SECURITY IMPROVEMENT IN CLOUD COMPUTING ENVIRONMENT THROUGH MEMORY ANALYSIS

by

Xiongwei Xie

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2017

Approved by:

_____
Dr. Weichao Wang

_____
Dr. Yu Wang

_____
Dr. Mohamed Shehab

_____
Dr. Bo Luo

_____
Dr. Chuang Wang

ABSTRACT

XIONGWEI XIE. Security improvement in cloud computing environment through memory analysis. (Under the direction of DR. WEICHAO WANG)

Cloud computing has attracted a lot of research efforts in recent years. More and more companies start to move their data and operations to cloud computing environment. However, the security issues in cloud computing environment have not been studied to a sufficient depth. For cloud computing end users, they do not have enough technology or methods to verify security Service Level Agreement violation. Furthermore, there are weaknesses while malware detection mechanism and malware are running in the same virtual machine. For cloud providers, if they improve the security of virtual machines at the hypervisor level, they could provide more secure cloud computing environment for the end users.

In this dissertation, we present a new understanding of security improvement in cloud computing environment. On both hypervisor and guest virtual machines, we propose mechanisms to detect malware or Service Level Agreement (SLA) violation in order to improve security in cloud computing environment. The key idea is to detect abnormal behaviors through memory analysis. We use techniques such as accumulated memory access latency, non-intrusive introspection of virtual machines, memory reconstruction, and cross-verification, to achieve this security improvement in cloud computing environment. In guest virtual machines, our mechanisms could detect unauthorized access to memory pages, violation of the memory deduplication policies, and under-allocation of memory to virtual machines. In hypervisor, we

can detect malware through cross-verification among different components of the reconstructed execution states of the virtual machine, and the operation system of a virtual machine could be Linux or Windows. We implement our approaches on Xen and VMware, and experiment results show that our detection mechanisms can effectively detect these abnormal behaviors with small increases in overhead.

ACKNOWLEDGMENTS

This dissertation would not be finished without the help, encouragement, and inspiration from a lot of people during this period. First and the most important, I want to thank my advisor, Dr. Weichao Wang for his help. I feel very fortunate to have the opportunity to be his Ph.D. student. Dr. Wang is always available to provide guidance, and suggestion not only in my research life, but also in my daily life. His support goes beyond his role as my Ph.D. advisor, and I will never forget the help he provided during these years.

I am so honored to have Dr. Yu Wang, Dr. Mohamed Shehab, Dr. Bo Luo, and Dr. Chuang Wang serve on my Ph.D. Committee. With their scrutiny, I improved this dissertation. I want to thank Dr. Yu Wang for teaching me network information. I want to thank Dr. Mohamed Shehab for the advice of giving a good presentation. I want to thank Dr. Bo Luo for his suggestion of my research. I want to thank Dr. Chuang Wang for the guidance of writing the dissertation.

I cannot complete my study at UNCC without the financial support of UNCC Graduate Assistant Support Plan, the NSF Collaborative Research, and Graduate Assistance. These programs not only provided financial support, but also led me to the right direction of cloud computing security and research.

Last, but not least, I would like to thank my family and friends, especially my parents, Jinhe and Shuzhen, and my wife, Mengfei, whose constant support gave me the encouragement to pursue my goals.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Motivation

Cloud computing is widely used by companies in today's world. It means that both applications delivered as services over the Internet and the hardware and system software in the servers which provide these services[8] are very popular. The trend is also to describe cloud computing as "Everything as a Service".

More and more companies start to move their data and operations to public or private clouds. For example, out of 572 business and technology executives that were surveyed in [15], 57% believed that cloud capability could improve business competitive and cost advantages, and 51% relied on cloud computing for business model innovation.

Due to these high demands, it is really important to develop mechanisms security of cloud computing. Although most companies use some security technology to hold sensitive information, security is one of the most often-cited objections to cloud computing[8]. Researchers have already proposed many methods to improve the security of cloud computing, which ranges from very theoretical efforts such as homomorphic encryption to very engineering mechanisms such as side channel attacks through memory and cache sharing.

In cloud computing, customers usually need to outsource their data processing

or storage to service providers. In order to guarantee the system performance and data safety, many customers rely on service level agreement (SLA) with providers to enforce these properties. Compared to the research in SLA enforcement for QoS parameters, research in security SLA validation falls behind in many aspects. End users of cloud computing environment may have some agreement with cloud providers on the usage of the memory of their virtual machines. They may ask the hypervisor to disable memory deduplication for their virtual machines, because this kind of page level memory sharing could create a side channel for information leakage[82, 100, 113]. However, end users have no solution to verify the execution of this agreement other than trusting the words of the cloud providers. As we know, cloud providers have the privilege to access the memory pages of the virtual machines through reconstructing memory [49, 65]. In order to protect their own sensitive data, end users could sign an agreement with providers that providers could not access their memory without their permission. However, if end users do not have the technology to detect such violation, it cannot be enforced. For cloud providers, they will use memory overcommitment technique in hypervisors to use memory resource efficiently [11, 33]. For end users, the performance could be severely impacted, if providers apply memory overcommitment technique [41]. There is also no mechanisms for end users to detect such violation, even if they have agreement on minimum physical memory with providers.

Malware, which gains access to private computer, gathers sensitive information, or displays unwanted advertising, is high spread through Internet, and infect unnumbered computer systems. In Symantec's internet security threat report [103], the number of new malware variants detected by them is 431 million in 2015, which is

a 36% increase from 2014. Furthermore, over half a billion personal records were stolen in 2015 [103]. Some malware are difficult to be detected since they could try to hide its existence from anti-malware programs, and some of them can detect and remove anti-malware programs in the victim machine [129]. The emergence of cloud computing opens a new horizon for solving this problem. In cloud computing environment, through memory reconstruction, and non-intrusive introspection of virtual machines, we could have a semantic view of virtual machines at the hypervisor level. The hypervisor can monitor the behaviors of virtual machines. It is really difficult for malware running in the virtual machine to hide malicious behaviors from the hypervisor. Existing hypervisor-based malware detection mechanisms use information from different modules of a virtual machine as individual components to conduct malware detection [49].

1.2 Contribution

This dissertation explores several approaches for improving security in cloud computing environment through memory analysis. The main contributions of this dissertation are the following. At guest virtual machine level, we propose mechanisms to enforce security service level agreement (SLA). We also implement the approaches to detect memory management SLA violation in virtual machine, so that customers can verify memory overcommitment policy, memory deduplication policy, and unauthorized memory access policy. At hypervisor level, we first use the reconstructed information from different modules of the virtual machine as an integrated, cohesive system for rootkit detection, which is based on cross-verification. Secondly, we pro-

pose a malware detection mechanism through lightweight examination of Dynamic Link Library (DLL) environments in virtual machines. Thirdly, we also present a system call examination approach at hypervisor level to continuously monitor the sequence of the system call in guest virtual machines to detect malware. They incur very limited overhead in the virtual machines, because these detection mechanisms use no-intrusive introspection of virtual machines. They are running in hypervisor so that malware running in a virtual machine could not easily escape our detection.

1.3 Outline

The remainder of this dissertation begins with Chapter 2, which introduces our detection mechanism of service level agreement (SLA) violation in guest virtual machine. Cloud computing users could use this detection mechanism to verify SLA violations to memory management in the virtual machine. Chapter 3 presents a rootkit detection on guest virtual machine through deep information extraction at the hypervisor level, while guest virtual machine's Operation System is Linux. Chapter 4 presents a lightweight examination of Dynamic Link Library (DLL) environments in virtual machines to detect malware. Chapter 5 presents a system call examination approach at hypervisor level to continuously monitor the sequence of the system call in guest virtual machines to detect malware. Since we use non-intrusive introspection of virtual machines, it is very difficult for malware running in a virtual machine to detect, remove, or avoid our detection. Chapter 6 concludes this dissertation.

CHAPTER 2: DETECTION OF SERVICE LEVEL AGREEMENT VIOLATION IN GUEST VIRTUAL MACHINE

2.1 Introduction

In the past few years, cloud computing has attracted a lot of research efforts. At the same time, more and more companies start to move their data and operations to public or private clouds. For example, out of 572 business and technology executives that were surveyed in [15], 57% believed that cloud capability could improve business competitive and cost advantages, and 51% relied on cloud computing for business model innovation. These demands also become a driving force for the development of cloud security, which ranges from very theoretical efforts such as homomorphic encryption to very engineering mechanisms such as side channel attacks through memory and cache sharing.

In parallel to the active research in cloud security, enforcement of service level agreement (SLA) also becomes a very hot topic. In cloud computing, customers usually need to outsource their data processing or storage to service providers. To guarantee the system performance and data safety, many customers rely on service level agreement (SLA) with the providers to enforce such properties. The resources that are monitored under SLA include CPU time [19, 29, 46], network downtime [28], and bandwidth [19]. Several multi-layer monitoring structures [28, 23, 30, 39] have been proposed to link low-level resources with high-level SLA requirements.

Compared to the research in SLA enforcement for QoS parameters, investigation in security SLA validation falls behind in many aspects. For example, in [38] the authors define the concept of an accountable cloud and propose an approach to differentiate the responsibility of a user from that of the service provider when some security breach happens. An infrastructure to enforce security SLA is described in [16]. However, the high-level discussion often lacks implementation details. It is very hard to generate concrete defense mechanisms for security SLA violations based on these descriptions.

To bridge this gap, in this chapter, we study mechanisms to detect violations of security SLA for memory management in virtual machines. Under many conditions, end users of a cloud environment may sign some agreement with the cloud provider on the usage and monitoring of the memory of their virtual machines. For example, many prominent virtual machine hypervisors such as VMware ESX and ESXi [106], Extended Xen [37], and KSM (Kernel Samepage Merging) [7] of the Linux kernel use the technique of memory deduplication to reduce memory footprint size of virtual machines. Since previous research has shown that page level memory sharing could create a side channel for information leakage [82, 100, 113, 101], many end users ask the hypervisor to disable memory deduplication for their virtual machines. However, there exists no solution for end users to verify the execution of this agreement other than trusting the words of the cloud provider.

As another example, cloud providers usually have the privilege to take a sneak peek at the memory pages of the virtual machines under their management and reconstruct their internal views [49, 65]. To protect their own privacy, end users could sign an SLA with the provider that prevents it from peeking at the memory pages without

their permission. However, if no technical mechanism can detect violations of such an SLA, it cannot be enforced.

Last but not least, because of the sharing property of cloud computing, memory overcommitment is a widely used technique by hypervisors [11, 33, 36, 2]. During the initiation of a virtual machine, a user can request the size of RAM and also identify the minimum physical memory that the hypervisor needs to guarantee. However, if no violation detection mechanism is implemented, a greedy hypervisor may cut the allocated memory of the virtual machine and allocate it to another virtual machine. Previous research [41] has shown that when a virtual machine gets too little physical memory, its performance can be severely impacted since the CPU will either be busy handling page swapping or use most of the time waiting for data to be loaded into the system.

To detect such violations, we propose to design mechanisms based on memory access latency. When we revisit the three types of violations described above, we find out that all attacks would lead to changes in access orders to the memory pages. For the attack on memory deduplication, although the other virtual machine is accessing only its own pages, the victim virtual machine is impacted because of the shared memory. For the attack of unauthorized peek, the attacker violates the security SLA and reads the memory of the victim virtual machine. For the memory under-allocation violations, the reduced physical memory resources to a virtual machine will lead to extra swapping. Under all cases, the order of accesses to the virtual machine's memory pages changes. Our detection mechanisms will try to capture such changes.

According to the documents released by major hypervisor companies such as VMware

and research results of other investigators [63, 107], Least Recently Used (LRU) memory pages are still the top choices during memory reclaiming. Therefore, unauthorized accesses to the memory pages of the victim virtual machine will lower their priority of being swapped out. We propose to introduce a group of reference pages into the virtual machine memory and access them with different time intervals. In this way, we can set up a series of reference points in time for memory swapping operations. Through comparing the access latency to these reference pages with that to the pages we try to monitor, we can determine which pages are still in the memory and which pages have been swapped out. Since the reference pages are hidden within the real data and program pages, it is very hard for the attacker to identify them and treat them differently. We have implemented the approaches in virtual machines under VMware and tested them. Our experiment results show that the approaches can effectively detect the violations to SLAs in memory management with small increases in overhead.

The contributions of the chapter are as follows. First, existing SLA enforcement mechanisms usually focus on hardware resources such as CPU cycles and network bandwidth. Our approaches study this problem from a different aspect and try to enforce security SLAs. Second, in this research, we choose SLA violations to memory management in virtual machines as an example. We design mechanisms to detect such attacks based on changes in memory access latency. Last but not least, we implement the approaches and evaluate them in real systems. The experiment results show the effectiveness of the approaches.

The remainder of the chapter is organized as follows. In Section 2.2 we introduce

the related work. Specifically, we focus on the enforcement of SLAs and information leakage through changes in memory and cache access latency. In Section 2.3 we present the details of our approaches. In Section 2.4 we present the experimental results in real systems. In Section 2.5 we discuss the safety and efficiency of the approaches. Finally, Section 2.6 concludes the chapter.

2.2 Related Works

2.2.1 Information Leakage among Virtual Machines

With the proliferation of cloud computing, more and more companies start to use it. One big security concern in the cloud is the co-residence of multiple virtual machines belonging to different owners in the same physical box. Existing investigation on this problem can be classified into two groups. In the first group, side channel attacks through shared cache have been carefully studied. The shared cache enables timing based attacks [88, 35, 67, 105] to steal information about key stroke or Internet surfing histories. Research in [119] shows that in a practical environment such as Amazon EC2, a cache-based covert channel can reach the effective bandwidth of tens of bits per second. Some implementations of the key extraction attack were presented in [128, 45]. More recently, researchers have shown that even when multiple virtual machines are put on different cores of a CPU, cache level information stealing is still a viable attack [47]. The delay caused by separation of deduplicated memory pages in virtual machines has been used to identify guest OS types [82] or derive out memory page contents [81].

In the second group, researchers have designed security mechanisms to prevent

information leakage among the virtual machines. In hardware-based approaches, special components have been embedded into the architecture to manage information flow. For example, in [50] the processor is responsible for updating the memory page mapping and the page table. Szefer *et al.* [104] proposed to use memory level isolation or encryption to protect guest virtual machines from a compromised hypervisor. The software-based approaches adopt more diverse mechanisms. For example, in [98] the cache that is used by security related processes is labeled with different colors to prevent side channel attacks. In both [64] and [75], the hypervisor monitors the memory access procedures to prevent cross-VM information flow. In [86], researchers tried to establish a very small compartment to allow virtual machines to run in an isolated state. Using lightweight run-time introspection, Baig *et al.* identify side channels which could potentially be used to violate a security policy, and reactively migrate virtual machines to eliminate node-level side-channels [10].

2.2.2 Service Level Agreement Enforcement

In cloud computing environments, service level agreements are often described at the business level. Their enforcement, however, is often at the technical level. Such discrepancy creates a challenge for researchers. Two groups of approaches have been designed to solve this problem. In group one, a middleware layer is implemented to bridge the high-level service requirements with low-level hardware resources [19, 29, 28]. Group two push the approaches one step further through formalization of both service capabilities and business process requirements [23]. In this way, a language can be used for direct communication between the two layers. Dastjerdi *et al.* [24]

designed an ontology-based approach that can capture not only changes in individual resources but also their dependency.

Compared to research in SLA enforcement for QoS parameters, investigation in security SLA validation deserves more efforts. As a pioneer, Henning [40] raised the question of whether or not security can be adequately expressed in an SLA. Casola *et al.* [20] proposed a methodology to evaluate and compare security SLAs in web services. Chaves *et al.* [25] explored security management through SLAs in cloud computing. An infrastructure to enforce security SLA in cloud services is described in [16]. Haeberlen [38] proposed an approach to differentiate the responsibility of a user from that of the service provider when some security breach happens. Efforts have also been made to quantify the security properties in cloud computing environments so that automated and continuous certification of security properties could be implemented [53, 85].

2.3 The Proposed Approach

In this section, we will present the details of the proposed approaches. We first introduce the techniques of memory overcommitment and deduplication and how they impact memory access in virtualization environments. We will then discuss the assumptions of the environments to which our approaches can be applied. Finally, the details of the approaches and mechanisms to turn the idea into practical solutions are presented.

2.3.1 Memory Overcommitment in Virtualization Environments

Virtualization enables service providers to consolidate virtual hardware on less physical resources. The consolidation ratio is an important measure of the virtualization efficiency. To boost up system performance, overcommitment has been adopted to enable the allocation of more virtual resources than available physical resources. For example, two virtual machines with 4GB RAM each could be powered on in a VMware ESXi server with only 4GB physical memory. To support smooth operations of the two virtual machines, various techniques such as page sharing, memory ballooning, data compression, and hot swapping have been designed [11]. Almost all prominent hypervisors use some type of memory overcommitment [73]. For example, both DiffEngine [37] and Singleton [97] have tried to use page deduplication to reduce the memory footprint of virtual machines. Ginkgo [42] implements a hypervisor-independent overcommitment framework to adjust CPU and memory resource allocations in virtual machines jointly.

Among different techniques for memory overcommitment, memory ballooning is an active method for reclaiming idle memory from virtual machines. If a virtual machine has consumed some memory pages but is not subsequently using them in an active manner, VMware ESXi attempts to reclaim them from the virtual machine using ballooning. Here an OS-specific balloon driver inside the virtual machine will first request memory from the OS kernel. Once granted, it will transfer the control of the memory to ESXi, which is then free to re-allocate it to another virtual machine. The whole procedure includes both inflation (getting idle memory from the first virtual

machine) and deflation (giving memory to the second virtual machine) of the balloon. The balloon driver is among the utilities installed in the guest operating system with VMware Tools. It effectively utilizes the memory management policy of the guest OS to reclaim idle memory pages. An example is shown in Figure 1.

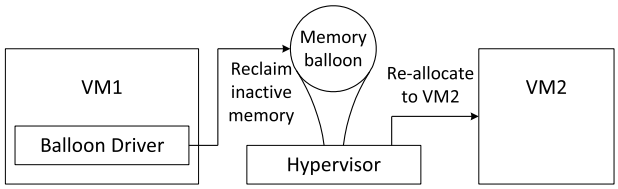


Figure 1: One technique of overcommitment: memory ballooning.

