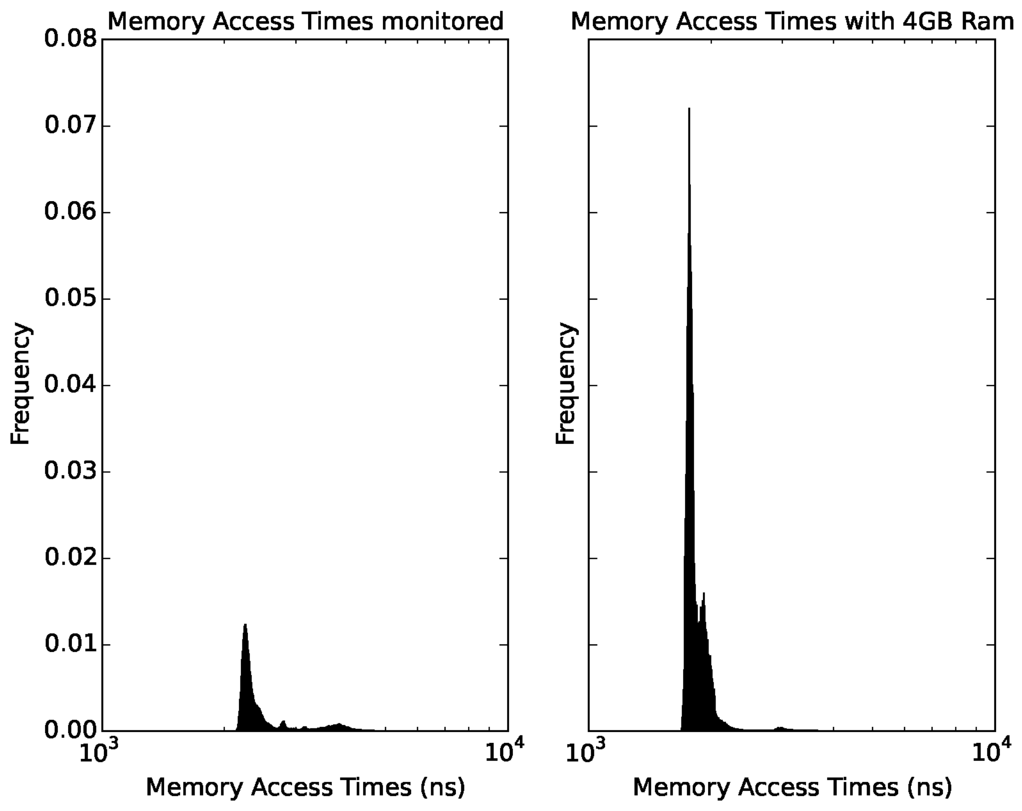

Figure 4.3: Histograms of the mmap time when a KVM VM observed by the process-list command (left) and when a KVM VM has 4GB of RAM (right)

it had earlier. The probe process described in section 4.2 is repeated. We again make the null hypothesis that the target VM is being monitored by the process-list program.

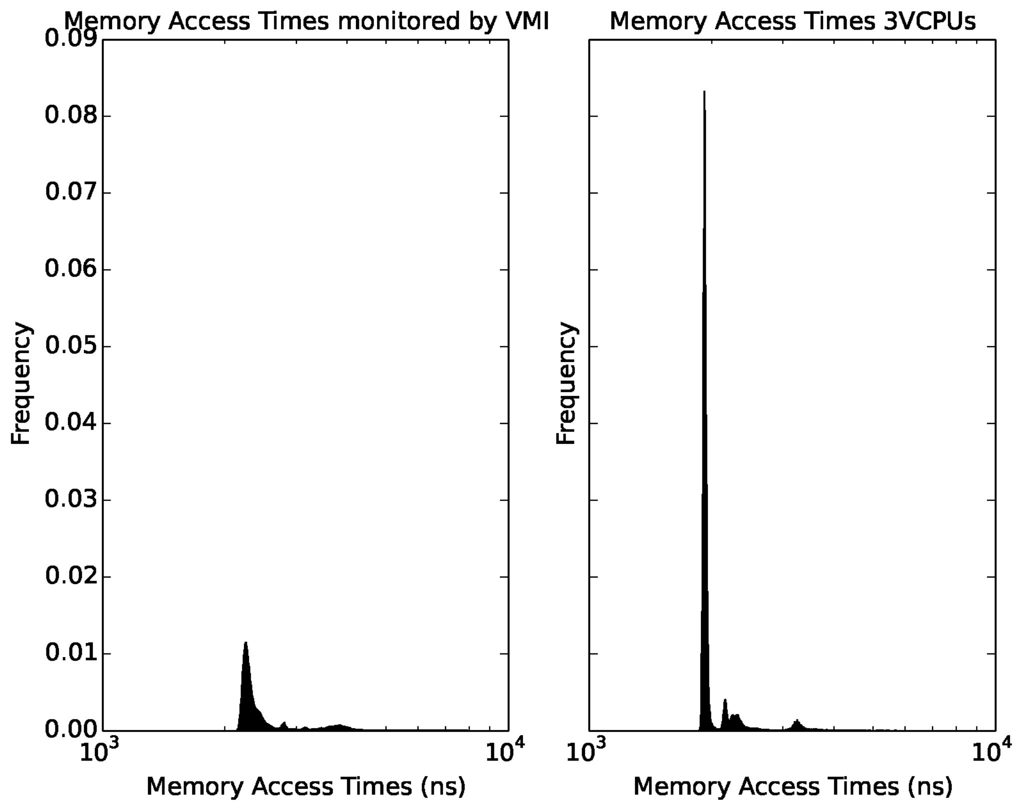

Figure 4.4: Histograms of the time when a KVM VM observed by the process-list command (left) and when a KVM VM has 3VCPUs (right)

We next attempt to determine whether or not the number of VMs running on the same host will produce a signal similar to the one produced by a VM being monitored by a VMI agent. We begin our experiment by making three identical clones of our initial VM (now called VM-A) which will be labeled VM-B, VM-C,

Table 4.3: U-stat and t-stat for populations taken when the VM had 4GB of RAM or 3VCPUs compared to the null hypothesis that they were being monitored by a VMI agent.

Machine Configuration	Mann-Whitney	t-test
3VCPU	$1.184 \cdot 10^7$	174.0
4GB RAM	$1.373 \cdot 10^7$	147.1

and VM-D as shown in fig 4.5. We run an experiment initially where only VMs A and B are running. The probe described in section 4.2 was run on VM-A while VM-B was idle. We repeat this process with VMs A-C running and again with VMs A-D running. We then compare each of these samples taken to the samples when a VM is being monitored by VMI 4.6.