Although memory overcommitment can improve the consolidation ratio in virtualization, it also has the potential to severely impact system performance. For example, the set of memory pages that are being actively accessed by a virtual machine is called the working set. Previous research [11] shows that when the total working set of virtual machines remains within the physical memory limit, the virtual machines will not experience significant performance loss. On the contrary, when the overcommitment factor becomes larger than 2, the Operations per Minute (OPM) of a virtual machine can reduce 17% to 200%, depending on the properties of the applications.

Because of the potential impacts, many end users choose to identify the minimum size of physical memory that a cloud provider needs to guarantee when they initiate their virtual machines. In this chapter, we will discuss mechanisms to verify the allocated memory resources.

2.3.2 Memory Deduplication in VM Hypervisors

The memory de-duplication technique takes advantage of the similarity among memory pages so that only a single copy and multiple handles need to be preserved

in the physical memory, as shown in Figure 2. Here each of the two virtual machines *VM1* and *VM2* needs to use three memory pages. Under the normal condition, six physical pages will be occupied by the virtual machines. If memory de-duplication is enabled, we need to store only one copy of multiple identical pages. Therefore, the two virtual machines can be fit into four physical pages (note that we illustrate both inter- and intra-VM memory de-duplication in the figure). This technique can reduce the memory footprint size of virtual machines and the performance penalty caused by memory access miss.

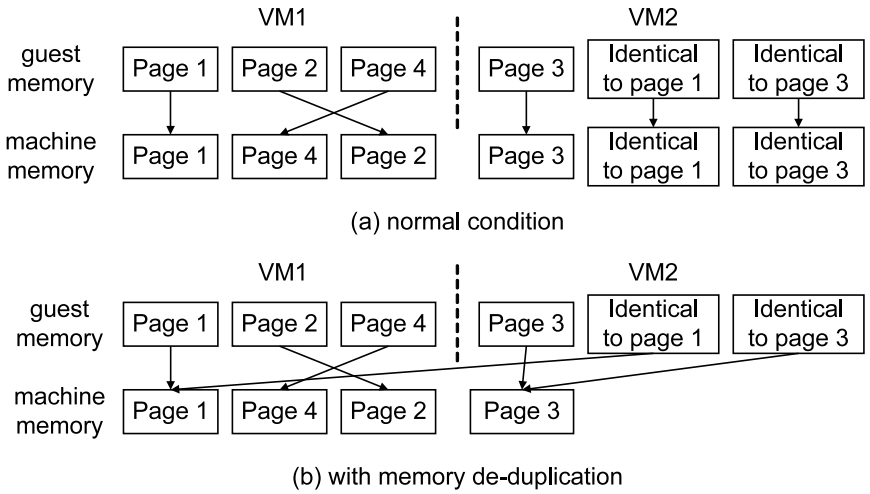


Figure 2: Memory de-duplication reduces the memory footprint size.

Although the implementation of memory deduplication in different hypervisors may be different, the basic idea is similar. To avoid unnecessary delay during page loading, whenever a new memory page is read from the hard disk, the hypervisor will allocate a new physical page for it. Later, the hypervisor will use idle CPU cycles to locate the identical memory pages in physical RAM, and remove duplicates by leaving pointers for each virtual machine to access the same memory block. Hash results of the memory page contents are used as index values to locate identical pages. To avoid

false de-duplication caused by hash collisions, a byte-by-byte comparison between the pages will be conducted. While the reading operations to the de-duplicated pages will access the same copy, copy-on-write is used to prevent one virtual machine from changing another virtual machine's memory pages. Specifically, on writing operations a new page will first be allocated and copied. This procedure will incur extra overhead compared to writing to not-shared pages, which will lead to a measurable delay when a large number of shared pages are allocated and copied. This delay will allow us to detect violations of security SLAs on memory management.

Newer operating systems include a technique known as Address Space Layout Randomization (ASLR). In this technique, operating systems try to prevent code injections from being successful by changing the memory locations of executables. For example, in Windows Vista, the memory is randomized in the whole blocks of 64 KB or 256 KB [102]. Since the memory pages have the size of 4KB, this technique will not impact the results of memory deduplication.

2.3.3 System Assumptions

In the investigated scenarios, we assume that the security SLA signed between the cloud provider and customers includes the following three requirements: (1) the provider cannot apply memory deduplication technique to the memory pages of the guest virtual machines; (2) without a customer's permission, the provider cannot peek at the memory pages of her virtual machine, and (3) the provider needs to guarantee the minimum size of allocated physical memory to the virtual machine. We assume that a customer has total control over the memory usage of her virtual machine. For

example, she can initiate application programs and load data files into the memory of the virtual machine. She can also measure time durations accurately so that they can be used for attack detection (more discussion in subsequent sections). We do not assume the customer can decide how much physical RAM her virtual machine can consume. Without losing generality, we assume that virtual machines use $4KB$ memory pages.

We assume that the cloud service provider is not malicious but very curious. At the same time, it wants to squeeze as many virtual machines as possible into a single physical box to maximize its profits. From this point of view, it has the motivation to enable memory deduplication or allocate less memory for virtual machines. The cloud service provider may not be the developer of the hypervisor software. For example, a private cloud provider runs VMware software to manage the cloud. It does not have the capability to alter the source code of the hypervisor. However, it may configure the software to enable memory deduplication. It may also read the memory page contents of other virtual machines through the virtual machine manager. All such operations can be conducted without the permission of end users.

2.3.4 Basic Ideas of the Proposed Approaches

In this part, we will first introduce the basic ideas of the approaches. The difficulties to turn these ideas into practical mechanisms and our solutions to these problems will then be discussed.

The basic idea of the proposed detection mechanisms is to map violations of security SLAs on memory management to changes in memory access latency. If a memory

page is located in physical memory, its access delay is at the level of several to tens of micro-seconds. On the contrary, if a memory page has been swapped out, its access delay is partially determined by the hard disc performance. The waiting time is usually at the level of milliseconds. From this point of view, we can easily differentiate a page in physical memory from that on a hard disk.

Now let us reexamine the three violations of interest. When the hypervisor or an attacker takes a sneak peek at memory pages of a guest virtual machine, it will change the sequence of access operations, thus impacting the priority of page swapping. We can choose a group of memory pages to serve as references and keep records of access operations to them. If we detect that the order of memory swapping is different from that of the memory access commands initiated by the guest virtual machine, we can confirm that some unauthorized access has happened.

A similar technique can be applied to detect the violation of memory deduplication policies. A user can load two files (we call them *F1* and *F2*) with the same contents into her virtual machine memory. If the hypervisor does not enable memory deduplication, the files will occupy different chunks of memory. Otherwise, their memory pages will be merged. To differentiate between these two cases, we need to conduct the following operations. We will access the pages of *F1* and *F2* regularly to keep them in the main memory. We can estimate the progress of deduplication based on our previous research [82]. When this procedure is finished, we can initiate "writing" operations to the pages. If the memory pages of the two files are not merged, the writing delay will be relatively short. On the contrary, if their pages are deduplicated, "copy-on-write" must be conducted for every page. A measurable increase in delay

can be detected. Based on the measurement results, we can figure out whether or not the SLA for memory deduplication has been violated.

Determination of the allocated physical memory for a virtual machine deserves more discussion. Here we consider two scenarios. In the first scenario, the working set of a virtual machine is larger than the allocated physical memory. Under this condition, the virtual machine performance will deteriorate because of the increases in page misses. Previous research [42] has shown that through a continuous sampling of application metrics under a variety of memory configurations and loads, we can derive out a performance-to-memory correlation model. In this way, a virtual machine can use the size of its active working set and measured system performance to estimate the allocated physical memory. It can then compare the estimation result to the promised memory by the SLA to detect possible violations.

In the second scenario, the size of the working set is smaller than the allocated physical memory. Under this condition, the virtual machine's performance is not restricted by the physical memory size. If the virtual machine determines that its working set is already larger than the physical memory size promised by the SLA, no further detection needs to be conducted. On the contrary, if the working set is smaller than the promised memory size, we can request new memory allocation to boost the usage to the promised value. We can then locate the pages from the working set that have the least recent access records and read them. If the hypervisor has allocated less than SLA promised memory to this virtual machine, these pages would have been swapped out. Therefore, through measuring access latency to these memory pages, we can derive out whether or not there exists a violation in memory allocation for

the virtual machine.

2.3.5 Details of Implementation

Although the basic ideas of the detection mechanisms are straightforward, we face several difficulties in implementation. For example, we need to carefully select the memory pages that we access to reduce false alarm rates. We also need to consider the time measurement accuracy and the order of memory page accesses. Below we discuss our solutions to these problems.

2.3.5.1 Choice of Memory Pages

The first problem that we need to solve is to choose the memory pages that will be used for the detection of SLA violations. There are several criteria that we need to follow when we choose these pages. First, the selected pages must incur a very small performance penalty on the system. If the proposed approaches impact the system efficiency to a large extent, it will become extremely hard to promote their wide adoption. Second, these pages should not be easily identified by the hypervisor or attackers. Otherwise, they may handle these pages differently to avoid detection.

We design different methods to choose memory pages for the detection of the investigated violations. The selection of memory pages for the detection of unauthorized access is very tricky. Theoretically, attackers or the hypervisor could choose any memory pages of the guest virtual machine to read. It is impossible for the guest virtual machine to know beforehand which pages to examine. In the real world, however, the selection range is much smaller. An end user usually cares most about the data files that she/he is processing with sensitive information. Therefore, we propose to insert

guardian pages into these sensitive data files. Since these pages are integrated into the real data files, the hypervisor or attackers will not be able to identify them. Therefore, when they conduct unauthorized memory access to the guest virtual machine, these pages have a high probability to be touched.

To detect whether or not the cloud provider secretly enables memory deduplication without notifying end users, we need to make sure that the following two requirements are satisfied: (1) there exist memory pages that can be merged; and (2) more importantly, the guest virtual machine can read from/write to these pages to measure the access delay. We propose to construct data files and actively load them into our virtual machine's memory. Since deduplication is conducted at the page level, the offsets of the pages in data files will not impact the final result. Therefore, we can construct different data files through reorganizing the order of the pages. This scheme will also introduce randomness into the data files so that it is difficult for the cloud provider to discover the detection activities. After constructing these files, we can initiate different application software to load them into memory. Since memory deduplication can happen in both intra-VM and inter-VM modes, we can read different files in different virtual machines. Under this case, we can use methods in [88] to make sure that these virtual machines are located in the same physical box so that deduplication can be conducted.

To detect under-allocation of physical memory to a virtual machine, we need to identify the pages with the least recent access records. Here a malicious hypervisor also knows these pages. However, there is very little that it can do to hide the fact that there is not enough memory allocation for the virtual machine. For example, the

SLA may have promised that at least 1GB physical memory will be guaranteed for the victim virtual machine while in real life only 800MB is provided. Under this case, when our detection mechanism boosts the memory usage to 1GB and the hypervisor refuses to provide new memory, at least 200MB data needs to be swapped out. Since the victim virtual machine can choose any memory pages in the working set as the detection sensors and measure access delay to them, it is very hard for the hypervisor to hide the increased loading latency when we consider the time difference between reading from memory and reading from hard disk.

2.3.5.2 Measurement of Access Time

To successfully detect the SLA violations, we must accurately measure the data access time. Traditionally a computer provides three schemes to measure the length of a time duration: time of the day, CPU cycle counter, and tickless timekeeping. The first method provides the measurement granularity of seconds which is too coarse for our application. The second method will be a good candidate for time measurement if the operating system completely owns the hardware platform. In a VM-based system, however, it cannot accurately measure the time duration. For example, if a page fault happens during our reading operation, the hypervisor may pause the CPU cycle counter while it fetches the memory page. Therefore, the delay caused by hard disk reading will not be measured. Based on the analysis in [108], tickless timekeeping can keep time at a finer granularity. Therefore, we choose the Windows API $QueryPerformanceCounter$ to measure the duration. Previous research [59] has also shown that the time measurement accuracy may be impacted by the workload on

the physical box. We can use the lightweight toolset $TiMeAcE.KOM$ [59] to assess and fix the measurement results.

2.3.5.3 Verification of Memory Access Order

As explained in Section 2.3.D, the detection of SLA violations in memory management depends on the verification of some facts: some memory pages that should have been swapped out are still in memory, while some other pages that should have stayed in memory are swapped out. The reason that these discrepancies happen is because some access operations change the order of swapping. To verify this fact, we need to set up a group of memory pages to serve as references in time. Through controlling access to these reference pages, we can derive out whether or not the data pages should have been swapped out. While the basic idea is straightforward, we need to consider several issues when we choose these reference pages. First, the reference pages should not belong to frequently used OS or application software. In this way, they will not be merged by the deduplication algorithm. Second, we want these reference pages to be randomly distributed in the memory. In this way, if the cloud provider or attackers access the guest virtual machine memory stealthily, they have a very low probability to read many reference pages.

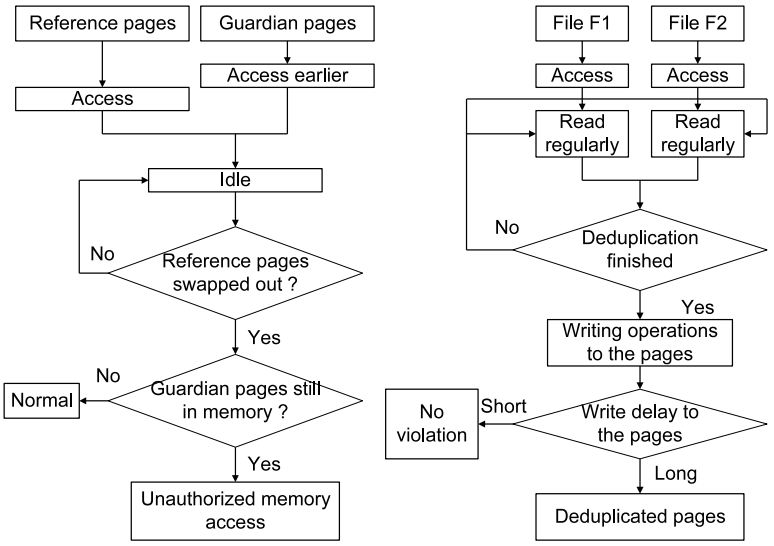
To satisfy these requirements, we propose to use the memory pages that are unique in the Windows 95 system as the reference pages. Our previous research [82] has successfully identified these pages. Since almost no users are still using Windows 95, these pages will not be deduplicated. We will allocate space for each individual page and chain them together with pointers to form a linked list. Since we do not allocate

a big chunk of continuous memory for these pages, they may distribute all over the memory. Therefore, it is very hard for the attacker to touch many reference pages if he randomly selects guest virtual machine pages to read. In this way, the reference pages can effectively serve their purposes.

2.3.6 Detection Procedures of the SLA Violations

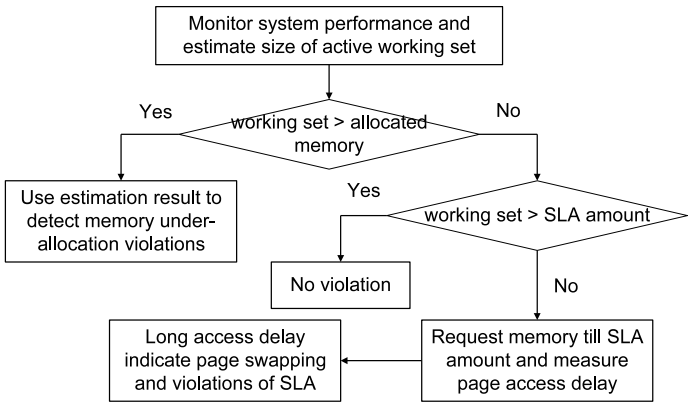
With all the building blocks we need to construct the detection algorithms, below we describe the details of the detection procedures. We introduce the algorithms respectively for the three SLA violations.

Figure 3.a illustrates the detection of unauthorized memory accesses. The guest virtual machine will load both the reference pages and guardian pages into its memory. Since the swapping operations heavily depend on the memory usage, we propose to divide the reference pages into multiple groups and access them at different intervals. As illustrated in Figure 4, we will first read all the guardian pages before the reference pages. In this way, the guardian pages would have been swapped out before the reference pages if no one else touches them afterward. These pages will then be left idle. We will access the reference pages with different intervals. For example, the intervals shown in Figure 4 for different groups of reference pages increase exponentially. Assuming that at time $7t$ the virtual machine is under pressure for more memory and has to swap many pages out. Since the guardian pages are accessed before the 4th group of reference pages, they will be swapped out first. Then the 4th group of reference pages are also swapped out. At time $8t$ when the predetermined interval for group 4 expires, we will read this group of reference pages. Since they



(a) Detection of unauthorized memory access.

(b) Detection of memory deduplication.



(c) Detection of memory under-allocation violations.

Figure 3: Detection procedures of the SLA violations.

have been swapped out, the reading delay will be long. As soon as we detect the long reading delay, we can derive out that these pages are no longer in memory. We will then immediately conduct reading operations on the guardian pages. If the reading delay is short, we can derive out that these pages are still in memory. Therefore, some unauthorized access to these pages must have happened after our first reading.

Figure 3.b illustrates detection of the violations of deduplication policies. Here two applications in the guest virtual machine will read the files $F1$ and $F2$ into its memory,

respectively. The two files contain identical memory pages. Should deduplication is enabled in the guest virtual machine, the pages of the two files will be merged. We will read $F1$ and $F2$ regularly so that their pages will not be swapped out. Using our previous research in [82], we will estimate the time that is needed to accomplish memory deduplication. When the time expires, we will conduct a group of writing operations to these pages. If the pages of $F1$ and $F2$ have very short writing delay, they have their own copies. On the contrary, if they are merged, the "copy-on-write" operations will introduce a measurable delay. We can use the results to determine whether or not the SLA has been violated.

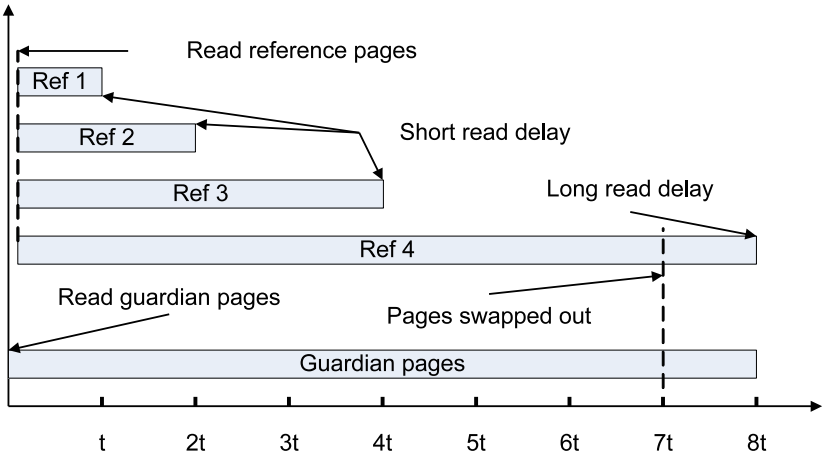


Figure 4: Using reference pages to detect memory swap operations.

Figure 3.c illustrates the detection of memory under-allocation violations. Here the virtual machine will first estimate the size of its active working set. It will then monitor the system performance and use the results in [42] to estimate the physical memory that is allocated to the virtual machine. If the working set is larger than the allocated memory, we can compare the estimation result to the minimum memory size promised by the SLA and detect any violations. On the contrary, the monitoring

results may show that the allocated physical memory is larger than the working set. Under this condition, if the working set is already larger than the promised minimum memory size, the hypervisor is keeping the SLA. If the working set is smaller than the promised value, we will request extra memory from the hypervisor and boost the usage to the SLA amount. We will then measure the access delay to the least recently accessed pages to detect any violations.

2.4 Implementation and Experimental Results

2.4.1 Experiment Environment Setup

The experiment environment setup is as follows. The physical machine has a dual core 2.4GHz Intel CPU, 2GB RAM, and SATA hard drives. We choose a machine with relatively small memory size so that it is easy for applications to exhaust the memory and trigger swapping. The hypervisor that we use is VMware Workstation 6.0.5. We choose this version since it provides explicit interfaces for memory sharing and access between virtual machines. All user virtual machines are using Windows XP SP3 as the operating system. Each virtual machine will occupy one CPU core and 8GB hard disk. The amount of virtual memory that we allocate for each virtual machine will be determined by the experiment. Our experiments show that when there are more than twenty (20) pages that need to be read from the hard disk or separated from the merged memory, the accumulated delay can be accurately detected. Therefore, in our experiments we choose the size of each group of reference pages and guardian pages to be 20.

2.4.2 Experiments and Results

We conduct three groups of experiments to evaluate the detection of unauthorized memory access, violation of deduplication policies, and memory under-allocation violations, respectively. Below we will present the results of the baseline experiments and the detection capabilities and overhead in more complicated scenarios.

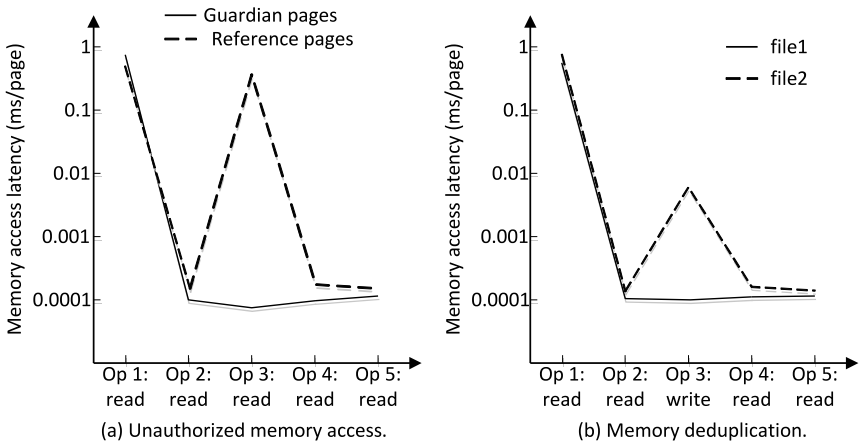


Figure 5: SLA violations detection capabilities of the approaches in baseline scenarios.

The first group of experiments try to evaluate the mechanism for detecting unauthorized memory access. To simplify the experiment setup and examine the practicability of our approach, we initiate only two virtual machines in the physical box: the guest virtual machine that runs our detection algorithms, and an attacker's virtual machine that stealthily accesses the memory of the victim. Instead of locating some malware to penetrate VMware and get access to the guest virtual machine's memory, we propose to use the Virtual Machine Communication Interface (VMCI) [109] to simulate such an attack. Specifically, in the guest virtual machine, we create a block of shared memory so that the attacker's virtual machine can access the guardian pages remotely. The guest virtual machine will load both reference pages

and guardian pages into its memory, as described in Section 2.3.F. After initial access to these memory pages, we would launch some applications to consume all of the memory. In this way, the system will choose memory pages to swap out. Since the attacker's virtual machine remotely accesses the guardian pages, they will be kept in the memory. On the contrary, the reference pages will be swapped out first. When the guest virtual machine measures the access delay to the guardian pages, it will figure out that they are still in the memory and detect the unauthorized access.

Figure 5.a illustrates the detection results. We conduct five reading operations to the memory pages. On the Y-axis we show the average access latency to every page. Since the delays span across multiple degrees of magnitude, we use log-scale Y-axis. Reading Operation 1 has a long delay for both groups of pages since they are loaded from hard disk. Reading Operation 2 is conducted immediately after Operation 1 to verify the contents. The interval between Reading Operations 2 and 3 represents the idle time. We can see that the access latency to reference pages at Reading Operation 3 is much longer than that to the guardian pages since they have been swapped out. After that, we conduct another two rounds of reading operations to measure the delay. From this figure, we could infer that the guardian pages must have been touched by someone after the initial access. Since the access command is not issued by our virtual machine, it is unauthorized access.

The second group of experiments assess the detection of the violations to deduplication policies. We configure the corresponding parameters in VMware so that the page sharing process will scan the memory and merge the pages with identical contents. As illustrated in Figure 3.b, the constructed files $F1$ and $F2$ contain many

such pages. We will access the two files regularly so that they will not be swapped out from the memory. These reading operations will not impact the deduplication procedures. Using the experiment results in [82, 81, 100], we can estimate the time that the algorithm needs to merge the pages. When the estimated delay expires, we will issue a "write" command to the pages of $F2$. If the pages have been merged, the "copy-on-write" operations will introduce a measurable increase in delay. On the contrary, if each page has its own memory, the write delay will be much shorter.

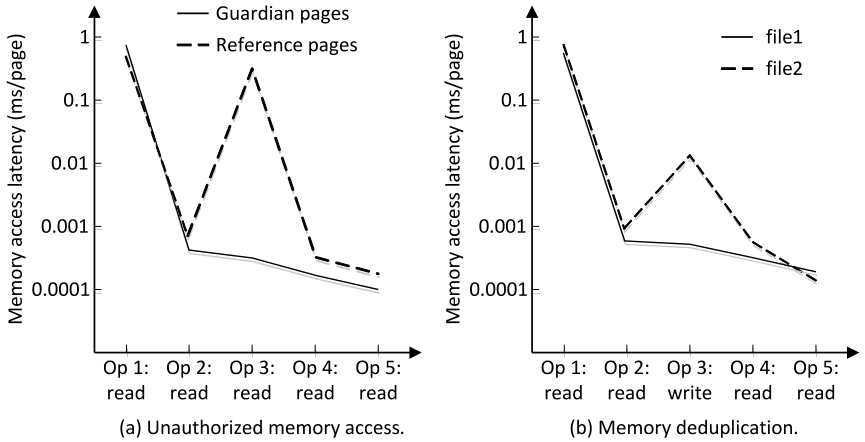


Figure 6: SLA violations detection capabilities of the approaches under intense CPU demand.

Figure 5.b illustrates the detection results. Reading Operation 1 has a long delay for both files since they are loaded from hard disk. The interval between Operations 2 and 3 represents the deduplication procedure. At the Writing Operation 3, we first write to pages of $F2$. We can see that the delay is very long because of the separation of the pages. After that, we write to the pages of $F1$. Since the merged pages have been separated, the writing delay is short. Using this result, we can figure out that the deduplication function is enabled.

We conduct another group of experiments to evaluate the detection capabilities of

the proposed approaches in more complicated scenarios. In this experiment, the guest virtual machine is running the software package *Prime* to generate prime numbers. This application demands a lot of CPU resources. We run the detection algorithms for the two violations. The results are shown in Figure 6.

From the Figure 6, we can see that the proposed mechanisms can still effectively detect the violations. Since our approaches will read from/write to memory pages at a sparse interval, they do not incur heavy CPU overhead. Therefore, the execution of CPU-intensive applications does not impact our approaches to a large extent.

In the third group of experiments, we assess the detection of the memory under-allocation violations. Two experiments with different memory demands are conducted. In experiment one, we initiate one virtual machine (Windows XP SP3) and assume that the hypervisor promises to allocate at least 4GB RAM to the virtual machine. However, the total physical memory in the machine is only 2GB. More memory resources are promised than available memory. We have three applications running in the guest virtual machine. Application 1 reads file $F1$ periodically so that it will be kept in memory. Application 2 reads file $F2$ only once so that its memory pages become the least recently accessed ones. Finally, application 3 consumes more than 2GB but less than 3.5GB memory. From the guest virtual machine's point of view, if it actually has 4GB RAM, the file $F2$ would have stayed in memory. However, Figure 7.a shows that the memory access latency to $F2$ is much longer than that to $F1$. This indicates that the actual RAM is less than 4GB. The memory under-allocation violation is detected.

In experiment two, we have two virtual machines (both Windows XP SP3) and the

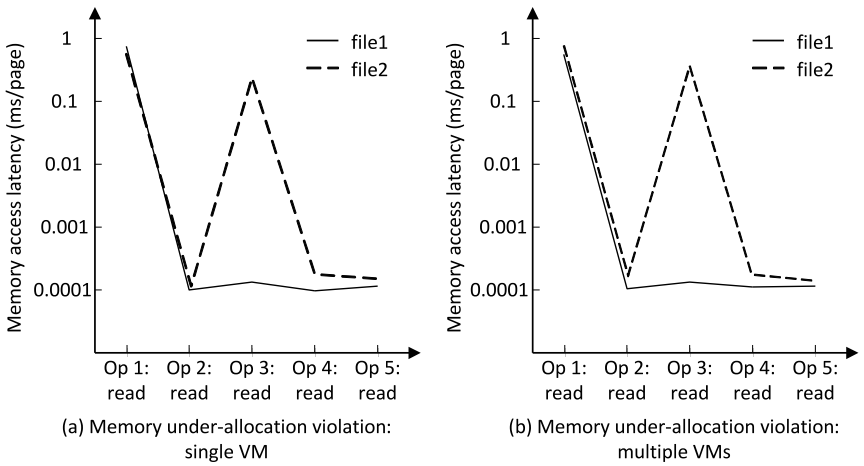


Figure 7: Detection of the memory under-allocation violations.

total physical memory in the machine is still only 2GB. The hypervisor promises 3GB physical memory to each of the virtual machines. Here $VM2$ is running a memory intensive application that demands about 2GB memory. $VM1$ needs about 1GB memory to read the files $F1$ and $F2$ as in experiment 1. The monitoring operations in $VM1$ find out that the active working set is way lower than 3GB. Therefore, extra memory is demanded from the hypervisor to reach the promised amount. After that, reading operations to $F1$ and $F2$ are conducted. Since we read $F1$ periodically, only the pages of $F2$ are swapped out, as shown in Figure 7.b. Again the memory under-allocation violation is detected.

2.4.3 Impacts on System Performance

The proposed violation detection mechanisms demand resources such as memory and CPU time during their execution. Therefore, they may impact the overall system performance. In the following group of experiments, we try to assess such impacts. Since the detection algorithm for unauthorized memory access demands more CPU and memory resources than the mechanisms for the other two types of violations, we

choose it as the object of evaluation. During our experiments, we want to test an extreme scenario. When the detection algorithm is launched, it will allocate many groups of reference pages and try to exhaust the main memory of the guest virtual machine as soon as possible in order to force swapping operations. Other application software will have to compete with our detection algorithm for resources for their execution.

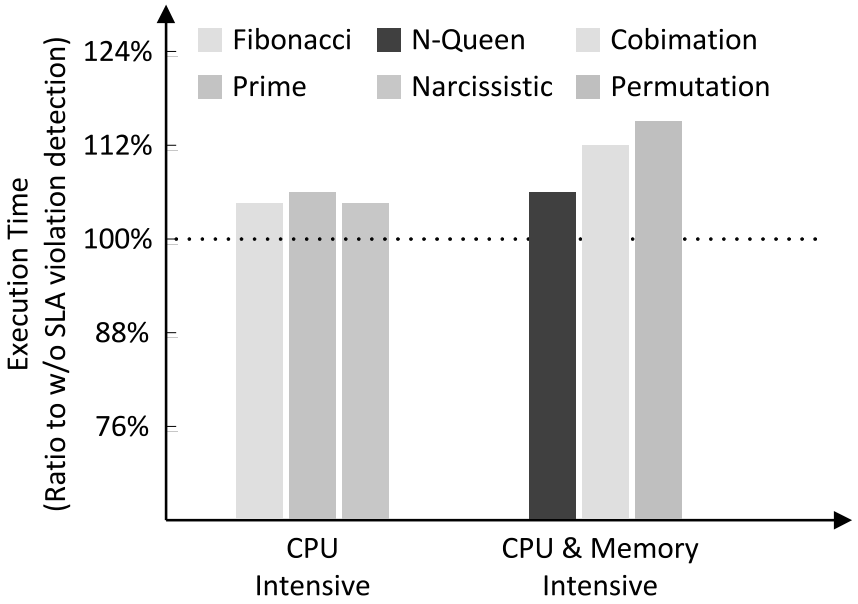


Figure 8: Impacts on system performance during SLA violations detection.

We are especially interested in the impacts on two groups of applications. The first group are the CPU intensive applications. We choose three examples: (1) the *Fibonacci* benchmark that computes the Fibonacci sequence; (2) the *Prime* benchmark that generates prime numbers; and (3) the *Narcissistic* benchmark that generates the narcissistic numbers. Each of these software packages is running in parallel with the detection algorithm for unauthorized memory access. The measured CPU usage is very close to 100%. We measure the execution time of the software since this is the most intuitive parameter that end users adopt to evaluate the system

performance.

The second group include those CPU and memory intensive applications. We also choose three examples: (1) the $N-Queens$ benchmark that computes solutions to the N-Queens problem in chess and stores the results in memory; (2) the *Combination* benchmark that computes all possible combinations of the input numbers; and (3) the *Permutation* benchmark that computes all possible permutations of the input numbers. We measure their execution time when each of them is running in parallel with the proposed mechanism.

From Figure 8, we can see that for CPU intensive applications, the increase in execution time is less than 6%. The increase in execution time for CPU/Memory intensive applications is smaller than 15%. Please note that this is an extreme case since the detection algorithm runs continuously and tries to force memory swapping by grabbing as much RAM as possible. In real world, end users can reduce the detection frequency (e.g. once every 5 minutes). Under that condition, the increased execution time is smaller than 1% on average for both groups of applications.

2.5 Discussion

2.5.1 Reducing False Alarms

The proposed approaches use memory access latency in guest virtual machines to detect violations of SLA on memory management. Different from many interactive security mechanisms that involve third parties, our approaches do not need collaborations from other virtual machines. In this way, it reduces the attack surfaces of the approaches. Below we discuss the schemes that attackers can adopt to avoid detec-

tion and our mitigation mechanisms. We will also discuss the schemes to reduce false alarms.

Since the access latency to a single memory page is too short to be accurately measured, the detection of the three types of SLA violations depends on the accumulated delay. Therefore, the hypervisor or attackers can reduce the memory access frequency and volume to the guest virtual machine to reduce the chance of detection. For example, the hypervisor can randomly select memory pages of the guest virtual machine to read. In this way, when our detection algorithm measures the access delay, only a small percentage of the guardian pages would still be in memory. Attackers can hide the short delay of these pages within other swapping operations. This scheme, however, will also impede the information stealing procedures. For example, our experiments show that if there are more than twenty pages under monitoring are impacted by an attack, the accumulated change in access time can be detected. This will restrict the speed at which an attacker reads the victim virtual machine' memory. For a 1MB data file, if the end user conducts a round of detection every 15 minutes and in every round, the attacker can read only twenty pages to avoid detection, it may take the attacker three hours to read all pages of the file. Many data files, however, may not stay in memory for that long. As another example, the hypervisor may adjust the memory deduplication parameters to reduce the merging speed. Under this condition, the virtual machines will keep a relatively large memory footprint size, which will diminish the purpose of deduplication.

One factor that may impact the detection accuracy of the proposed approaches is the prefetching technique. Prefetching tries to predict the information that the

OS will need in the near future and loads the data in before the actual instructions are issued. This technique may introduce false positive alarms into our system since some guardian pages may be read into memory even though no unauthorized access has been conducted. To mitigate such problems, we can adopt two schemes. First, prefetching uses the property of locality and is usually applied to the subsequent pages of current contents. If we know the number of pages that a prefetching operation will load, we can use the guardian pages beyond the prefetching range to detect violations. Another scheme that we can use is to chain the guardian pages together to form a linked list. This technique can also diminish the impacts of prefetching.

2.5.2 Impacts of Extra Memory Demand

Some users may worry that the extra memory demand for the detection of under-allocation violations will impact the system performance. We will justify the approach from the following aspects. First and most importantly, the extra memory demand will happen only when the active working set is smaller than the minimum memory amount promised by the SLA. At the same time, if the hypervisor is keeping its SLA, this memory would have belonged to the virtual machine any way. Under this case, the demand will not impact the system performance. Second, the memory demand is incurred only during the detection procedures, which would happen infrequently. Last but not least, our experiment results in Section 2.4.C have shown that the detection algorithms will cause very small impacts on the system performance.