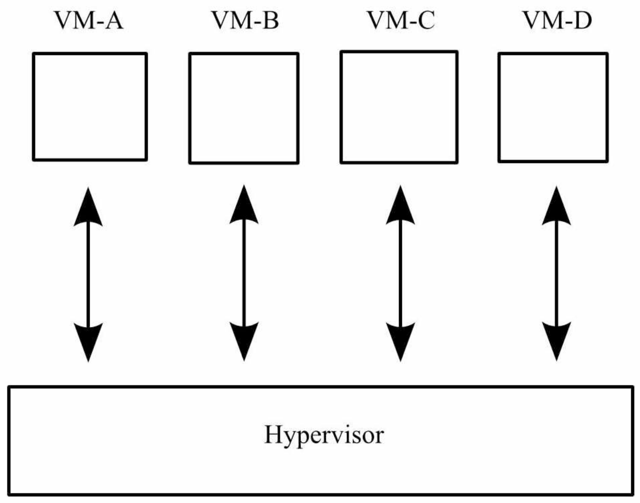

Figure 4.5: Diagram of multi-VM experiment

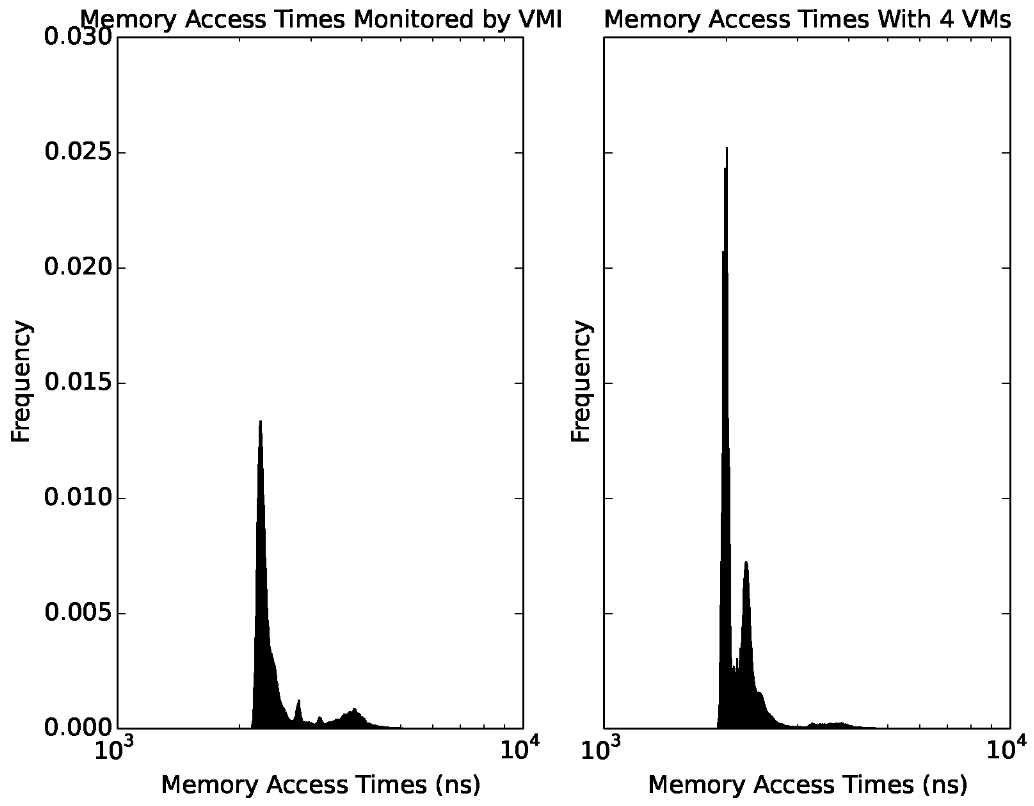

Figure 4.6: KVM monitored by the Process-list command (left) and KVM running 4VMs (right)

4.5 Conclusion

In this chapter we successfully introduced a method for detecting VMI as well as distinguishing between which VMI agent was used. It can also distinguish between multiple VMs being used as well as different levels of resource allocation. While it is impossible to exhaustively test all scenarios we consider this experiment to be a success as it did detect VMI. However it was not without its limitations. The VMI agent had to be run continuously without break on the VM in order for this method to be successful. This means that while successful in a technical sense it isn’t especially realistic.

Chapter 5

Using Cache Timings to Detect VMI

5.1 Motivation and Introduction

In this chapter we discuss a statistical analysis of the time taken to write to a page after it has been forcibly ejected from cache. When we look at resources shared between a host and guest, one of the obvious ones is the CPU cache. When a VMI agent fetches memory from a guest VM it ends up in the CPU cache. The question is can this caching be measured by the guest VM? We hypothesize that there will be a measurable decrease in the time taken to access a page at intervals where VMI has been performed on the VM. In order to make this more general we aim our experiment at the L3 cache specifically. While each core has its own L1 and L2 caches, the L3 cache is shared across all the cores on a CPU. This will mean we will be able to tell if a VM has been monitored by VMI agent regardless of which core it's on.

5.2 Experiment

We begin this experiment, which we call the Cache Experiment, with the same physical setup described in chapter 2. Since this experiment will be specifically measuring the L3 cache timings we don't need to pin the hypervisor and VM to the same VCPU. This experiment is broken into two portions: the monitoring portion and the host portion.

When a VMI agent is used on a VM the memory being analyzed is copied or mapped from the VM to the VMI agent. This will move this data into the CPU cache of the CPU on which the VMI agent is being run. Suppose the VMI agent were co-located on the same CPU as the VM being monitored. This would mean that the VMI agent would be able to fetch the memory from cache rather than having to go out to main memory. This would cause a significant decrease in the time taken to fetch the data. Now suppose that the process inside the target VM which is being monitored by the VMI agent flushes cache lines which its memory occupies. This should cause a measurable increase in the amount of time required to access the data which has been evicted.

For this experiment a process (the monitoring process) will be run on the guest VM. This process begins by allocating a page of memory and fill it with random values using the Mersenne Twister algorithm [83]. The Mersenne Twister pseudo-random number generator is chosen to increase the probability that page will be distinct from all other pages in memory. This is to ensure when the data is evicted from cache it will not be inadvertently loaded by other processes, which have the same data in memory. While most pseudo-random number generators would be acceptable the Mersenne Twister is common and included as part of the new C++ standard library.