2.5.3 Building A Unified Detection Algorithm

For the clarity of the chapter, we have presented the detection algorithms for the three types of violations separately. In real life, we can build a unified detection algorithm. As shown in Figure 9, the unified detection mechanism will consist of five components. The memory management component will communicate with the virtual machine to request and return memory pages. A timer will issue commands to the memory reading/writing component based on pre-determined clock intervals so that certain pages will be kept in memory. The reading/writing component will also work with the time measurement component to get the accurate access latency. The measurement results of these operations will be provided to the detection component. Based upon different target violations, the algorithm will return detection results to the end user.

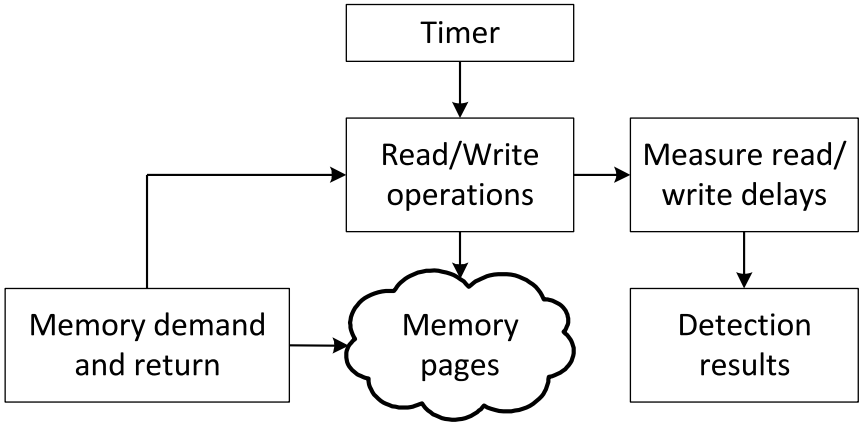


Figure 9: Architecture of a unified SLA violations detection algorithm.

2.6 Conclusion

In this chapter, we propose mechanisms to detect violations of the SLA on memory management in virtual machines. Instead of proposing a generic security SLA

enforcement architecture, we design mechanisms to detect three types of memory management violations in guest virtual machines. We have implemented the detection approaches under VMware and tested them. The results show that they can effectively detect the violations with a small increase in overhead. We also discuss the techniques to improve our approaches.

Immediate extensions to our approaches consist of the following aspects. First, we plan to explore other types of security SLA violations in memory management and design a generic approach for their detection. We will also experiment with other hypervisors such as extended Xen and Linux KSM to generalize the mechanisms. Second, we want to study the relationship between our approaches and existing security SLA enforcement architectures. If we can integrate them into the existing architecture, we will have a solid platform for future extension. The research will provide new information for strengthening protection to virtual machines and end users of cloud computing.

CHAPTER 3: ROOTKIT DETECTION ON GUEST VIRTUAL MACHINES THROUGH CROSS-VERIFIED EXTRACTION INFORMATION AT HYPERVISOR-LEVEL

3.1 Introduction

Computer systems face the threats from many kinds of stealth attacks such as rootkits [48, 17]. A rootkit is a stealthy type of software, often malicious, designed to hide the existence of certain processes or programs from normal methods of detection and enable continued privileged access to a computer [71]. Once attackers have obtained root or administrator access to a system, they will install rootkits to hide the evidence so that system administrators cannot detect them. In addition to stealing sensitive information, attackers also use rootkits to create backdoors for subsequent attacks.

Rootkits are difficult to detect since a rootkit tries to hide its existence from anti-malware programs. Existing approaches to rootkit detection can be classified into three groups. In the first group, researchers analyze and characterize rootkit behaviors [122, 60, 123, 87, 83, 54]. HookFinder [122] provides valuable insights and details about the underlying hooking mechanisms used by attackers. K-Tracer [60] and Panorama [123] are automatic tools that can efficiently analyze the data access and manipulation behaviors of rootkits. In the second group, researchers try to detect the rootkits through certain symptoms exhibited by the intrusion [61, 62, 9, 90]. For

example, SBCFI (state-based control-flow integrity) [77] monitors kernel integrity of the OS to detect malicious changes. Copilot [76] implements a similar approach in coprocessor platforms. In the third group, approaches are designed to prevent rootkits from changing the OS kernel[43, 117, 94]. In [58], the authors present a technique that uses static analysis to identify instruction sequences of malicious activities.

The host-based rootkit detection mechanisms have their limitations. For example, some rootkits such as Agobot variant [129] can detect and remove more than 105 types of anti-malware programs in the victim machine. The emergence of cloud computing opens a new horizon for solving this problem. In a virtualized environment, the hypervisor can monitor the behaviors of the virtual machines. While a rootkit may be able to fool the guest OS, it will be very difficult to hide the malicious process from the hypervisor. Several security systems have been developed for rootkit detection in virtual machines [93, 44, 120, 5, 126]. For example, VMwatcher [49] uses the general virtual machine introspection (VMI) [32] methodology in a non-intrusive manner to inspect the low-level virtual machine states. UCON (usage control model) [118] is an event-based logic model. It maintains the lowest level accesses to the system and ensures that such accesses cannot be compromised by internal processes of a virtual machine.

Existing hypervisor-based rootkit detection mechanisms use information from different modules of a virtual machine as individual components to conduct malware detection. Since the data is not cross-verified among different modules, some malware could have escaped detection. For example, VMwatcher [49] compares the process names at the virtual machine level and hypervisor level to identify any hidden pro-

grams. An attacker can change the process' name to a frequently-used text editor to avoid detection. However, when we cross-examine the process table with opened files, we may detect that the text editor has opened a TCP connection and turned on the microphone. The mismatch of information among different modules will allow us to detect the rootkit that hides deeper in the system.

In this chapter, we propose a rootkit detection mechanism based on deep information extraction and cross-verification at the hypervisor level. Since the hypervisor sees only the raw memory pages of a virtual machine, we need to first reconstruct the semantic view of a virtual machine's memory in order to recover its execution states. The recovered information includes processes, network connections, kernel-level modules, and opened files. After reconstructing the semantic view of a virtual machine's memory, we examine the execution states that are directly obtained from the virtual machine and those reconstructed by the hypervisor. Through cross-verification among different modules of the two views, we can find the discrepancy between them and identify the hidden malware in the guest OS.

While the basic idea is straightforward, we must overcome two challenges to turn it into a practical approach. First, there is a "semantic gap" [21] between the memory viewed by the virtual machine and that by the hypervisor. In hypervisor, we see only the raw memory pages, registers, and disk blocks. Therefore, we must establish a native view of the virtual machine's memory just like we are in the virtual machine. The second challenge that we face is to cross-verify different modules of the reconstructed memory view and identify any mismatch in the information. In this preliminary version of research, we define a static table that links frequently used

applications to the file types and network connections that they can operate on. A more intelligent approach will be designed in future work.

Compared to existing rootkit detection mechanisms in virtualization environments, the proposed approach has the following advantages. First and most importantly, we use the reconstructed information from different modules of the virtual machine as an integrated, cohesive system for rootkit detection. This provides us a stronger detection capability than existing approaches. Second, our rootkit detection mechanism uses non-intrusive introspection of virtual machines. Therefore, it incurs very limited overhead in the virtual machines. Last but not least, we implement the proposed approach in XEN and show that it has very small performance impacts on the virtualization environment.

The remainder of this chapter is organized as follows. In Section 3.2 we describe the details of the proposed approach. We discuss the reconstruction of the semantic view of a virtual machine's memory. In Section 3.3 we present the implementation of the rootkit detection mechanism and assess the performance impacts when XEN Hypervisor is used. Section 3.4 discusses several issues in the detection procedure. Finally, Section 3.5 concludes the chapter.

3.2 The Proposed Approach

In this section, we will present the details of the proposed approaches. We first discuss the assumptions of the environments to which our approaches can be applied. We also introduce the technique of memory reconstruction which we use to extract high-level execution states of the virtual machine. Finally, we present the details of

the approaches and mechanisms to detect rootkits.

3.2.1 System Assumptions

In the investigated scenario, we assume that an attacker has acquired the administrator privilege in the target virtual machine and she/he can install malware in the system. The rootkit embedded by the attacker can modify the returned results to the auditing programs on the virtual machine (e.g., ps, netstat or lsof on the guest OS) to hide the intrusion. The attacker can leave a backdoor in the system for subsequent attacks. However, similar to the approaches in [125], we assume that the attacker cannot compromise the hypervisor.

Our investigation has the following design goals. First, the proposed approach should be transparent to end users. It can accurately extract and recover the virtual machine's execution states without any help from the virtual machine. Second, we need to minimize the performance impacts of the proposed approach on the target virtual machine. This requirement will help us avoid difficulty in future adoption. Third, the proposed approach must be hypervisor independent. The design should support VMM in both full virtualization (e.g. KVM [56] and VMware [110]) and paravirtualization (e.g., XEN [12]). This property will allow more users to benefit from the approach.

3.2.2 Memory Reconstruction

The first step of the proposed approach is memory reconstruction of the virtual machines at the hypervisor level. Through memory reconstruction, we can extract the high-level semantic information of the virtual machine. To accomplish this procedure,

we need to locate the static data entries that are essential for the kernel and the boot-up procedures. Since many of these entries are accessed frequently, their addresses are determined at the compilation time and can be found from the kernel symbol table (i.e., *System.map* in Linux), which can be viewed as a look-up table between symbol names and their addresses in memory.

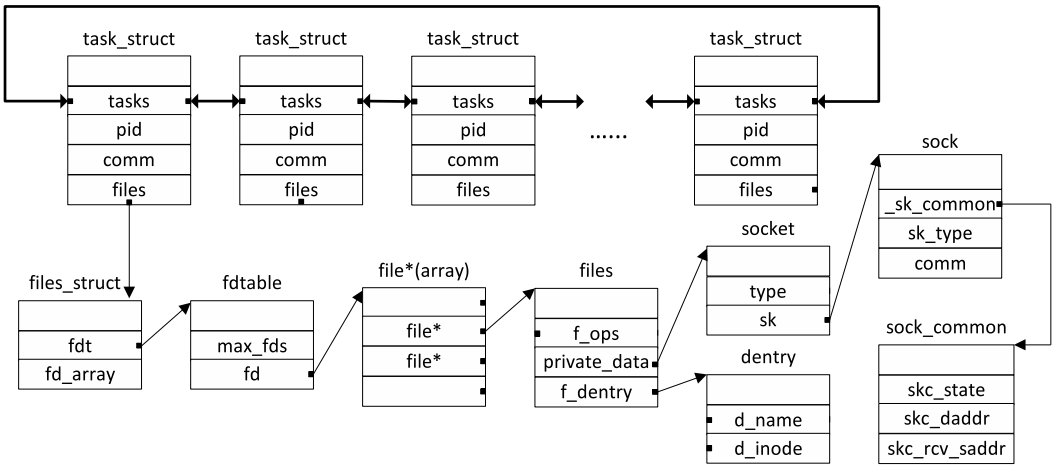


Figure 10: Semantic view of a virtual machine's memory through memory reconstruction.

In our approach, we first reconstruct the disk image of the virtual machine on the hard drive in order to get access to the kernel symbol table (i.e., $/boot/System.map-\$(uname-r)$ in Linux). Once we know what the file system is and how the files and directories are organized in the virtual disk, we can reconstruct the semantic view of the virtual disk from the raw virtual disk. For example, it is easy to reconstruct the raw virtual disk of a Linux virtual machine because of its open-source kernel structure. The reconstruction results will then allow us to determine the memory addresses of the static variables in the kernel symbol table.

To reconstruct the semantic view of a virtual machine's memory, we need to translate the memory address in a virtual machine to the physical address in the host

machine. Through the reconstruction of the virtual disk, we already know the addresses of many static variables. For example, *init_task* is a statically declared task that is at the head of the process list in Linux. Starting from this position, we can go through the complete process list and get the information of all processes using the offsets of the entries such as PID, process name, and the pointer to the next process.

To better explain the memory reconstruction procedure, below we use an example of a 32-bit Linux guest OS to illustrate the information extraction operations. The procedure is shown in Figure 10. In this example, the user space occupies the bottom 3GB memory while the kernel occupies the top 1GB. From the kernel symbol table, we know that the address of *init_task* is $0x81c0d020$. The offset of tasks (struct *list_head*) is $0x240$. Therefore, the starting address of the first process is $*(init\_task+offset\_tasks)$ - $offset\_tasks$. Using the similar technique, we can reconstruct a number of other important data structures such as files and sockets. We have used several functions such as *vmi_read_addr_va* and *vmi_read_32_va* to translate the virtual addresses to corresponding physical addresses.

3.2.3 Rootkits Detection Based On Cross-Verification

As described in Section 3.1, it is not sufficient to only examine whether or not the execution states at the virtual machine level and the hypervisor level match with each other. In this part, we design a rootkit detection mechanism that explores deep information about the virtual machine, especially the relationships among the running processes, active network connections, and opened files to detect the anomalies caused by malware. The procedure is illustrated in Figure 11.

When both the virtual machine and the hypervisor have finished extracting the execution states of the virtual machine, they can compare the results. If any discrepancy is detected, the hypervisor can raise an alarm of the hidden information. If the two views are identical, the hypervisor will then cross-examine different components of the extracted information. In our preliminary implementation, we study the relationships among the processes, network connections, and opened files. Using the guest OS Ubuntu (64bit) with 3.5.0-23-generic kernel, we have generated a table for the processes of frequently used applications. In this table, we have defined the file types that this process can open, whether or not this process can be associated with network connections, and any special properties of the connections (such as the number of concurrent connections, the protocols, and the port numbers). The hypervisor will cross-examine different components of the extracted execution states against this table. If any violation is detected, an alarm will be sent to the end user. In the next section, we will present a concrete example to show the effectiveness of the proposed approach.

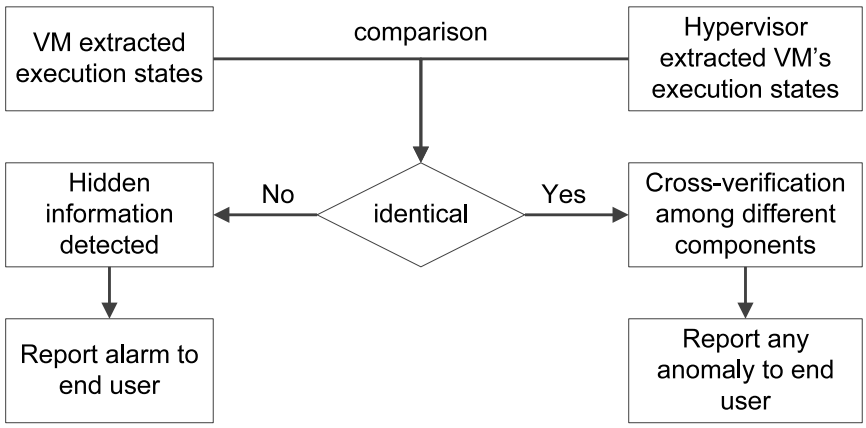


Figure 11: Rootkit detection procedure through cross-verification.

3.3 Implementation and Experimental Results

3.3.1 Experiment Environment Setup

To evaluate the detection capabilities of the proposed approach and assess its impacts on the system performance, we conduct two groups of experiments on the hypervisor Xen version 4.1 with the library libvir 0.9.8. In the first group, we test our approach on Paravirtualization (PV) over Xen. The host OS is Ubuntu Desktop 12.04 LTS (64bit). The PV guest OS is Ubuntu (Precise 64bit) with 3.5.0-23-generic kernel. In the second group, we test our proposed approach on Hardware Virtual Machine (HVM) guest. The host OS is Ubuntu Server 12.04 LTS (64bit). The HVM guest OS is Ubuntu Desktop 10.04 LTS (32bit) with 2.6.32-24-generic kernel.

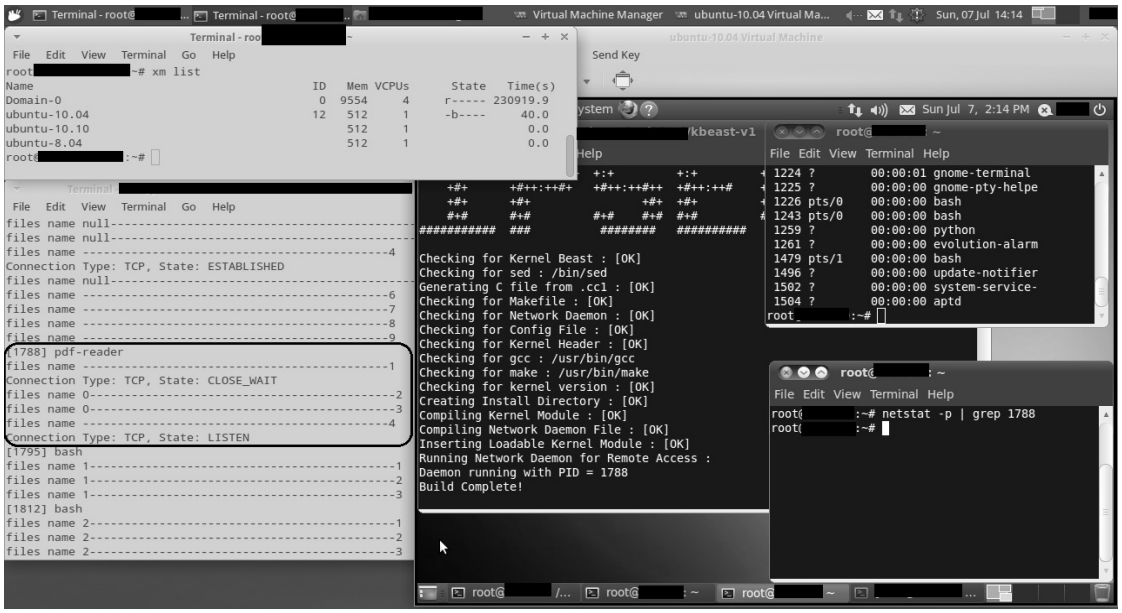


Figure 12: The detection of KBeast in Xen through cross-verification.

While the virtual machines can use different types of guest OS, below we use Linux as an example to show the information extraction procedure. As shown in Figure 10, in Linux for any opened file or socket we can access the data structure $file$ to get

its information. Through comparing the field f_ops to the value of $socket\_file\_ops$ in the Kernel Symbol Table, we can determine whether the handle points to a file or a socket connection. If the handle actually points to a file, we can use the field d_name in the structure *dentry* to get the file's name. Otherwise, we can use the *socket* structure to get the information about the network connection.

3.3.2 Experiments and Results

View comparison-based rootkit detection discovers malware through finding information mismatch between the hypervisor and the virtual machine. The discrepancy could be caused by either information hidden by the malware, or the anomaly links among the processes, files, or network connections. Below we use a concrete example to illustrate the detection capability of the proposed approach. Here we use an advanced Linux kernel rootkit KBeast as the investigation target. KBeast can hide loadable kernel modules, processes (ps, top, lsof, pstree), socket and connections (netstat, lsof), and anti-kill processes from the infected OS. Furthermore, it leaves a hidden backdoor open for subsequent attacks. KBeast also allows the attackers to change its process name so that it looks like a benign application. It is so stealthy and elusive that many anti-malware packages such as chkrootkit and rkhunter cannot detect it. Figure 12 shows the screenshot of an infected virtual machine in which the KBeast runs and hides a process with the PID 1788. In Figure 12, the background GUI screen on the right side shows the inside view of the virtual machine while the foreground screen on the left side shows the reconstructed semantic view of the memory of the same virtual machine.

In the Figure 12, we can observe that the rootkit KBeast is built completely and it is running with PID = 1788, which is hidden from the virtual machine. The attacker has also changed the process' name to 'pdf-reader' so that looking at only the process table is not sufficient for its detection. The malware opens a backdoor for remote access through telnet. In the virtual machine, we cannot detect any anomaly through ps or netstat. The reconstructed view of the virtual machine at the hypervisor level is shown on the left side of the Figure 12. In the left bottom of the Figure 12, we can see that the proposed approach reveals a running process with PID=1788 called *pdf-reader*. If we look at only this information, we may assume that the virtual machine has the application such as Acroreader running. However, when we link the processes to opened files, we find that this process has opened four files/connections. The first TCP connection is the telnet backdoor and its state is CLOSE_WAIT. The state of the second TCP connection is LISTEN. It is the backdoor that KBeast opens. This is a very suspicious activity when the PDF reader holds an open connection and waits for external requests. The proposed approach detects this anomaly and reports it to the virtual machine.

In addition to the experiments described above, we have conducted another group of experiments to test the proposed approach upon a PV virtual machine. Here the rootkit replaces the system auditing programs (including lsof and netstat) with some malicious interfaces so that it can control the returned contents. Our experiments show that the proposed approach can also reveal the hidden information through memory reconstruction, including the opened files, Socket connections, and the IP addresses of the communication parties. The experiments show that our approach is

not restricted by the type of virtualization.

3.3.3 Overhead and Performance Analysis

To protect a virtual machine from rootkit infection, we need to execute the proposed approach at the hypervisor level periodically. Since we need to freeze the virtual machine during memory reconstruction, we must study the relationship between the detection frequency and its impacts on the system performance. We conduct two sets of experiments to assess the impacts.

The first experiment tries to measure the memory reconstruction time at the hypervisor level. Since each memory reconstruction consumes a very short period of time, we measure the accumulative delay of 5000 reconstructions. In order to simulate the real working environment, after each memory reconstruction the host OS will sleep for one second so that the virtual machine will get the CPU back. Our measurement shows that the average execution time of memory reconstruction is about $20ms$ regardless of the size of the allocated memory of the virtual machine since we access only its high level data structures.

In the second group of experiments, we try to evaluate the impacts of the proposed rootkit detection mechanism on the performance of the guest virtual machine. We choose three applications in the guest virtual machine as benchmarks and measure their execution time when the rootkit detection frequency is changed. As shown in Figure 13, the *make* benchmark compiles the kernel of Linux version 3.10 and incurs intensive CPU and I/O workload. The *gzip* benchmark compresses a large file and demands more resources from the CPU. Lastly, the $find$ benchmark tries to locate a

specific file on the hard-drive and incurs intensive disk and file system accesses.

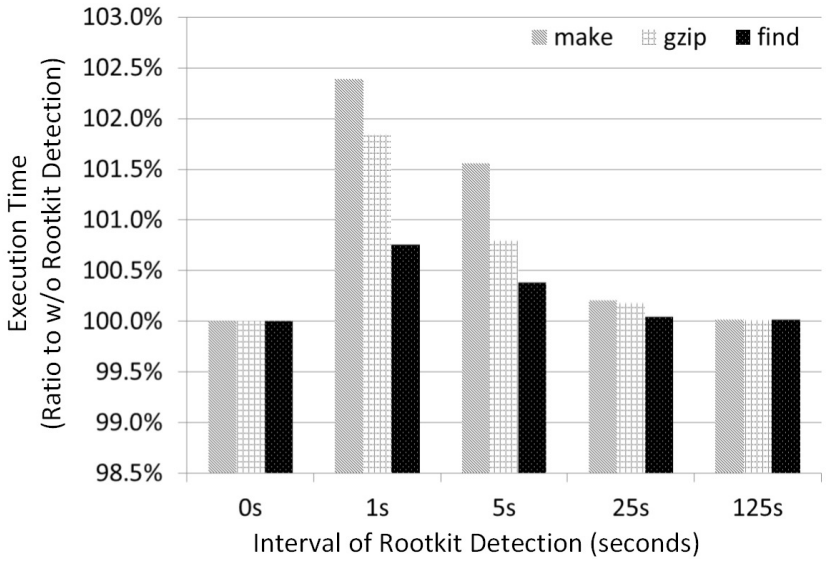


Figure 13: Impacts of rootkit detection frequency on system performance through cross-verification.

From the Figure 13 we find that when we increase the interval between the rootkit detection, its impacts on the virtual machine performance are decreasing. When the interval is equal to 125 seconds, the applications are almost not affected at all. Even when we execute the proposed approach every second, the increase in application execution time is less than 2.5%. Based on the results, we conclude that our proposed approach introduces very low overhead.

3.4 Discussion

While there are many different ways for hypervisors to communicate with hosted virtual machines, in our implementation we use sockets to allow the two parties to exchange information. We establish a socket in the hypervisor that listens to the rootkit detection requests from virtual machines. When a request is received, the hypervisor will temporarily freeze the virtual machine and access its memory. The

memory reconstruction and malware detection results will then be sent back to the virtual machine. To protect authenticity and integrity of the data, the hypervisor can digitally sign the hash result of the exchanged information. The two parties do not need accurate synchronization to look at the same snapshot of the memory since the hypervisor can identify any suspicious information in the reconstructed view.

Our proposed approach extracts the execution states of a virtual machine through reconstructing the semantic view of its memory. If an attacker has compromised the virtual machine, she/he can intentionally modify the significant values of the system in the Kernel Symbol Table. Under this condition, we cannot reconstruct a true semantic view of the virtual machine's memory. Similarly, the attacker can also alternate other components of the kernel to fabricate a false view that the information from the virtual machine and that from the hypervisor match to each other. Fortunately, changes to OS kernels can be detected through attestation of the integrity of the images [72].

3.5 Conclusion

In this chapter, we propose a new rootkit detection mechanism for virtual machines through deep information extraction and reconstruction at the hypervisor level. The hypervisor will first rebuild the semantic view of the virtual machine's memory. Through cross-verification among different components of the reconstructed view, the hypervisor can detect hidden information and mismatch among different active modules in the virtual machine. Our experiment results show that the proposed approach is practical and effective in rootkit detection. Furthermore, the performance overhead is very low since we access only the high-level data structures of the virtual machine.

Immediate extensions to our approach consist of the following aspects. First, we plan to experiment our approach with Windows virtual machines so that we can evaluate its practicability in other environments. Second, we will introduce intelligence into the construction of and anomaly detection in the linkage table among different modules of the virtual machine. Finally, we plan to extend our approach to other hypervisors so that more end users can benefit from our research.

CHAPTER 4: LIGHTWEIGHT EXAMINATION OF DLL ENVIRONMENTS IN VIRTUAL MACHINES TO DETECT MALWARE

4.1 Introduction

Computer systems face the threats from a high spreading rate of computer malware (worms, Trojan horses, rootkits, botnets, etc.). Malware intrudes into computer systems and causes millions of dollars in damage. Host-based malware detection mechanisms have their limitations. On one side, since the anti-malware systems are installed and executed inside the hosts that they are monitoring, they can collect rich information from the local host. On the other side, since they are visible and tangible to advanced malware running in the system, effective attacks towards them become feasible. For example, some malware such as Agobot variant [129] can detect and remove more than 105 types of anti-malware programs in the victim machine.

Since it becomes increasingly difficult to trick end users to download and run executable files from unknown sources, attackers refer to more stealthy ways to avoid detection. Based on a report released by Kaspersky [78], about 60% of malware collected at KingSoft anti-malware lab are DLL files. From this point of view, protecting the authenticity and integrity of DLL files that are loaded into computer systems is essential for the safety of end users.

The emergence of cloud computing opens a new horizon for combating with the trends in malicious attacks on DLL files. Some researchers [51, 57, 114] proposed

to place the intrusion detection mechanisms outside of the virtual machine being monitored. A well-implemented hypervisor will enforce strong isolation among virtual machines and the programs running within them. Under this condition, even if a virtual machine is compromised by malware, it is difficult for the attacker to compromise the hypervisor. For example, VMwatcher [49] uses the general virtual machine introspection (VMI) methodology in a non-intrusive manner to inspect the low-level virtual machine states. UCON (usage control model) [118] is an event-based logic model. It maintains the lowest level access to the system and ensures that such access cannot be compromised by internal processes of a virtual machine.

Existing malware detection approaches often use information from DLL or other executable files in the following way. They will collect a large number of PE or DLL files and conduct static analysis of the API calling graphs in these applications [14, 96]. The learned knowledge will then be used as features to detect malware. Example solutions include [91, 95, 121]. These approaches are effective in the detection of infected files when they are stored in hard-drive. However, if no continuous examination is conducted, they cannot capture infections to the DLL files that happen after their initial screening. Another thread of research uses signatures of different pieces of kernel code [68] or cross-compares code segments among multiple virtual machines [3] for code integrity checking purposes. However, it quickly becomes cumbersome and time-consuming to maintain a database of all legitimate signatures. For example, ModChecker [3] will introduce a delay of 0.2 second when it tries to compare *http.sys* on two mostly-idle virtual machines.

In this chapter, we propose a lightweight approach at the hypervisor level to con-

tinuously monitor the status of loaded DLL files in guest virtual machines to detect malware. Instead of using information that is extracted from different modules of a virtual machine as individual components, our cross-verification schemes cover a wide range of properties of the DLL files including their loading path, loading order, and RVA (relative virtual address) of the functions. Our overall approach can be divided into three steps: collection, analysis, and monitoring. Through memory reconstruction technology of the virtual machines, we record the execution states of different applications in the virtual machines at the hypervisor level. Using freshly installed virtual machines, our collection procedures will extract and record information of DLL files in malware-free environments. After we collect enough information from the training data, we will start the analysis procedures. We will explore the relationship between the active processes and the loaded DLL files. We will also generate a fingerprint of the loading order and RVAs of the DLL files. We will then continuously monitor the DLL files running in the virtual machines and compare them to the extracted features. If attackers make any changes to the information under surveillance, we can detect the infection in real time.

The contributions of our research can be summarized as follows. First, instead of examining the DLL files for only once when they are loaded, our approach conducts continuous monitoring on the files. In this way, infections to the libraries can be detected on the fly. Second, our malware detection mechanism uses non-intrusive introspection of virtual machines. Since the detection mechanism is running at the hypervisor level, it is very difficult for malware running in a virtual machine to detect, remove, or avoid our approach. Third, instead of examining the contents of the whole

libraries, we focus on some high level yet essential information such as the RVA of the functions. In this way, even when we examine the virtual machines' memory at a relatively high frequency, the impacts on their performance are still very low. Last but not least, we conduct extensive experiments to evaluate the proposed approach. We use more than 100 malware of different types (Trojans, stealth backdoors, adware, and virus) to test our detection mechanism. Our solution detects almost all malware samples with very low false negatives.

The remainder of this chapter is organized as follows. In Section 4.2, we discuss the related work. In Section 4.3, we present the details of the proposed approach. In Section 4.4, we describe the implementation of the malware detection mechanism and experiment results. Section 4.5 discusses a few issues in the detection procedure. Finally, Section 4.6 concludes the chapter.

4.2 Related Works

Existing approaches to manipulating DLL files in a computer system can be classified into two groups. In the first group, malware will try to load its own DLL files into the system through DLL injection [4]. An attacker can achieve the goal through either remote thread injection or registry DLL injection. For example, Conficker worm injects undesirable DLLs into legitimate software [99]. Another way of manipulating the DLL files is to change the files directly. For example, attackers can use API hooking [122] to redirect a benign function call to malicious code segments. In-line code overwriting can also be used to achieve the goal [112].

Our approach can effectively detect the attacks described above. During the col-

lection phase, we have learned the relationship between the application software and the DLL files loaded by it. Therefore, if attackers try to link their malicious DLL with an application, we can detect the anomaly. Since our approach will examine the RVA of the function calls, API hooking can be detected as well.

Researchers have experimented with malware detection through verifying the integrity of system files. For example, SBCFI (state-based control-flow integrity) [77] monitors kernel integrity of the OS to detect malicious changes. Copilot [76] implements a similar approach in coprocessor platforms. The disadvantage of these approaches is their relatively heavy overhead on the system. In our approach, we choose to examine only the high-level information instead of the complete file contents to reduce impacts on the virtual machines. More discussion on this choice is presented in Section 5.

4.3 The Proposed Approach

In this section, we will present the details of the proposed approaches. We first discuss the assumptions of the environments to which our approaches can be applied. We also introduce the technique of DLL information extraction which we use to extract DLL information of virtual machine. Finally, we present the details of the approaches and mechanisms to detect malware.

4.3.1 System Assumptions and Design Goals

In the investigated scenario, we assume that we can get access to malware-free virtual machines during the collection and learning phases of the approach. This can be achieved through using freshly installed systems. After the collection procedures,

we do not restrict user behaviors on the virtual machines. They may be tricked by attackers and download or install some adware, virus, worm, or spyware into the system. We also assume that an attacker can acquire the administrator privilege of a compromised virtual machine after she/he intrudes into the system. Malware installed by the attacker may modify return results to anti-virus programs that are running in the local virtual machine to hide the malicious process. The malware may also inject in some benign process, such as explorer.exe, and start several threads to conduct malicious activities. However, similar to the approaches in [125], we assume that the attacker cannot infect the hypervisor through the virtual machine.

Our investigation has the following design goals. First, the malware detection mechanism should be transparent to end users. Moreover, it can extract and recover the virtual machine's execution states accurately. This goal is realized through the non-intrusive introspection technology. Since we examine the virtual machine execution states from the hypervisor, it is very difficult for the malware to mislead the detection procedure. Second, the approach should be independent of specific hypervisors. The design should support VMM in both full virtualization mode (e.g., VMware[110] and KVM [56]) and paravirtualization (e.g., XEN [12]) modes. This property allows more users to benefit from the approach. The analysis in later parts will show that our malware detection mechanism does not depend on any specific hypervisors. Third, we also need to control the performance impacts of the proposed approach on virtual machines. This requirement is essential for future deployment and adoption of our malware detection mechanisms. Our experiments will show that examining only the execution states in virtual machine memory, even periodically with a short time

interval, will introduce a small increase in overhead.

4.3.2 DLL Information Extraction

Before we can collect and analyze information from guest virtual machines, we must first reconstruct memory of the virtual machines at the hypervisor level. Since the hypervisor sees only the raw memory pages of a virtual machine, we need to rebuild its semantic view, so that we can extract high-level semantic information.

To better explain the memory reconstruction procedure, below we use an example of a Windows guest OS to illustrate the information extraction operations. We can go through the process list from $PsActiveProcessHead$ (the head of the double linked list). In Windows XP, each process has an $\_EPROCESS$ object through which we can traverse the whole list. Each $\_EPROCESS$ object contains both $Flink$ (forward link) that points to the next $\_EPROCESS$ structure and $Blink$ (backward link) that points to the previous $\_EPROCESS$ structure. $PsActiveProcessHead$ is a member of the kernel debugger data block ($\_KDDEBUGGER\ \_DATA64$), which is used by the kernel debugger to find out the states of the operating system [66]. Furthermore, the Kernel Processor Control Region (KPCR) is a data structure used by the Windows kernel to store information about each process. It is located at virtual address $0xffdff000$ in Windows XP. In KPCR, the data structure $KdVersionBlock$ contains a linked list of $\_KDDEBUGGER\_DATA64$.

Through memory introspection technique, we can extract high-level execution states of a virtual machine. The execution states that we can get include process list, network connections, opened files, Dynamic-link library, and relative virtual address of

functions in DLLs. In the process list, we could get the full path of the files in execution. For each process, the order of the extracted DLL files is identical to the order in which the process loads them into memory. Moreover, we can get the relative virtual address (RVA) of functions in different DLLs. Access to the information provides a rich data set for us to conduct subsequent analysis and design of detection mechanisms.

4.3.3 Design of the Malware Detection Mechanisms

At the high level, we propose a malware detection mechanism that runs in the hypervisor to detect infected DLL files in guest virtual machines. This is accomplished in three steps (Figure14): (1) the collection phase, in which a process collects information about different applications from malware-free virtual machines; (2) the analysis phase, in which we analyze the execution states of each benign process, and extract the characteristics of these benign applications; and (3) an on-line detection phase, in which the detection program is used to detect infected DLLs in a guest virtual machine through comparing their execution states to the learned information. These three steps are described in more details below.