The monitoring process will then take one time stamp using the chrono object described in section 2.6, write to a random element in the page, then take another time stamp, and record the difference between the timestamps. The monitoring process then flushes the cache using the X86 *CLFLUSH* instruction [84]. This instruction flushes a cache line from all levels of cache on the CPU. Since the *CLFLUSH* instruction only flushes a cache line the page is simply iterated through until all cache lines are flushed. This process is then repeated 1,000,000 times.

On the host side the VMI agent will fetch the memory from that process and page. The VMI agent was run at regular intervals during the experiment. Trials were performed where the VMI agent was run every 50 μ s, 100 μs, 200 μs, 1 ms, and 10 ms. Trials were also performed where the target VM was not monitored by a VMI agent.

5.3 Analysis

The result were plotted and it was expected that noticeable drops in the time taken to write to a page would be present at regular intervals where the VMI agent had mapped the guest data. From fig 5.1 we can see that these drops in the memory access time are not present. Why is this the case? According to Hennesy and Patterson [85], cache misses take approximately 25 ns. This is a full order of magnitude less than the overhead taken for our timing measurements. Can anything be salvaged from these results?

When one looks at a histogram of the results one can see that the histograms are slightly different so we employ our statistical tests to determine if they can be distinguished that way. For the initial set of tests the *t*-test and Mann-Whitney

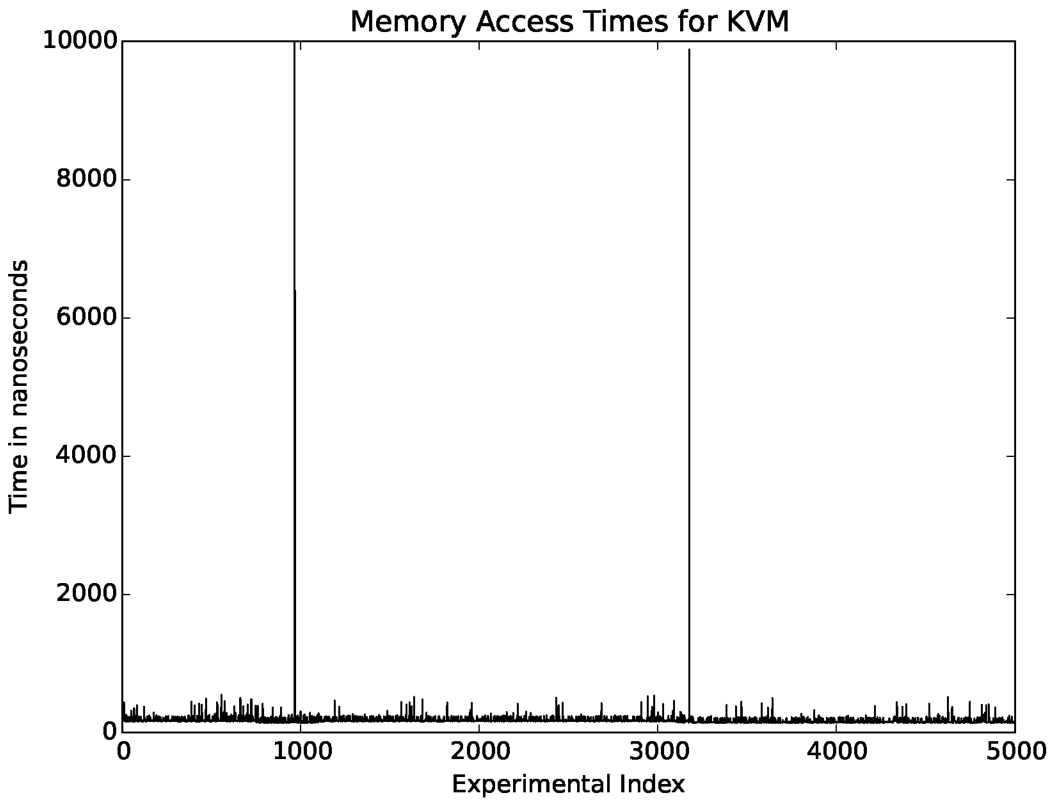

Figure 5.1: KVM memory access timings (ns) zoomed to show 5,000 results

Table 5.1: T Tests for Xen and KVM vs the null hypothesis that no VMI has been used

hypervisor	50 μs	100 μs	200 μs	1 ms	10 ms
Xen 10^6 Samples	-9.196	-44.15	-18.40	-4.742	-32.22
Xen 100 Samples	-40.66	-29.62	-32.62	-28.71	-34.00
KVM 10^6 Samples	-12.25	-12.33	-11.29	-12.25	-11.36
KVM 100 Samples	-32.63	-34.63	-3.159	-33.29	-13.43

U-test are run between pairs of populations with the control population being unmonitored by VMI and the variable population being monitored by VMI at some regular interval. Table 5.1 shows the results. In all the tests performed the p-value was less than 10^{-6}. A p-value this small indicates that we can reject the null hypothesis that both samples were drawn from the same population. As a result we can take conclude that VMI has been detected by this probe. In these tests 1,000,000 samples were used. This is an extremely large sample size and may not be useful for real time applications. The question now becomes: how far can the sample be reduced and produce acceptable results?

To determine this, we model the experiment as two random variables and considering the mutual information between them as discussed in chapter 3. The results of the mutual information between the timing data and the presence or absence of VMI are shown in table 5.3, and demonstrate that by taking timing

Table 5.2: Mann-Whitney U-stats for Xen and KVM vs the null hypothesis that the no VMI has been used

hypervisor	50 μs	100 μs	200 μs	1 ms	10 ms
Xen 10^6 Samples	$4.046 \cdot 10^9$	$3.960 \cdot 10^9$	$4.368 \cdot 10^9$	$4.020 \cdot 10^9$	$4.293 \cdot 10^9$
Xen 100 Samples	138.0	175.0	24.5	100.0	33.0
KVM 10^6 Samples	$3.086 \cdot 10^9$	$2.657 \cdot 10^9$	$2.422 \cdot 10^9$	$2.803 \cdot 10^9$	$2.589 \cdot 10^9$
KVM 100 Samples	100.0	81.0	176.0	55.5	73.0

Table 5.3: Information Gain Results for Xen and KVM in bits

hypervisor	50 μs	100 μs	200 μs	1 ms	10 ms
Xen	0.061	0.060	0.060	0.060	0.064
KVM	0.070	0.069	0.071	0.070	0.073

measurements, we gain as much as 0.08 bits of certainty about the VMI hypothesis.

This indicates that the number of samples required can be reduced significantly from the original 1,000,000. To test this hypothesis, we reduce the sample size to 100 samples which is a 10,000 fold decrease in sample size. We perform the t-test again to determine if this reduction in sample size will still give positive results. We can see from tables 5.1 and 5.2 that we are still able to determine the difference between the different samples. Again all p-values computed for these tests are less than 10^{-6}.

5.4 Support Vector Machines

5.4.1 Theory

After discovering that we need less than 100 samples in order to classify whether a sample has been monitored by a VMI agent we need a good machine learning classifier to test this on. For this classification we will use Support Vector Machines (SVM) introduced originally by Cortez and Vapnik [58] in 1995. SVMs begin by assuming data has the form

$$\{x_k, y_k\} \in \mathbb{R}^n \times \{-1, 1\} \tag{5.1}$$

where x_k is some data point and y_k is its classification (in our case monitored by VMI or not). We now wish to find a maximum margin hyperplane parameterized by $((\boldsymbol{w}), b)$. This hyperplane is selected such that it will separate the classes $y_i = 1$ and $y_i = -1$. Next we must choose a $((\boldsymbol{w}), b)$ such that we have the greatest distance between

$$\boldsymbol{w} \cdot \boldsymbol{x} - b = 1 \tag{5.2}$$

and

$$\boldsymbol{w} \cdot \boldsymbol{x} - b = -1 \tag{5.3}$$

Now we make the assumption that the data is linearly separable. While it may seem like this is severely limiting at first we can make this assumption using the "kernel trick" and map our features to some high-dimensional feature space. This will not be necessary in our case however given that our data is quite composed of a single feature repeated 100 times. By assuming our data is linearly separable we can select a $((\boldsymbol{w}), b)$ such that there are no points in between the planes denoted by equ. 5.2 and 5.3.

By examining fig 5.2 we can see that we want to minimize the distance $\frac{2}{\|w\|}$. In order to ensure that no points are in the margin we add the constraints

$$\boldsymbol{w} \cdot \boldsymbol{x}_i - b \geq 1 \tag{5.4}$$

and

$$\boldsymbol{w} \cdot \boldsymbol{x}_i - b \leq -1 \tag{5.5}$$

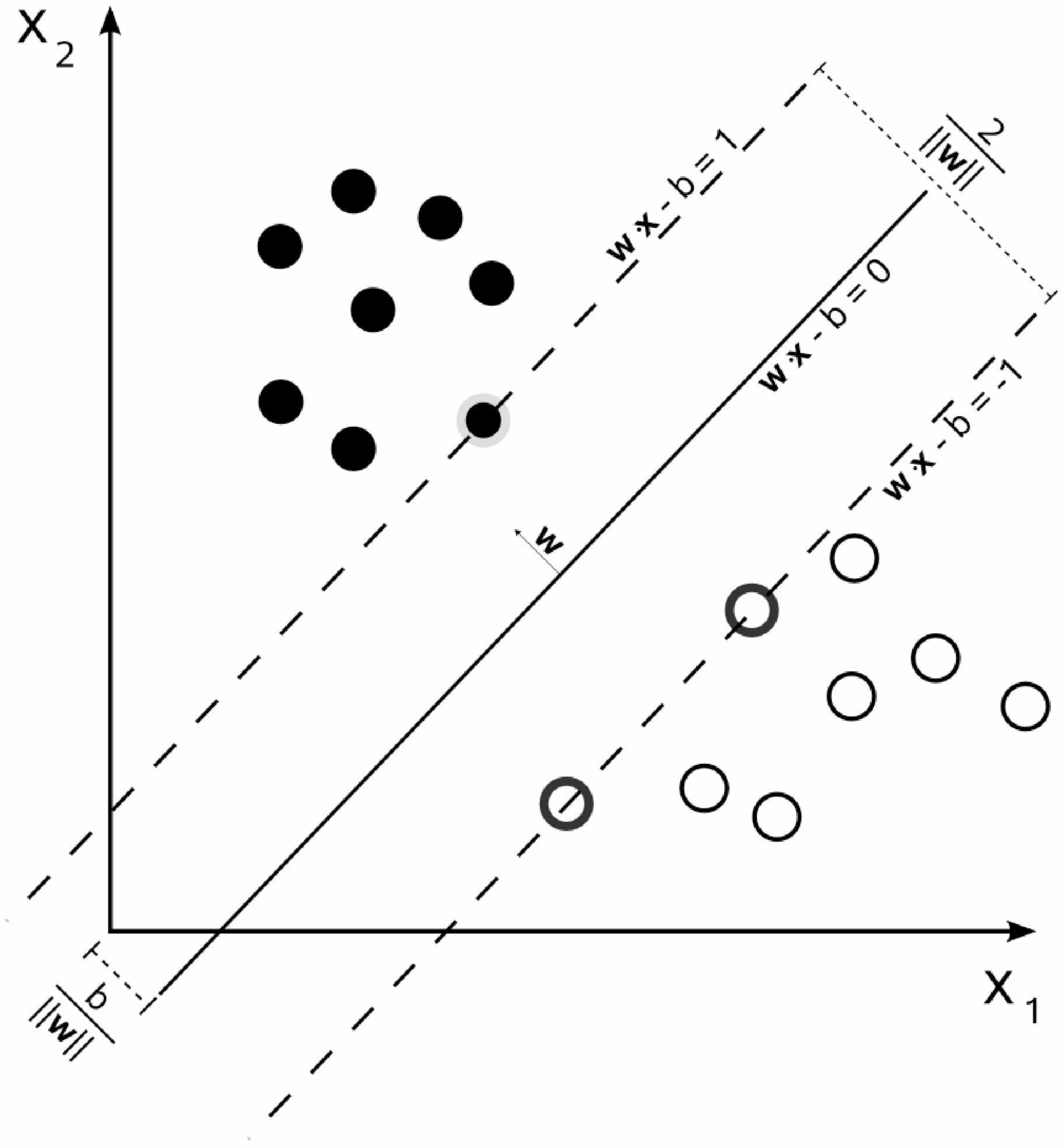
X_2
X_1
$\mathbf{w}\cdot\mathbf{x} - b = 1$
$\mathbf{w}\cdot\mathbf{x} - b = 0$
$\mathbf{w}\cdot\mathbf{x} - b = -1$
$\frac{2}{\|\mathbf{w}\|}$
$\mathbf{w}$
$\frac{b}{\|\mathbf{w}\|}$

Figure 5.2: A simple two dimensional example of an SVM

for $y_i = 1$ and $y_i = -1$ respectively. This can then be reduced to the form

$$y_k \times (\boldsymbol{w} \cdot \boldsymbol{x}_i - b) \geq 1 \tag{5.6}$$

for $1 \leq i \leq n$.