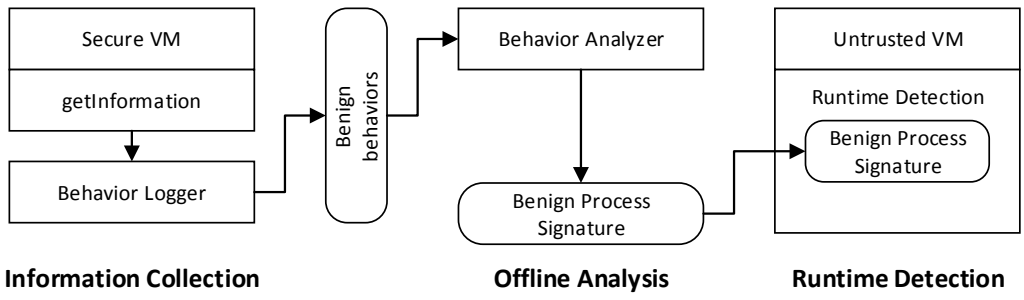


Figure 14: DLL examination malware detection procedure.

First, we use a running example to illustrate the information collection procedure. In order to get a comprehensive view of the execution states and avoid impacts from a specific running environment, our information collection program will run many times in the hypervisor to extract behaviors of each process from different installations of malware-free virtual machines. For example, in a PE file, the export structure is called $Image\_Export\_Directory$ with eleven (11) members in it. $AddressOfFunctions$ is the head of the array that keeps RVAs to all functions in the module. $AddressOf-Names$ is the head of another array that keeps the names of functions in the module. Combining these two arrays, we can get export functions and RVAs in pairs.

Although the information that we gather through the collection phase contains a lot of data that can be used to differentiate malware from benign applications, two reasons lead us to apply some learning algorithms to filter out noises and generate behavior patterns of benign processes. The first reason is that different versions of the same software demonstrate different behaviors. We can use an example of the RVAs of the same function to illustrate this. In $ws2\_32.dll$ with version 5.1.2600.5512, the RVA of function *socket* is $0x00004211$, while the RVA of the same function in version 6.3.9600.17415 is $0x00003BD0$. The second reason is that even when the same application software with the same version number is installed, the virtual machines may still demonstrate different behaviors in different environments. For example, under most conditions, *explorer.exe* will not load $avcuf32.dll$ into memory. However, if we install $Bitdefender$ in our virtual machine, which is an anti-virus application, *explorer .exe* will load $avcuf32.dll$ into memory after $KERNEL32$ $.dll$. Similarly, *explorer.exe* will load 7-*zip.dll* into memory only if we install 7-*zip*. Because of the

differences in behaviors, we need to experiment with the same application in different running environments to learn their behaviors.

The knowledge that we learn from the malware-free virtual machines covers a wide range of properties of the applications. For example, we will check the DLL names, their full loading paths, and RVAs of functions in them. In this way, if an attacker loads his own malicious DLLs into the system, we will be able to catch them. Some malware may impersonate a popular process name, such as *svchost.exe*, to fool the detection algorithm. The fake process, however, usually needs to load DLLs from a folder that is different from that of the real application. Therefore, we can distinguish between them based on these differences.

Another feature that we can use to detect malware is the relationship between the functionality of an application and the DLLs it loads. DLL files usually serve specific purposes. For example, $ws2\_32.dll$, $hnetcfg.dll$, $pstorec.dll$, and $crypt\_32.dll$ are used for networking, firewall maintenance, access to protected storage, and cryptography, respectively. An application should load only the DLLs that it needs to use. Through analyzing the functionality of different applications and the DLL files that they load, we can expose their relationship. If DLL files that deviate from the functionality of an application are loaded, we need to conduct further investigation. For example, if a malicious application calls functions from all four DLLs we mention above, it can read confidential information from the system, encrypt it with secret keys, and send it out to the attacker. This type of anomaly can be detected through cross comparison among the DLLs that are loaded by the applications with similar functionality.

We can also use dependency among DLL files to detect malware. Their dependency

could impact the order in which they are loaded into the system. For example, $IEXPLORE\ .EXE$ is a frequent target of attackers. We analyze its behaviors and find out that there are 16 groups of consecutive loading orders of DLL files. The length of these consecutive segments ranges from 2 to 20. Some DLL injection attacks will break these consecutive segments. Therefore, we can use this change to detect the malware.

Last but not least, through checking the RVA addresses of the functions, our detection mechanism can catch several types of code injection and in-line code overwriting attacks upon DLLs. In order to increase the efficiency of our detection mechanism, we will examine the RVAs of the functions as one unit by calculating and checking their hash result.

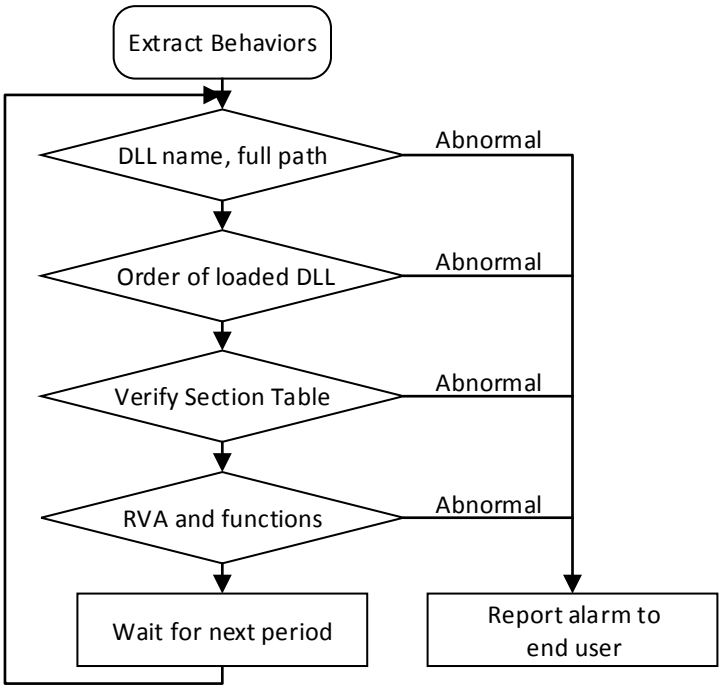


Figure 15: Runtime detection procedure of DLL examination.

After analyzing the extracted information from malware-free virtual machines, we will start an on-line monitoring and detection procedure in the hypervisor. The details of the detection procedure are shown in Figure 15. At the beginning of each round of detection, we need to take a snapshot of the memory pages of a virtual machine that contain its execution states. This operation will take a very short period of time and will not impact the user experiences with the guest virtual machine [57]. After the system data structures are reconstructed from the memory pages, we will compare the information to the knowledge that we have learned through Step 2. If any anomaly is detected, we will raise an alarm.

4.4 Implementation and Experimental Results

4.4.1 Experiment Environment Setup

To evaluate the detection capabilities of the proposed approach and assess its impacts on the system performance, we implement the mechanisms in Xen and conduct two groups of experiments. In the first group, we investigate the effectiveness of our malware detection approach through a group of real-world malware. In the second group, we evaluate the impacts of our detection mechanism on the guest system performance. The experiment environment setup is as follows. The physical machine has a four-core 3.30GHz Intel CPU, 10 GB RAM, and SATA hard drives. The hypervisor that we use is Xen version 4.1.2 with the libvir 0.9.8. The host operating system is Ubuntu Server 12.04 LTS (64bit). We test two virtual machines. One virtual machine is using Windows XP SP3 (32bit) as the operating system. The other virtual machine is using Windows 7 Professional (32bit) as the operating system. Each virtual

machine occupies one CPU core and 20GB hard disk. We vary the allocated memory to virtual machines from 1GB to 3GB to evaluate the impacts of the memory size on our proposed approach.

4.4.2 Detection Capability of the Proposed Approach

To collect behavior patterns of the benign systems and software, we install popular benign applications in the guest virtual machines. We download 100 benign and freely available applications from a trustworthy and reputable website. These benign applications cover freeware programs in a wide range of different domains (such as system utilities, office applications, media players, instant messaging, and browsers). We experiment with different combinations of the software in different environments so that the analysis phase can extract their special properties. After that, we download and install six groups of different malware to evaluate the detection capabilities of our program. We conduct malware detection experiments on two types of virtual machines. In the first group, the guest virtual machine runs Windows 7 Professional 32 bit. Our malware collection consist of 75 real-world malware samples, including 10 stealth backdoors, 20 trojans, 15 adwares, 10 worms, 10 rootkits, and 10 virus. All of them are publicly available on the Internet (e.g., from websites such as http://oc.gtisc.gatech.edu:8080/).

Figure 16 summarizes the detection results. Here 'False Negative' represents the number of malware that is missed by our detection mechanism. 'DLL Path' represents the number of malware that is detected based on the anomaly in name/loading path of the DLL files. 'Loading Order' represents the number of malware that is detected

Category	Total	False Negative	DLL Path	Loading Order	RVA
Backdoor	10	0	8	6	8
Trojan	20	1	18	17	18
Adware	15	2	13	13	13
Worm	10	0	9	10	10
Rootkit	10	0	10	10	10
Virus	10	0	10	8	10

Figure 16: Summary of DLL examination malware detection results on Windows 7 VM.

based on the order in which the DLL files are loaded. 'RVA' represents the number of the anomaly in hash results of the relative virtual addresses. For example, 17 out of 20 Trojan attacks are detected by our approach since they change the order in which DLL files are loaded into the system.

From Figure 16, we can see that our proposed mechanism is able to correctly identify most of the malware samples. There are three false negatives in our detection results. One of them is a Trojan that attempts to redirect our browser to another website. This Trojan is a JavaScript Trojan that does not have any DLL related behaviors. The other two are JavaScript adware. They attempt to display pop-up and pop-under advertisements when we are visiting some website in a JavaScript-enabled browser. The advertisements pop-up as separate windows to the active browser window so that they can bring additional profit to the designer. None of the missed malware demonstrates abnormal behaviors of a process or tries to infect DLL files. Therefore, they are not detected by our approach.

In the second group, the guest virtual machine runs Windows XP SP3 32 bit. From Figure 17, we can see that our proposed mechanism is able to correctly identify most

of the malware samples. There are two false negatives in the detection results. One of them is a Trojan that changes a registry value. The computer is also showing an advertisement in the Yahoo Messenger chat window. Hence, its behavior resembles a benign application. If we click on that advertisement, it would download and execute a setup file that will run at every system boot-up. Our malware detection mechanism will catch it if this behavior shows up. The other false negative is an adware, which is a download manager. Every time a user wants to download a file from the internet, a window with the advertisement will appear. However, no user data will be reused, stored, or shared. The reason that it is missed is because our malware detection mechanism can identify only the abnormal behaviors of a process, but not its intent or phishing. In real life, it is not difficult for a user to identify this type of advertisement.

Category	Total	False Negative	DLL Path	Loading Order	RVA
Backdoor	5	0	3	3	3
Trojan	10	1	9	8	9
Adware	5	1	4	4	4
Worm	5	0	4	5	5
Rootkit	3	0	3	3	3
Virus	4	0	4	3	4

Figure 17: Summary of DLL examination malware detection results on Windows XP VM.

We have conducted a third group of experiments to assess the false positive mistakes of our approach. We download 100 benign and freely available applications that are different from our training set. These applications cover a wide range of different domains (such as browsers, audio players, video players, instant messaging,

and security applications). From Figure 18, we can see that our malware detection mechanism causes no false positive mistakes.

4.4.3 Overhead and Performance Analysis

To protect a virtual machine from malware infection, we need to run the proposed approach at the hypervisor level periodically. Since our detection mechanism needs to temporarily freeze a part of the memory in the virtual machine, it will impact the operations of the guest virtual machine. Therefore, we must study the relationship between the detection frequency and its impacts on the system performance. We conduct two sets of experiments to assess the impacts.

In the first group of experiments, the guest virtual machine is running CPU intensive applications. We choose two examples: (1) the *Fibonacci* benchmark that computes the Fibonacci sequence; and (2) the *Prime* benchmark that generates prime numbers. Each of the software is running in parallel with the malware detection mechanism. When they are running in a virtual machine, the measured CPU usage is very close to 100%. The malware detection algorithm is running in the hypervisor. We measure changes in execution time of the software since this is the most intuitive

	Total	False Positive	Browser	Audio	Video	Office	IM
Windows 7	50	0	10	10	10	10	10
Windows XP	50	0	10	10	10	10	10
	Total	**False Positive**	**Utilities**	**Graphic**	**Education**	**DVD/CD Tools**	**Security**
Windows 7	50	0	10	10	10	10	10
Windows XP	50	0	10	10	10	10	10

Figure 18: Tests for false positive mistakes of malware detection through DLL examination.

parameter that end users adopt to evaluate the system performance.

In the second group of experiments, the guest virtual machines are running CPU and memory intensive applications. We also choose two examples: (1) the *N-Queens* package that tries to generate all possible solutions to the *N-Queen* problem in chess; and (2) the *Combination* benchmark that computes all possible combinations of the input numbers and stores them in memory. We also measure their execution time when each of them is running in parallel with the proposed detection mechanism.

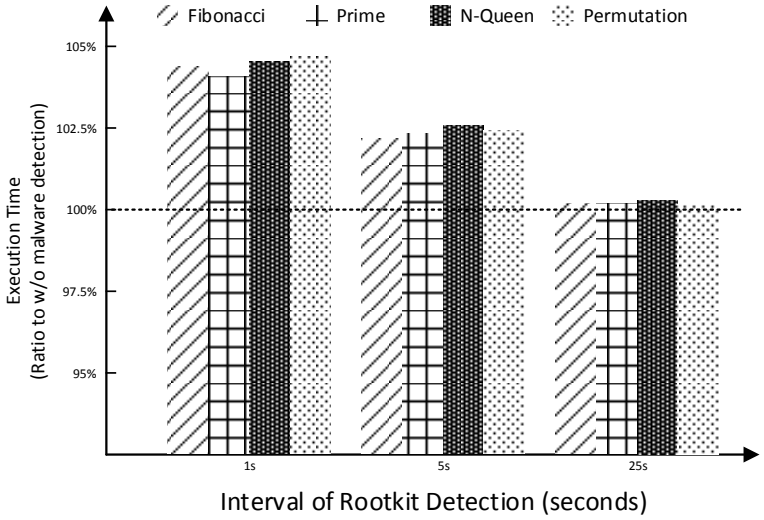


Figure 19: Relationship between DLL examination malware detection frequency and its impacts on system performance.

From Figure 19, we find out that when we increase the interval between malware detections, the impacts on virtual machine performance are decreasing. When the interval is equal to 25 seconds, there are almost no measurable increases in execution time at guest virtual machines. Even when we execute the proposed approach every second, the increase in application execution time is less than 5%. At the same time, since our approach does not need a lot of memory from the virtual machines, the

difference between the two groups of experiments is not large. Based on the results, we can see that our proposed approach has very low performance impacts on the virtual machine.

We also study the relationship between the allocated memory to guest virtual machines and the impacts of our malware detection mechanism on system performance. In this experiment, we use CPU and memory intensive applications to assess the performance impacts. In each guest virtual machine with different memory allocation, we run the *N-Queens* package while we test the malware detection mechanism with different execution intervals. Here we allocate 1GB, 2GB, and 3GB RAM to the virtual machine. From Figure 20, we can see that the system performance stays almost the same as long as the intervals between malware detections do not change. In our malware detection mechanism, we extract only high-level semantic information from the system data structures of the virtual machines. While malware detection is running at the hypervisor level, the differences in memory allocation size to guest virtual machines would not bring a huge difference to system performance since we do not conduct "brute force" memory scanning.

To reduce performance impacts, in the proposed mechanism we examine only the RVA of the DLL functions. A more comprehensive detection mechanism would examine the contents of all read-only sections of DLL files. To compare the two schemes, we conduct a group of experiment and measure their execution time. The execution time measures only the hashing and comparison delay but not the loading time of the files. From Figure 21, we can see that the execution time of hashing all read-only sections of DLL files is about 3 times longer than that of examining only RVAs. When

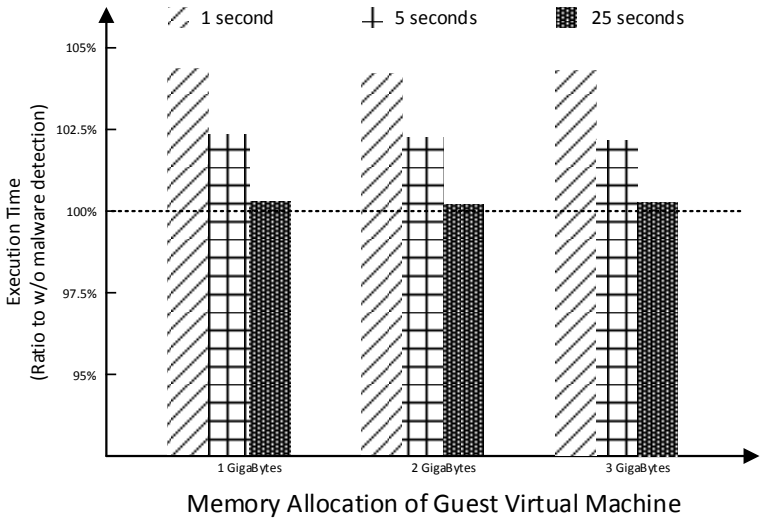


Figure 20: Relationship between memory allocation size and the impacts of DLL examination malware detection mechanism on VM performance.

the total size of DLL files is about $350MB$, the execution time of RVA examination is about $250ms$, while it takes about $850ms$ to examine the whole read-only contents.

4.5 Discussion

In this section, we will discuss a few potential extensions to our approach. We are especially interested in the tradeoff between detection capability and increases in overhead.