5.4.2 Experiment

Since we have determined that fewer than 100 samples is necessary to make a classification we begin by transforming our data into units of 100. Since we initially took 1,000,000 samples this is an easy transform, giving us a $10,000 \times 100$ data set. The first 100 sequential samples are transformed into one 100 dimensional data point, the next 100 are turned into the second data point and so forth. Each data point is labeled with a 1 for monitored by VMI and 0 for unmonitored by VMI. Since we know which samples were taken while being monitored by a VMI agent and which ones were not we can easily label our data.

We then split the data into a training set and a testing set using the *train_test_split* function in the *cross_validation* module provided by *Scikit-learn* [80]. Despite being in the cross validation namespace the *cross_validation.train_test_split* function does not perform any kind of cross validation. It instead generates a testing and training set randomly from the data. We take 60% of the data for the training set and the remaining 40% for the testing set. We then train our support vector machine using *Scikit-learn* and test it using the same. For our initial test we only tested whether

Table 5.4: Accuracy at classifying whether or not VMI was used at certain intervals for Xen and KVM

hypervisor	50 μs	100 μs	200 μs	1 ms	10 ms
Xen	0.9930	0.9905	0.9893	0.9940	0.9860
KVM	0.9858	0.9876	0.9870	0.9881	0.9881

our SVM could distinguish whether or not VMI was used on the target VM. We then fitted our SVM with our training data. Each point in the testing set was then classified using the SVM derived from our training data.

5.4.3 Results

For this experiment we tested our dataset which was unmonitored by VMI against the data sets which were taken when VMI was used at the intervals 50 μs, 100 μs, 200 μs, 1 ms, and 10 ms. The results of the accuracy are shown in table 5.4.

The accuracy is as high as 99.24%. In order to make sure this experiment does not suffer from the same failing as the experiment in chapter 2 we look at the false positive to determine if the accuracy is not skewed. We look at the results for Xen and KVM comparing a set of data taken when the guest was monitored every 100 μs to the set where the guest was unmonitored by VMI. We can see from fig 5.5 that the false positive rate is as low as 0.98% and 1.3% for Xen and KVM respectively. These results are representative of all the samples taken. This qualifies as

Table 5.5: The Raw Scores for Xen and KVM under the null hypothesis that the 50 μs set and set with no VMI are the same

hypervisor	*TP*	*FP*	*TN*	*FN*
Xen	3994	39	3930	37
KVM	3983	51	3918	48

a successful test for our purposes, however it is not perfect. As with all statistical methods it suffers from the base rate fallacy. That is while an overwhelming majority of points may be correctly classified, the sheer volume of data points which are involved would result in an incredibly large number of improperly classified data points.

5.5 Conclusion

In this chapter we were able to successfully demonstrate a technique to detect VMI used on a VM at set intervals. While this technique was successful boasting upwards of 99.24% accuracy it is not without its limitations. It cannot detect VMI with much success if it is used more than 10 ms apart. Further like all statistical methods it is subject to the base rate fallacy making its utility limited unless paired with another method which is not statistics based.

Chapter 6

Leveraging Kernel Samepage Merging to Detect VMI

6.1 Motivation and Introduction

Two side-channel attacks using same page merging were detailed by [52, 54]. Based on these examples we propose to leverage same page merging to construct a side-channel for the detection of the use of VMI on a target VM. In this experiment we leverage the memory de-duplication mechanism available in Linux called Kernel Same-Page Merging (KSM) [31]. KSM works by scanning the running Linux processes and marking those pages which produce identical hashes as candidates for merging. Those which are marked as candidates are then checked byte by byte to ensure they are indeed identical. The page table entries for the merged copies are all switched to point to only one of the previous pages (see fig 6.1). The other pages are then marked as being reclaimable by the OS. A copy-on-write scheme is employed in KSM much the same as it is in ESXi.

Since KVM is part of the Linux kernel [13] it leverages much of the existing kernel code to perform such tasks as scheduling and storage management. As a result VMs being run on a KVM hypervisor are subject to memory de-duplication with others VMs as well as processes running on the host. Since a VMI agent and a VM running on the same physical host are both treated as processes by the Linux kernel we hypothesize that the shared memory between the two can be de-duplicated. Further if these pages have been merged then they will be subject to

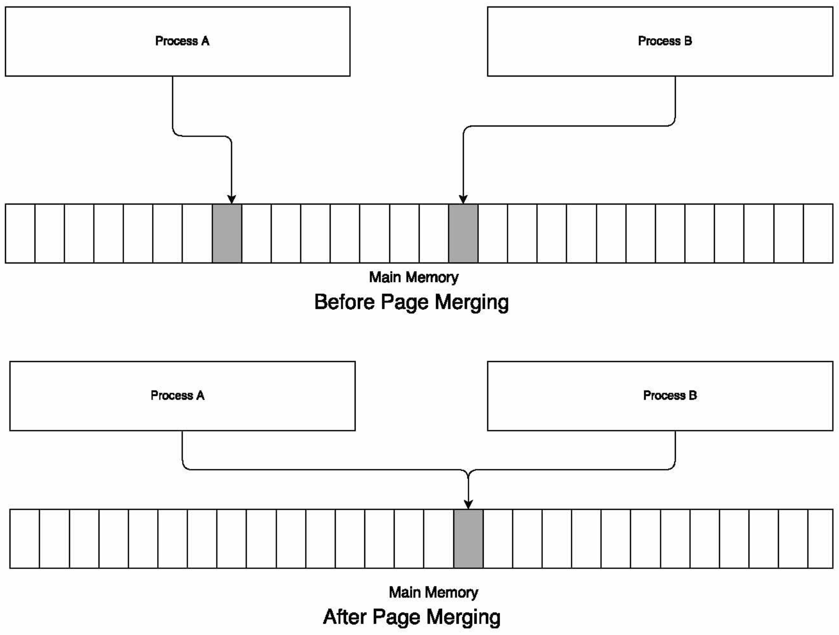

Figure 6.1: Kernel Samepage Merging

COW and therefore writing to these pages will be measurably slower than ordinarily writing to a page.

6.2 Experiment

For this experiment, which we call the Page Merging Experiment, we use the same apparatus as before. Only KVM is used for this experiment however. While Xen does employ a memory de-duplication technique it is only applicable to hardware virtualized guests. Since the Dom0 VM is necessarily paravirtualized [20] and the

VMI agent typically runs on the Dom0 this experiment is inappropriate for use with Xen.

As in chapter 3 this experiment is broken into two portions; the host portion and the guest portion. On the guest a monitoring process is run. This process allocates 10,000 pages in memory and again fills them with random values using the Mersenne Twister [83] to ensure that the values in the page are unique with a high degree of probability. The address of the data in memory is then printed as well as the process ID (PID). The data in the pages is printed to ensure that it is not optimized out by the compiler for having no output which is dependent on the data in those pages. While these prints are slower than the measurements being timed, they are not themselves timed and will not contribute any measurable overhead.

On the host side the memory for each of the pages in the monitoring process on the target VM is mapped one time by our VMI agent. These pages are again printed out to avoid the possibility of the compiler optimizing out the operations. The program then waits, keeping the memory mapped from the target VM in memory.

On the guest side a random number is added to a random element in each of the previously mapped and seeded pages. The process then pauses for 1 s between each write. The resulting memory access times are then printed and analyzed (see the following section).

Table 6.1: T-Tests for KVM vs the null hypothesis that the no VMI has been used

Number of VMs	VMI	Apache and No VMI	Apache with VMI
1VM	-44.15	-18.40	-4.742
2VMs	-29.62	-32.62	-28.71
3VMs	N/A	-8.368	N/A

6.3 Results

We begin our analysis by plotting both the timing data taken when the VM has not been monitored by VMI and data taken when the VM was monitored by our VMI agent (see fig 6.2). While fig 6.2 does not show a clear increase in the time taken to access a page, it does show that the two plots are slightly different. Are the two plots substantially different or do they merely look different? To answer this we again perform the t-test to check our data against the null hypothesis that both samples are the same. We see the results in table 6.1 and note that all p-values are less 10^{-6} . Based on these results we conclude that the two samples are different and we have thus detected the use of VMI on these pages.

As in chapter 4 we test other results to make sure that our signal is not easily reproducible. We repeat our experiment with two VMs running, three VMs running, and when apache is being run on the targeted VMs. The comparisons are shown in table 6.1.

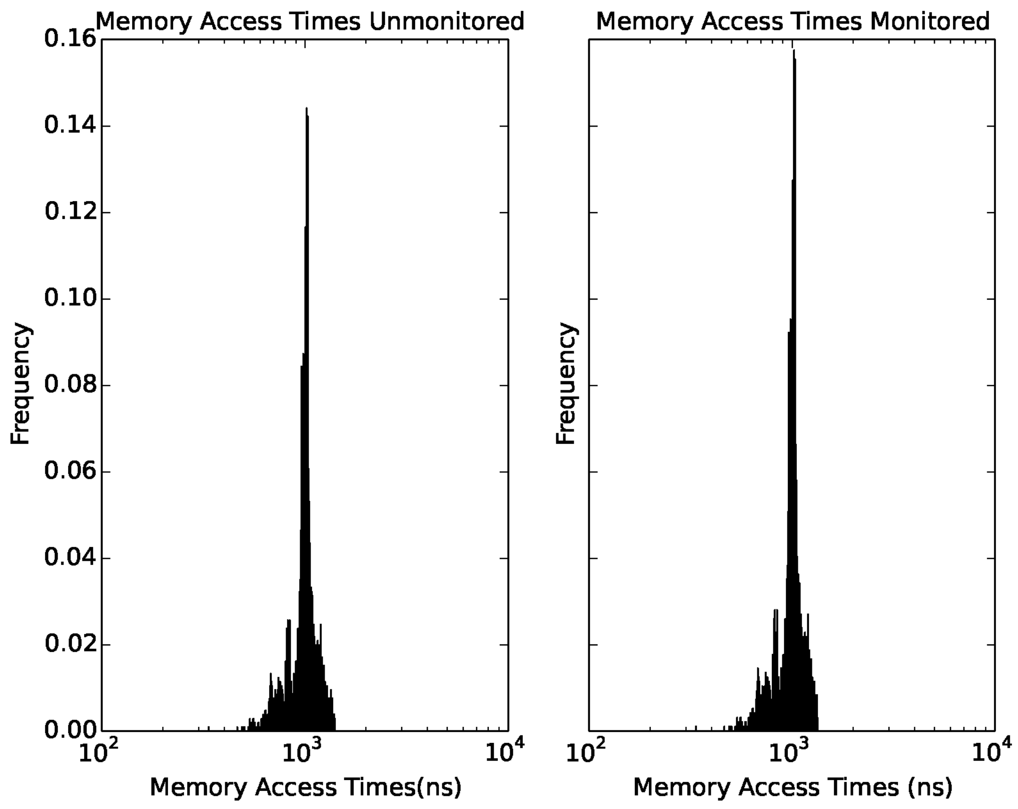

Figure 6.2: Histogram of the Memory Access Times for KVM not monitored by VMI (left) and monitored by VMI (right)

These results are discouraging in the case when multiple VMs are running. With t-statistics of -29.62, -32.62, and -28.71 it is difficult to distinguish which state the VM is in. We still reject H_0 we just do so more weakly when VMI and Apache are used. This makes this result somewhat suspect. When we run the t-test between the samples with apache and no VMI and the sample where VMI is run on the guest we get a t-stat of 4.960 with a p-value of $7 \cdot 10^{-7}$. This being below our critical value of 0.001 we reject the null hypothesis that those two are the same. While we can tell the case where Apache is running on a VM with and without VMI it is difficult for us to tell those two apart when comparing them to the sample that runs on one VM with no VMI.

This indicates that for this to be useful several tests must be performed rather than one simple t-test as in our previous results.

6.4 Conclusion

While we were able to detect the use of VMI on a specific page due to the KSM feature available in the kernel we had difficulties distinguishing the use of VMI versus the use of an Apache web server. This approach has both benefits and drawbacks. It allows us to determine whether or not a specific page has been accessed in memory which can be of use if a set page is being monitored regularly. The drawback however is that this specific page has to be accessed in order to be detectable by this scheme. In addition the fact that there can be a fairly substantial amount of time for the pages to merge can make this scheme somewhat difficult to add into a real time detection scheme.

Chapter 7

Conclusion and Future Work

In this dissertation we were able to detect the use of VMI on a single guest with a high degree of accuracy. We consider this a success. These successes are not without limitations however. The Sar method of detecting VMI did not work as it was too coarsely grained. The Page Merging and the MMap Method managed to detect VMI with a fair degree of accuracy, however they were limited by the amount of data needed to take or the time needed to make a successful measurement.

However we did have one method which was able to detect VMI with an extremely reliable accuracy. In addition it was able to determine individual points where VMI had been used, not just whether it had been used during the entire course of taking measurements. This method is however limited to systems where all the CPUs being discussed share the same L3 cache. This is not possible during cases where a machine has multiple sockets.