4.5.1 RVA only *vs* Read-only Content Examination for DLLs

To reduce the overhead of the proposed approach, in this chapter we examine only the virtual addresses of the functions in the DLLs. The assumption is that if attackers inject malicious code segments into a DLL, the relative address will change. This assumption may not hold under some cases. For example, an attacker may apply compression algorithms to in-line overwriting contents so that the size of the

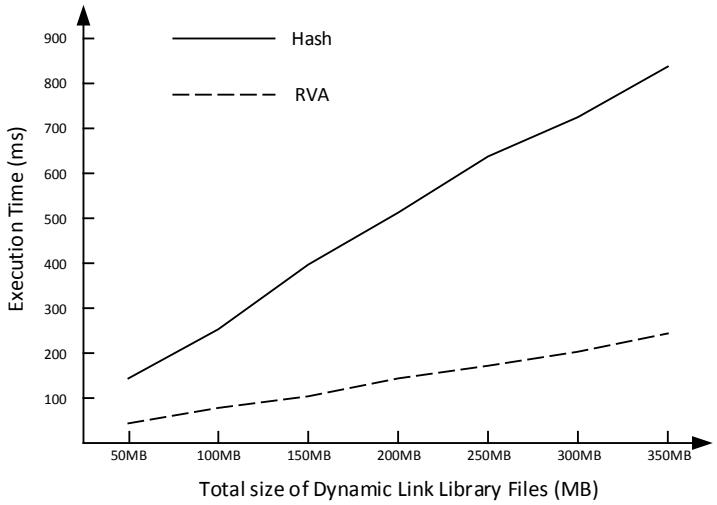


Figure 21: Comparison of hashing all read-only sections of DLL files with examining only RVAs.

malicious segments does not grow beyond the function boundaries [89].

To detect such attacks, we can use one of the following schemes. First, we could generate the hash result of the read-only contents of DLL files during the learning phase so that any small changes could be caught. Our experiment results in Figure 21 show that we will have to pay for the increases in computation and memory access delay. Another mechanism is to scan the DLL files and try to locate the code segment for decompression [111]. This method has the same memory access overhead but avoids computation.

4.5.2 Diversity and Order of Loaded DLLs

Researchers have proposed several mechanisms to use function call graphs to detect malware [55, 27]. In this chapter, we try to use information at a higher level of abstraction with the DLL names and their loading orders. One difficulty that we

face is the diversity of the DLL functions. Very frequently the same operation can be accomplished by functions from different DLL files. Another challenge that we face is the usage of DLL files that are not directly related to the functionality of the application software. For example, an increasing number of applications will collect user information, encrypt it, and send it back to the company server so that user profiling can be conducted. Such operations also increase the difficulty of malware detection.

In this chapter, we investigate the order in which the DLL files are loaded into the system and use it for malware detection. Here we do not differentiate tight dependency from loose dependency (several DLLs may switch their order of loading without impacting the software functionality). The lack of such knowledge may lead to false positive alarms. In the next step, we plan to investigate this problem and classify their dependency.

4.6 Conclusion

In this chapter, we propose a lightweight malware detection mechanism for virtual machines. The hypervisor will collect, analyze, and monitor the execution states of virtual machines and detect compromised DLL files. In the experiments, we have evaluated more than 100 real-world malware samples. We use both Windows XP and Windows 7 as test operation systems. Our experiment results show that the proposed approach is practical and effective. Furthermore, we conduct several groups of experiments to evaluate the increased overhead under different situations. The increases in overhead at virtual machines are very low since we access only a small

portion of their memory pages through high-level data structures.

Immediate extensions to our approach consist of the following aspects. First, we plan to experiment our approach with Linux virtual machines so that we can evaluate its practicability in other environments. Second, we plan to design innovative mechanisms to extract more high-level information from virtual machines. The rich data set will allow us to better understand the difference between benign and malicious software. Finally, we plan to extend our approach to other hypervisors (such as KVM) so that more end users can benefit from our research.

CHAPTER 5: MALWARE DETECTION BASED ON EXAMINATION OF SYSTEM CALL IN VIRTUAL MACHINES

5.1 Introduction

With the rapid development of cloud computing, enterprises transform themselves to cloud in order to provide better services to customers, and private users move their data to the cloud because cloud computing makes their life easier. According to 2016 review data from Synergy Research Group, cloud services revenues reached 148 billion dollars, and have grown by 25%[34]. While users move their service and data to cloud, they have concerns about the location of their data, confidentiality, and availability. Security is still the top concern, while they are using cloud computing technology [70].

Malware, which gains access to private computer, gathers sensitive information, or displays unwanted advertising, is widely spread through Internet, and infects unnumbered computer systems. Malware refers a variety of malicious programs (viruses, worms, Trojan horses, rootkits, botnets, backdoors.). In Symantec's Internet security threat report [103], the number of new malware variants detected by it is 431 million in 2015, which increased 36 percent from 2014. Furthermore, over half a billion personal records were stolen in 2015 [103].

Malware detection has been studied for many years, and a lot of different detection approaches have been proposed and implemented. Based on the differences in analysis

methods, existing approaches to malware detection can be classified into two groups. The first group is using static analysis technology to analyze disassembled malware binary files [31, 92, 52]. Applying static analysis technology to detect malware can achieve high efficiency, but malware could thwart the detection easily through code obfuscation techniques (packing, encryption, dummy function, etc.) [74, 124, 18, 80]. In the second group, dynamic analysis methods extract and analyze the behaviors that malware performs when malware is activated. Compared with static analysis detection methods, dynamic analysis detection could achieve more effective detection, because dynamic analysis could extract more features and behaviors than static analysis.

In dynamic analysis malware detection, different researchers focus on different features and behaviors, for examples, processes information, network information, PE(Portable Executable) files, DLL(Dynamic Link Library), API(Application Program Interface) call, function call, and system call [6, 84, 26]. In [114, 49], researchers extract and analyze processes information, so that they will detect hidden malware based on comparing processes information. In [13], researchers proposed a real-time PE malware detection mechanism, based on the analysis results of the information stored in PE file. In [79, 115], authors present effective and efficient heuristic techniques to detect malware, using DLL dependency tree or DLL order. In [1, 127], they use lightweight classification techniques to detect malware at API level.

Host-based malware detection mechanisms are very popular in dynamic analysis malware detection. However, they have their limitations. On one side, since the anti-malware systems are installed and executed inside the hosts that they are monitoring,

they can collect rich information from the local host. On the other side, since they are visible and tangible to advanced malware running in the host, effective attacks towards them become feasible. For example, some malware such as Agobot variant [129] can detect and remove more than 105 types of anti-malware programs in the victim machine.

In a cloud computing environment, hypervisor-based malware detection mechanisms provide a new method to detect malware, which could be stealthy and accurate to detect malware running within virtual machines. Some researchers [57, 114, 32, 116, 51] proposed to place the intrusion detection mechanisms outside of the virtual machine being monitored. Since well-implemented hypervisor will enforce strong isolation between virtual machines and the programs running within a hypervisor, it is difficult for an attacker to detect or remove anti-malware program running in a hypervisor. For example, VMwatcher [49] uses the general virtual machine introspection (VMI) methodology in a non-intrusive manner to inspect the low-level virtual machine states. UCON (usage control model) [118] is an event-based logic model. It maintains the lowest level access to the system and ensures that such access cannot be compromised by internal processes of a virtual machine.

Existing malware detection mechanisms that are running on hypervisor usually work in the following procedure. The mechanisms will first extract execution information of processes inside a virtual machine, and they will send the information to the hypervisor. They will also extract the execution information of virtual machines at the hypervisor. Through analyzing the information sent from virtual machine and information extracted at hypervisor, anti-malware programs will find abnormal pro-

cesses. Example solutions include [57, 114, 32, 51, 49]. These approaches are effective in detection of infected files when malware has process level abnormal behaviors. Furthermore, if malware with new behaviors or malware with normal process level behaviors show up, they cannot capture this kind of malware.

In this chapter, we propose a system call examination approach at hypervisor level to continuously monitor the sequence of the system call in guest virtual machines to detect malware. Instead of using high-level information that is extracted from virtual machines, we extract and monitor system call sequences at the hypervisor, which present process's behaviors clearly. Through analyzing extracted system call sequences, we could figure out hardware-related behaviors, creating and execution of new processes behaviors, network communication behaviors, and registry behaviors. Overall, our approach can be divided into three procedures: system call behaviors extraction procedure, system call sequences analysis procedure, and abnormal behaviors monitoring procedure. Through memory reconstruction technology of the virtual machines, we extract the system calls of the virtual machine at the hypervisor level. Using freshly installed virtual machines, our system call behaviors extraction procedure will extract and record information of system call sequences in guest virtual machines, while we are executing malware samples. After we record enough system call sequences from the training malware dataset, we will start the analysis component. We will compare the system call sequences among the training malware dataset, and extract the pattern of system call sequences while they have similar process behaviors. We will exclude the same patterns as benign processes to reduce false positive rate. After analysis procedure, we will continuously monitor the system

call sequences in the virtual machine, and compare them to our extracted features. If malware is executed in the virtual machine by attacker or victim, we can detect this intrusion in real time.

The contributions of our research can be summarized as follows. First, instead of examining the high-level information of process in a virtual machine, our approach conducts continuous monitoring at the system call level in a virtual machine. Through this monitoring, infection to a virtual machine can be detected once it starts. Second, we use non-intrusive introspection of virtual machines in our malware detection mechanism. Since the malware detection mechanism is running at the hypervisor level, it is very difficult for malware running within a virtual machine to detect, remove, or avoid our detection mechanism. Third, we conduct extensive experiments to evaluate our malware detection mechanism. We use more than 300 malware of different types (Trojans, stealth backdoors, and virus) to evaluate our proposed approach. Our malware detection mechanism detects almost all malware samples with very low false negatives. Last but not least, although our malware detection environment is currently implemented on Windows and Xen, the key methods could also be applied to other operation systems and hypervisors. Experiments results on real-world malware show that our proposed malware detection mechanism could achieve high detection rate at an acceptable performance overhead.

The remainder of this chapter is organized as follows. In Section 5.2 we present the details of the proposed approach. In Section 5.3 we describe the implementation of the malware detection mechanism and experiment results. Section 5.4 discusses a few issues in the detection procedure. Finally, Section 5.5 concludes the chapter.

5.2 The Proposed Approach

In this section, we will present the details of the proposed approaches. We first discuss the assumptions of the environments to which our approaches can be applied. We also introduce the technique of system function call extraction which we use to extract system function calls of the virtual machine. Finally, we present the details of the approaches and mechanisms to detect malware.

5.2.1 System Assumptions

In the investigated scenario, we assume that we can get access to malware-free virtual machines while we are collecting system call sequences of the benign program. We also assume that we can get access to virtual machines which contain only the malware with our permission, while we are collecting system call sequences of malware. We can achieve this through using freshly installed systems. After the collection procedure is finished, we do not require any restricted behaviors in the virtual machines. Users may be tricked by attackers, and download or install some worm, rootkit, virus, adware, or spyware into the system. We also assume that an attacker has acquired the administrator privilege in the compromised virtual machine after she/he intrudes into the system. Malware installed by the attacker may modify return results to anti-virus programs in the virtual machine so that it could hide the malicious process. The attacker may leave a backdoor in the system for subsequent attacks. The malware may be injected in some benign process, such as explorer.exe, and start several threads to conduct malicious activities. However, similar to the approaches in [125], we assume that the attacker cannot infect the hypervisor through

the virtual machine because of the virtual machines isolation.

Our investigation has the following design goals. First, the malware detection mechanism should be transparent to end users. Moreover, it can extract and recover the virtual machine's execution states accurately. This goal is realized through the non-intrusive introspection technology. Since we examine the virtual machine execution states from the hypervisor, it is very difficult for the malware to mislead the detection procedure. Second, the approach should be independent of specific hypervisors. The design should support Virtual Machine Manager (VMM) in both full virtualization mode (e.g., VMware[110] and KVM [56]) and paravirtualization (e.g., XEN [12]) modes. This property allows more users to benefit from the approach. The analysis presented in later parts will show that our malware detection mechanism does not depend on any specific types of the hypervisor. Third, we also need to control the performance impacts of the proposed approach on virtual machines under monitoring. This requirement is essential for future deployment and adoption of our malware detection mechanisms. Our experiments will show that examining only the execution states in virtual machines memory, even periodically, will introduce an acceptable increase in overhead.

5.2.2 System Function Call Extraction

In order to implement malware detection based on examination of system call in virtual machines, we must first apply memory reconstruction technology to reconstruct memory of virtual machines at the hypervisor level, because the hypervisor only sees the raw memory pages of virtual machines without any semantic view. Af-

ter we rebuild a semantic view of virtual machines at hypervisor level, we can extract execution information of virtual machines.

To better explain the memory reconstruction procedure, we use an example of a Windows guest OS to illustrate the execution information extraction operations. In the beginning, we will use Rekall forensics tool[22] to extract the debug data provided by Microsoft so that we could build a map of internal system call functions. We also extract the kernel symbol table, which indicates symbol names and their addresses in memory. We can go through the process list from $PsActiveProcessHead$ (the head of the double linked list). In Windows, each process has an $\_EPROCESS$ object which we can use to traverse the whole list. Each $\_EPROCESS$ object contains both $Flink$ (forward link) that points to the next $\_EPROCESS$ structure and $Blink$ (backward link) that points to the previous $\_EPROCESS$ structure. $PsActiveProcessHead$ is a member of the kernel debugger data block ($\_KDDEBUGGER\_DATA64$), which is used by the kernel debugger to easily find out the states of the operating system [66]. Furthermore, the Kernel Processor Control Region (KPCR) is a data structure used by the Windows kernel to store information about each process. It is located at virtual address $0xffdff000$ in Windows. In KPCR, the data structure $KdVersionBlock$ contains a linked list of $\_KDDEBUGGER\_DATA64$. Since we have extracted the Relative Virtual Address(RVA) of all kernel symbols, we could do the calculation to calculate the kernel base address. Then, we will conduct a binary planning injection so that we can directly trap internal system function call.

Through memory introspection technology, we can extract high-level execution states of virtual machines. The execution states that we can get include process list,

network connections, opened files, dynamic-link library, relative virtual address of functions, and system function call. In our malware detection mechanism, we use system function call as the execution information of virtual machines.

5.2.3 Design of the Malware Detection Mechanisms

At the high level, we propose a malware detection mechanism that runs in the hypervisor to detect malware in guest virtual machines base on system call analysis. This is accomplished in three phases (Fig.22): (1) the information collection phase, in which a process collects information about different applications among virtual machines with and without malware; (2) the offline analysis phase, in which we analyze the system call sequences of each malware and benign process, and extract the system call patterns of each malware and benign application; and (3) a runtime detection phase, in which the detection program is used to detect malware in a guest virtual machine through comparing its system call sequences to the learned system call patterns. These three steps are described in more details below.

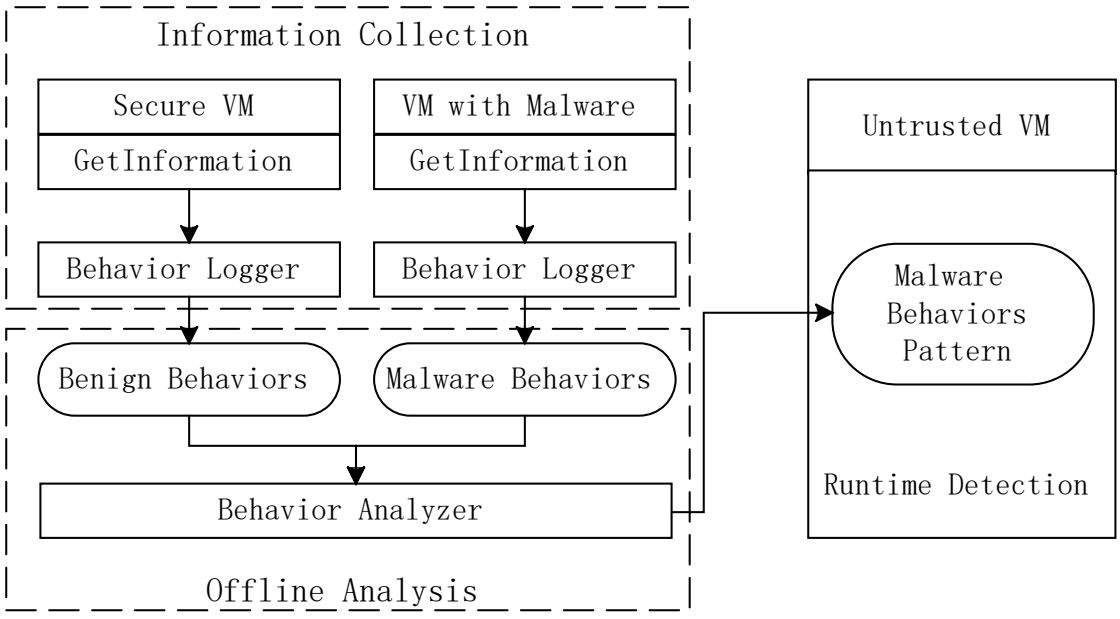


Figure 22: Malware detection procedure through system call function examination.

In order to get a comprehensive view of the execution states and avoid impacts from a specific running environment, our information collection program will run many times in the hypervisor to extract behaviors of each process from different installations of malware-free virtual machines. We collect system call sequences from two different types processes: benign processes and malware. We will exclude the same system call pattern of benign processes from malware's system call pattern, so that we will reduce the false positive rate. As we discussed in Section 5.2.2, we conduct a binary planting injection to directly trap system function calls. Since there are more than 11,000 different kinds of system calls in Windows, virtual machines will be really slow if we plan to trap all of the system function calls. Even though we can detect malware more accurately if we can extract more execution information of processes, we should consider the performance of a virtual machine which is also important to a customer. According to their different features, we classify all of the system calls into different categories. **Communication**: Communication related system calls. It includes create, delete communication connection; send, receive messages; listened port and any others. **Registry Access**: Registry related system calls. It includes create, delete, query, write registry key, etc. **Process Control**: Process related system calls. It includes create, terminate process; get, set process attributes; thread management; etc. **Memory management**: Memory management related system calls. It includes memory allocation and destroy, **Device Management**: Device related system calls. It includes open, read, write, and load operations; get device attributes; etc. **System Information Management**: System management system calls. It includes query, set system information. We select system calls as trap targets which will change the

execution states of virtual machines or will access significant system information. For example, $OpenProcess$, $OpenThread$, $LoadDriver$, $QuerySystemInformation$, and so on. After this selection, the trap targets set still contains a large group of system calls. We will conduct a static analysis to select system calls according to their performance impact. During the static analysis, we will exclude the system calls which will be called frequently by every process so that we will achieve acceptable performance overhead, for example, $GetCurrentStackPointer$. We use a running example to illustrate the information collection procedure. For example, we have three sample records:

$$Sequence_1 = \{Call_1, Call_2, Call_3, Call_4, Call_1\}$$

$$Sequence_2 = \{Call_2, Call_3, Call_2, Call_5, Call_4\}$$

$$Sequence_3 = \{Call_3, Call_2, Call_6, Call_1, Call_7, Call_3\}$$

An array $SClist$ stores system calls that all programs have called including maliciours and benign programs.

$$SClist = \{Call_1, Call_2, Call_3, Call_4, Call_5, Call_6, Call_7\}$$

We count the frequency of each system calls in each program.

$$Count_1 = \{Call_1 : 2, Call_2 : 1, Call_3 : 1, Call_4 : 1\}$$

$$Count_2 = \{Call_2 : 2, Call_3 : 1, Call_3 : 1, Call_4 : 1, Call_5 : 1\}$$

$$Count_3 = \{Call_1 : 1, Call_2 : 1, Call_3 : 2, Call_6 : 1, Call_7 : 1\}$$

According to the percentage of each system calls in all program, we will exclude the

system calls which are called frequently, because they have the significant impact on performance.