Our next experiment would be to scale this experiment to single socket CPUs which are connected on a cluster. Our hypothesis is that as long as all the members of the cluster are single socket machines, it will show the same results as our experiment on the cache timing.

We also wish to see if main memory or disk latency can be used to determine whether or not VMI has been used across CPU sockets.

What does this mean however, for the users? For the paranoid user (or sensible user) there are steps that can be taken to make one's computer far more secure. The first thing they would need to is generate a cache profile which reflects the user's computer use. This profile must be taken at a time when the user knows that they are not being monitored by VMI. This becomes problematic because the the system being analyzed must be virtualized for this to be a relevant comparison. If the user attempts to take their baseline measurements while their system is non-virtualized these measurements will be useless if their system becomes virtualized due to the difference in time required to make a timing measurement in a virtualized vs non-virtualized system.

Assuming that the user is able to create this idealized environment for this system to work, there are countermeasures which can be taken against VMI. For instance the DKSM attack [1] can shift around the location of kernel structures such that most VMI agents become unable to bridge the semantic gap. In addition some VMI agents are somewhat buggy. The VIX toolsuite [8] contains known bugs. The process-list implementation in VIX has a hard limitation of 300 processes. If a guest has more than 300 processes running a segmentation fault will occur in VIX. Armed with this knowledge a guest can defend themselves against some forms of introspection.

From the introspector's side there countermeasures which can be taken to prevent the detection and circumvention of VMI. Kernel integrity monitoring of VMs such as that in vShield [65] can prevent the user from implementing a DKSM style attack by only inspecting kernels which have not been altered. In addition VMI tools can be corrected and hardened such that they do not have trivially exploitable vulnerabilities.

Can the VMI agent get away without being detected however? Martin et al. [63] introduced a scheme where random values are added to the timestamps when rdtsc faults to the hypervisor. This will prevent most microarchitectural attacks, such as the ones presented in this paper, from being successful. However there is a tradeoff with their technique as not all applications will run correctly when the timestamps from the OS are altered. Since these modifcations require direct alteration of the hypervisor, this gives way to the possibility of detecting such alterations in the future which may give away the introspector in a different light.

While we have made advances in this dissertation we can see that the back and forth in security between attacker and defender is not going to be solved in this dissertation.

7.1 References

[1] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting virtual machine introspection for fun and profit," in *Reliable Distributed Systems, 2010 29th IEEE Symposium on*. IEEE, 2010, pp. 82–91.

[2] J. Pfoh, C. Schneider, and C. Eckert, "Exploiting the x86 architecture to derive virtual machine state information," in *Emerging Security Information Systems and Technologies (SECURWARE), 2010 Fourth International Conference on*. IEEE, 2010, pp. 166–175.

[3] B. Dolan-Gavitt, B. Payne, and W. Lee, "Leveraging forensic tools for virtual machine introspection," 2011.

[4] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 297–312.

[5] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process implanting: A new active introspection framework for virtualization," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 2011, pp. 147–156.

[6] Y. Fu and Z. Lin, "Bridging the semantic gap in virtual machine introspection via online kernel data redirection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 16, no. 2, p. 7, 2013.

[7] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection." in *NDSS*, 2003.

[8] K. N. B. Hay, "Forensics examination of volatile system data using virtual introspection," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 3, pp. 74–82, 2008.

[9] M. Bishop, *Computer security: art and science.* Addison-Wesley, 2012, vol. 200.

[10] "gmail," http://www.gmail.com, September 2015.

[11] *iAPX 286 Programmer's Reference Manual*, 1983, vol. 2014. [Online]. Available: http://bitsavers.trailing-edge.com/pdf/intel/80286/210498-001_1983_iAPX_286_Programmers_Reference_1983.pdf

[12] *Microsoft Windows*, 2014, vol. 2014. [Online]. Available: http://windows.microsoft.com/en-us/windows/home

[13] "The linux kernel archives," 2014. [Online]. Available: https://www.kernel.org/

[14] A. A. V. Codenamed, "Pacifica," *Technology: Secure Virtual Machine Architecture Reference Manual*, pp. 1–124, 2005.

[15] T. V. Vleck, *The IBM 360/67 and CP/CMS*, dec 2010, vol. 2014, no. March 17 2014.

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

[17] Intel, *Intel 64 and IA-32 Architectures Software Developer Manuals*, 2015. [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

[18] M. Rosenblum, "VMWare's virtual platform," in *Proceedings of Hot Chips*, 1999, pp. 185–196.

[19] O. Agesen, A. Garthwaite, J. Sheldon, and P. Subrahmanyam, "The evolution of an x86 virtual machine monitor," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 3–18, 2010.

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[21] *Understanding Full Virtualization, Paravirtualization, and Hardware Assist*, vol. 5-21-14. [Online]. Available: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[22] M. K. McKusick and G. V. Neville-Neil, *The design and implementation of the FreeBSD operating system.* Addison-Wesley Professional, 2004.

[23] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization." *Intel Technology Journal*, vol. 10, no. 3, 2006.

[24] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[25] J. Von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.

[26] A. Fog, "The microarchitecture of intel, amd and via cpus," *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2011.

[27] D. Kanter, "Inside nehalem: IntelâĂŹs future processor and system," April 2008. [Online]. Available: http://www.realworldtech.com/nehalem/8/

[28] "File:X86 Paging 64bit.svg Wikimedia Commons," apr 2009. [Online]. Available: http://upload.wikimedia.org/wikipedia/commons/9/9b/X86_Paging_64bit.svg?uselang=de

[29] AMD Corporation, "The AMD x86-64 architecture programmers overview," p. 63, January 2001, publication # 24108.

[30] C. S. Pabla, "Completely fair scheduler," *Linux Journal*, vol. 2009, no. 184, p. 4, 2009.

[31] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proceedings of the linux symposium*, 2009, pp. 19–28.

[32] B. Payne, *VMITools Library.*, mar 2014, no. March 28 2014.

[33] F. Zhao, Y. Jiang, G. Xiang, H. Jin, and W. Jiang, "Vrfps: A novel virtual machine-based real-time file protection system," in *Software Engineering Research, Management and Applications, 2009. SERA'09. 7th ACIS International Conference on*. IEEE, 2009, pp. 217–224.

[34] T. K. Lengyel, J. Neumann, S. Maresca, B. D. Payne, and A. Kiayias, "Virtual Machine Introspection in a Hybrid Honeypot Architecture." *CSET*, 2012.

[35] E. Weingartner, C. Terwelp, and K. Wehrle, "Promox: A protocol stack monitoring framework," *Electronic Communications of the EASST*, vol. 17, 2009.

[36] B. Marken and B. Hay, "Using memory map timings to discover information leakage to a live vm from the hypervisor," in *Services (SERVICES), 2014 IEEE World Congress on*. IEEE, 2014, pp. 48–52.

[37] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *ACM SIGPLAN Notices*, vol. 25. ACM, 1990, pp. 246–256.

[38] M. Kerrisk, *lsmod(8) - Linux manual page*, accessed: 5-21-2014. [Online]. Available: http://man7.org/linux/man-pages/man8/lsmod.8.html

[39] D. MacKenzie, *date(1) : print/set system date/time - Linux man page*, accessed: 5-21-2014. [Online]. Available: http://linux.die.net/man/1/date

[40] B. Lankester, *ps(1) - Linux manual page*, accessed: 9-17-2015. [Online]. Available: http://linux.die.net/man/1/ps

[41] M. Crawford and G. Peterson, "Insider threat detection using virtual machine introspection," in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE, 2013, pp. 1821–1830.

[42] J. Rushby, "Critical system properties: Survey and taxonomy," *Reliability Engineering & System Safety*, vol. 43, no. 2, pp. 189–219, 1994.

[43] C. Harrison, D. Cook, R. McGraw, and J. Hamilton, "Constructing a cloud-based IDS by merging VMI with FMA," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2012 IEEE 11th International Conference on*. IEEE, 2012, pp. 163–169.

[44] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM, 2010, pp. 206–213.

[45] O. Dain, R. Cunningham, and S. Boyer, "Irep++, a faster rule learning algorithm." in *SDM*. SIAM, 2004, pp. 138–146.

[46] B. Hay and K. Nance, "Circumventing cryptography in virtualized environments," in *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE, 2012, pp. 32–38.

[47] S. Yu, X. Gui, and J. Lin, "An approach with two-stage mode to detect cache-based side channel attacks," in *Information Networking (ICOIN), 2013 International Conference on*. IEEE, 2013, pp. 186–191.

[48] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[49] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[50] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: a metric for measuring information leakage," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3. IEEE Computer Society, 2012, pp. 106–117.

[51] Amazon, *AWS-Amazon Elastic Computing Cloud (EC2) - Scalable Cloud Hosting*, vol. 2014, no. 9/27. [Online]. Available: http://aws.amazon.com/ec2/

[52] R. Owens and W. Wang, "Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines," in *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International*. IEEE, 2011, pp. 1–8.

[53] C. Chaubal, "The architecture of vmware ESXi," *VMware White Paper*, 2008.

[54] J. Xiao, Z. Xu, H. Huang, and H. Wang, "Security implications of memory deduplication in a virtualized environment," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on.* IEEE, 2013, pp. 1–12.

[55] J. Butler and S. Sparks, *Windows Rootkits of 2005, part one*, 2005, vol. 2014. [Online]. Available: http://www.symantec.com/connect/articles/windows-rootkits-2005-part-one

[56] G. Hoglund, *A *REAL* NT Rootkit, Patching the NT Kernel*, 1999, vol. 9. [Online]. Available: http://phrack.org/issues/55/5.html

[57] O. Aciiçmez, "Yet another microarchitectural attack:: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture.* ACM, 2007, pp. 11–18.

[58] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[59] C. M. Bishop, *Pattern recognition and machine learning.* springer New York, 2006, vol. 1.

[60] A. Bates, B. Mood, J. Pletcher, H. Pruse, M. Valafar, and K. Butler, "Detecting co-residency with active traffic analysis techniques," in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, ser. CCSW '12. Raleigh, North Carolina, USA: ACM, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2381913.2381915

[61] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *Security and Privacy (SP), 2011 IEEE Symposium on*. IEEE, 2011, pp. 313–328.

[62] A. N. Pettitt and M. A. Stephens, "The Kolmogorov-Smirnov goodness-of-fit statistic with discrete and grouped data," *Technometrics*, vol. 19, no. 2, pp. 205–210, 1977.

[63] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking time-keeping and performance monitoring mechanisms to mitigate side-channel attacks," vol. 40, no. 3, pp. 118–129, 2012.

[64] A. More and S. Tapaswi, "Dynamic malware detection and recording using virtual machine introspection," in *Best Practices Meet (BPM), 2013 DSCI*. IEEE, 2013, pp. 1–6.

[65] VMWare, *VMware vShield Endpoint: Virtualization Security*, 2014, vol. 2014.

[66] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[67] "Linux cross reference: Linux/include/linux/sched.h," http://lxr.free-electrons.com/source/include/linux/sched.h, accessed: 2015-10-17.

[68] S. Godard, "Sysstat utilities home page," 2010.

[69] Anon, *Chrono - C++ Reference*, 2014, vol. 2014.

[70] J. Devore, *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2011.

[71] D. Wackerly, W. Mendenhall, and R. Scheaffer, *Mathematical statistics with applications*. Cengage Learning, 2007.

[72] B. L. Welch, "The generalization of student's problem when several different population variances are involved," *Biometrika*, pp. 28–35, 1947.

[73] F. E. Satterthwaite, "An approximate distribution of estimates of variance components," *Biometrics bulletin*, pp. 110–114, 1946.

[74] E. Jones, T. Oliphant, P. Peterson *et al.*, *SciPy: Open source scientific tools for Python*, 2001, [Online; accessed 2015-01-04]. [Online]. Available: http://www.scipy.org/

[75] T. Lumley, P. Diehr, S. Emerson, and L. Chen, "The importance of the normality assumption in large public health data sets," *Annual review of public health*, vol. 23, no. 1, pp. 151–169, 2002.

[76] T. M. Cover and J. A. Thomas, *Elements of information theory*. John Wiley & Sons, 2012.

[77] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[78] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, pp. 1482–1493, 1991.

[79] R. Marimont and M. Shapiro, "Nearest neighbour searches and the curse of dimensionality," *IMA Journal of Applied Mathematics*, vol. 24, no. 1, pp. 59–70, 1979.

[80] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[81] "mmap(2) - linux manual page," Jan. 2014. [Online]. Available: http://man7.org/linux/man-pages/man2/mmap.2.html

[82] R. Bryant and O. David Richard, *Computer systems: a programmer's perspective.* Prentice Hall, 2003.

[83] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[84] *Intel Virtualization Technology List*, 2014, vol. 2014.

[85] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach.* Elsevier, 2012.