In the offline analysis, we classify the system call sequences based on the training dataset. In the same training data set, malware belong to the same malware family, which have similar abnormal activities. For example, they use the same method to conduct Dynamic Link Library Injection, or they apply the same technology to hide their existence. We will extract the pattern of system call sequences which are in the same group. According to combination theory, if we have a sequence with $lengh = n$, there are

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

different kinds of subsequences. In the same group, the total number of system call sequences is m. No matter what optimized methods we will apply, time complexity or space complexity would be exponential. We can imagine the size of time complexity and the size of space complexity when n is greater than 10^3. Furthermore, the main purpose of offline analysis is to extract the pattern of system call sequences, instead of detecting malware accurately. Because of these reasons, we consider extracting the common continuous system call sequences as the pattern of behaviors. Even though some malware add some dummy system calls, it will not change the result, because malware in the same group have similar behaviors. Through suffix arrays [69], we could extract the common continuous system call sequences in linear time-complexity $O(n)$, where n is the length of sequences. We will exclude the same patterns of benign programs from the patterns of malware, so that we will reduce false positive.

Pseudo Code:

```
Initialize Extracted Patterns[N][L].
Initialized Length[N].
Initialize Matched[N] with 0 for each process.

Update(call, pid)
    get Matched[N] according pid
    for i from 0 to N
        if Patterns[i][Matched[i]] == call
            ++Matched[i]
        if Matched[i] == Length[i]
            raise alarm: process pid has Pattern[i] behavior

Runtime_Detect()
    for every system call, pid
        Update(call, pid)
```

Figure 23: Runtime detection pseudo code through system call function examination.

In the runtime detection procedure, we use a two-dimensional array to store extracted patterns of malware behaviors. N is the number of extracted patterns of system call sequences, while we have an array to store its length of each sequence. $Length[i]$ is the length of $Patterns[i]$. For each process, we have a array $Matched$ to record the matched system call sequences. $Matched[i]$ is the position of next expected system call in $Patterns[i]$, which means $Patterns[i][Matched[i]]$ is the expected system call. For a coming system call, we will get the $Matched$ array based on its *pid*. Then, we will update $Matched$ based on the comparison between expected system call ($Patterns[i][Matched[i]]$) and coming system call (*call*). Since the length of system call sequence M is much larger than the number of extracted patterns N, $M \gg N$, we

can consider the time complexity is $O(MN) \ll O(M^2)$. If we find the extracted system call pattern is matched in the system call sequence of some processes, we will raise an alarm that we detect some abnormal behaviors of these processes. For example, we extract the pattern $(OpenFile, CreateFile, CreateHandle, DestroyHandle, DeleteFile)$ repeating multiple times from our learned system call sequences. $Rootkit.Sirefef.Gen$ will conduct DLL injection and has the same system call behaviors. In system call sequences of benign program, it will have the following system call sequences to query a registry and set it with a proper value $(OpenKey, QueryValueKey, SetValueKey)$. However, some malware will use the following system call sequences to open a registry, retrieve data in the key, and alter it $(OpenKey, EnumerateKey, OpenKey, SetValueKey)$.

5.3 Implementation and Experimental Results

5.3.1 Experiment Environment Setup

To evaluate the detection capabilities of the proposed approach and assess its impacts on the system performance, we implement the detection mechanisms in Xen and conduct two groups of experiments. In the first group, we investigate the effectiveness of our malware detection approach through a group of real-world malware. In the second group, we evaluate the impacts of our detection mechanism on the guest system performance. The experiment environment setup is as follows. The physical machine has a four-core 3.30GHz Intel CPU, 10 GB RAM, and SATA hard drives. The hypervisor that we use is Xen version 4.1.2 with the libvir 0.9.8. The host operating system is Ubuntu Server 12.04 LTS (64bit). The virtual machine is using Windows 7 Professional (32bit) as the operating system. Each virtual machine

occupies one CPU core, 2GB RAM, and 20GB hard disk.

5.3.2 Detection Capability of the Proposed Approach

To collect system function call sequences patterns of benign systems and software, we install popular benign applications in the guest virtual machines. We download 40 benign and freely available applications from a trustworthy and reputable website. These benign applications are freeware programs in a wide range of different domains (such as system utilities, office applications, media players, instant messaging, and browsers). We experiment with different combinations of the software in different environments so that the analysis phase can extract their special properties. After that, we download and install six groups of different malware to evaluate the detection capabilities of our program. Our malware collection consist of 300 real-world malware samples, including 80 stealth backdoors, 40 trojans, 40 adwares, 20 worms, 80 rootkits, and 40 virus. All of them are publicly available on the Internet (e.g., from websites such as https://virusshare.com/, and http://openmalware.org/). We classified them into different groups according to their behaviors. For example, malware which hide their processes are in the same group, malware which have DLL injection are in the same group. In offline analysis phase, we analyzed and extracted the pattern of system function call sequences in the same group.

Fig.24 summarizes the detection results. Here 'Total' represents the number of different malware samples. 'Training Set' represents the number of extracted system function call sequences. We use 150 malware samples as training malware set in the information collection phase. Every malware sample was activated for 5 times, so

that we have total 750 different extracted system function call sequences. The reason is that all of the extracted system call sequences are different, when we activate the same malware for several times. In our malware detection mechanism, we mainly focus on system function call sequences. 'Validation Set' represents the number of malware that we used to verify our detection mechanisms. 'False Negative' represents the number of malware that was missed by our detection mechanism, while 'True Positive' represents the number of malware that we detected. The value represents the number of malware that is detected since our on-line monitoring program detects that the system function call sequences of the virtual machine are matched with the knowledge that we learn from malware. For example, 39 out of 40 Rootkit attacks are detected by our approach since they have the same system function call sequences as hiding their existence.

From Figure 24, we can see that our proposed mechanism is able to correctly identify most of the malware samples. There are three false negatives in our detection results. One of them is a rootkit, which shows up as a global clock. The computer is also showing an advertisement. Hence, its behavior resembles a benign application

Category	Total	Sequence learned	Validation Set	False Negative	True Positive
Backdoor	80	200	40	0	40
Trojan	40	100	20	1	19
Adware	40	100	20	1	19
Worm	20	50	10	0	10
Rootkit	80	200	40	1	39
Virus	40	100	20	0	20

Figure 24: Summary of detection results against malware through system call sequences examination.

in our malware detection mechanism. If we click on that advertisement, it would jump to some website. The second false negative is an adware, which is a download manager. Every time a user wants to download a file from the Internet, a window with advertisements will appear. The third one is a trojan. It cannot be executed alone. However, no user data will be reused, stored, or shared. The reason that they are missed is because our malware detection mechanism can identify only the abnormal behaviors of a process, but not its intent or phishing. In real life, it is not difficult for a user to identify this type of advertisement.

	Total	False Positive	Browser	Audio	Video	Office	IM
Windows 7	15	0	3	3	3	3	3
	Total	**False Positive**	**Utilities**	**Graphic**	**Education**	**DVD/CD Tools**	**Security**
Windows 7	15	0	3	3	3	3	3

Figure 25: Tests for false positive mistakes of malware detection through system call sequences examination.

We have conducted a second group of experiments to assess the false positive mistakes of our approach. We download 30 benign and freely available applications that are different from our training set. These applications cover a wide range of different domains (such as browsers, audio players, video players, instant messaging, and security applications). From Figure 25, we can see that our malware detection mechanism causes no false positive mistakes.

In the third group, we conduct experiments to evaluate the relationship between the number of different extracted system function calls and the detection capability. We exclude the extracted system function calls randomly, and evaluate the detection

capability. From Figure 26, we can see that the detection accuracy increases while the number of different extracted system function call increases. When the total number of different extracted system function call is 50, the detection accuracy is low, because the patterns of system function call sequences are deeply impacted by it. When the total number is 200, the detection accuracy is about 90%.

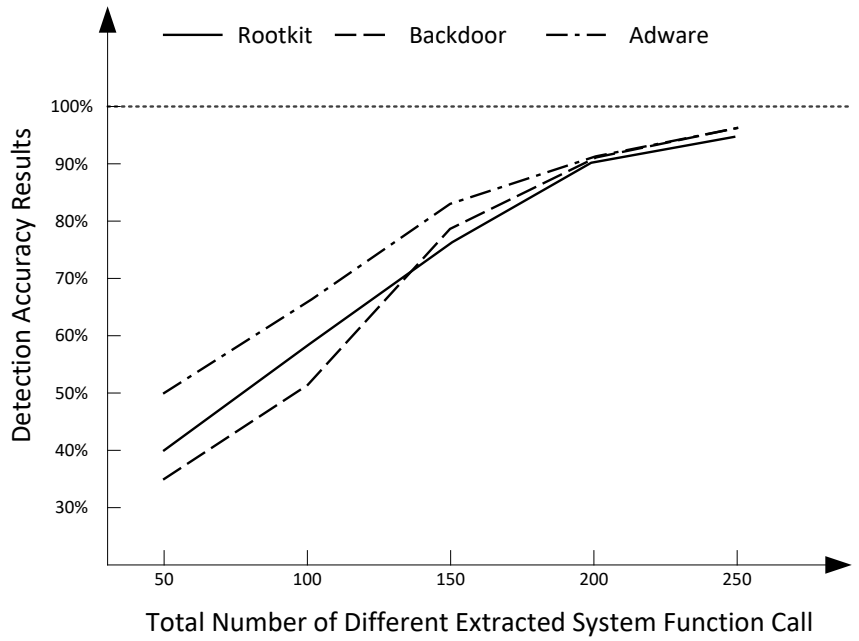


Figure 26: Relationship between the number of different extracted system calls and the detection capability.

5.3.3 Overhead and Performance Analysis

To protect a virtual machine from malware infection, we need to execute the binary planting injection to trap internal system function calls at the hypervisor level. Since every time the trapped system function calls is executed, it will impact the operations of the guest virtual machine, we must study the relationship between the number of the trapped system call and its impacts on the system performance. We conduct two

sets of experiments to assess the impacts.

In the first group of experiments, the guest virtual machine is running CPU intensive applications. We choose two examples: (1) the *Fibonacci* benchmark that computes the Fibonacci sequence; and (2) the *Prime* benchmark that generates prime numbers. Each of the software is running in parallel with the malware detection algorithm. When they are running in a virtual machine, the measured CPU usage is very close to 100%. The malware detection algorithm is running in the hypervisor. We measure changes in execution time of the software since this is the most intuitive parameter that end users adopt to evaluate the system performance.

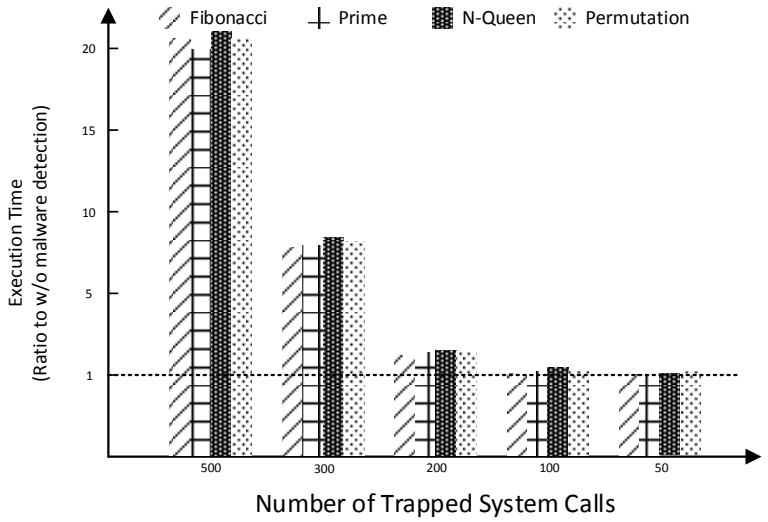


Figure 27: Relationship between the number of trapped system calls and its impacts on system performance.

In the second group, the guest virtual machines are running CPU and memory intensive applications. We also choose two examples: (1) the $N-Queens$ package that tries to generate all possible solutions to the $N-Queen$ problem in chess; and (2) the *Combination* benchmark that computes all possible combinations of the input

numbers and stores them in memory. We also measure their execution time when each of them is running in parallel with the proposed detection mechanism.

From Fig.27, we find that when we decrease the number of trapped system calls, the impacts on virtual machines performance are decreasing. When the number of trapped system calls is about 500, the performance impacts to the guest virtual machine is huge. The execution times of applications with malware detection are more than 20 times of the execution time of applications without malware detection. However, when the number of trapped system calls is about 100, there are almost no measurable increases in execution time at guest virtual machines. As we discussed in Section 5.3.2, during our runtime detection phase, the number of trapped system calls is about 200, when we conduct the experiments of detection capability. At the same time, since our approach does not need a lot of memory from the virtual machine, the difference between the two groups of experiments is not large. Based on the results, we can see that our proposed approach introduces an acceptable performance impacts to the guest virtual machine system.

5.4 Discussion

In this section, we will discuss a few potential extensions to our approach. We are especially interested in the tradeoff between detection capability and increases in overhead. To reduce the overhead of the proposed approach, in this paper we examine only the selected system function calls. The assumption is that malware will have similar behaviors within the same family or malware will use the same technology to hack computers even though they are the new versions. If attackers

improve their malware with the system function calls that are not in our selected data set, malware will avoid our proposed malware detection. To detect such attacks, we should increase our data set of selected system function calls, so that it will include more system function calls that could be used by the attacker.

Furthermore, researchers have proposed several mechanisms to use function call graphs to detect malware [55, 27]. In this chapter, we treat system function call in sequences, not graphs. Graphs could indicate more execution information of processes, and they may lead to improved detection accuracy. We would try to build graphs of the system function calls of processes. One difficulty that we face is that the information extraction technology at the hypervisor isn't mature enough to extract the information which we need to build the system function call graph. In the next step, we plan to investigate this problem and try to improve the extraction technology.

5.5 Conclusion

In this chapter, we propose a malware detection mechanism based on examination of system call sequence for virtual machines. The hypervisor will extract, analyze, and monitor the system call sequences of virtual machines and detect running malware. In the experiments, we have evaluated 300 real-world malware samples. Our experiment results show that the proposed approach is practical and effective. Furthermore, we conduct several experiments to evaluate the increased overhead based on different situations. The increases in overhead at virtual machines are acceptable since we only monitor selected system calls.

Immediate extensions to our approach consist of the following aspects. First, we plan to experiment our approach with Linux virtual machines so that we can evaluate its practicability in other environments. Second, we plan to improve our system function call extraction so that we could extract the information of virtual machines with low-performance overhead. Finally, we plan to extend our approach to other hypervisors (such as KVM) so that more end users can benefit from our research.

CHAPTER 6: CONCLUSION

In this dissertation, we proposed some security improvement mechanisms in cloud computing environment. More specifically, through memory analysis, we not only proposed some service level agreements violation detection mechanisms, which customers could use to verify the execution of service level agreements other than trusting the words of the cloud provider, but also proposed some malware detection mechanisms at hypervisor, which could detect malware running within virtual machine, We have focused on:

1. Detection of service level agreements violation in the guest virtual machine.

2. Rootkit detection on guest virtual machines through cross-verified extraction information at hypervisor-level.

3. Lightweight examination of DLL environments in virtual machines to detect malware.

4. Malware detection based on examination of the system call in virtual machines.

To detect service level agreements violation in the guest virtual machine, we proposed violation detection mechanisms based on memory access latency. Instead of proposing a generic security SLA enforcement architecture, we designed and implemented the security SLA violation detection approaches under VMware and tested them. In our mechanisms, we could detect 3 kinds of service level agreements violations, unauthorized memory accesses violation, deduplication policies violation,

and memory under-allocation violations. We also analyzed the impacts on system performance of our security SLA violation detection approach. We evaluated the performance of guest virtual machines under the extreme case. The experiments' result showed that our detection algorithm would bring a small increase in overhead. Immediate extensions to our approaches consist of the following aspects. More other types of security SLA violations in memory management and a generic approach for their detection will be designed. More other hypervisors, such as extended Xen and Linux KSM, could be considered to be implemented with security SLA violation detection.

We have proposed a new rootkit detection mechanism for virtual machines through cross-verified extraction information at hypervisor-level. Using memory reconstruction technology, we rebuild the semantic view of the virtual machine's memory. Then, through cross-verification different components of the reconstructed view, the hypervisor can detect hidden information and mismatch among different active modules in the virtual machine. Since we build the semantic view of the memory, the performance overhead is really small that customers using virtual machines could hardly feel it. This defense could be further extended by introducing intelligence into the construction and anomaly detection in the linkage table among different modules of the virtual machines.

We have also proposed a lightweight malware detection mechanism for virtual machines. The hypervisor will collect, analyze, and monitor the execution states of virtual machines and detect compromised DLL files. Instead of examining the DLL files for only once when they are loaded, we conduct continuous monitoring of the files, so that the infection to the DLL files can be detected on the fly. It is very

difficult for malware running within a virtual machine to detect, remove, or avoid our detection at hypervisor. This defense could be further extended through innovative mechanisms to extract more information from virtual machines.

Finally, we used binary planning injection to trap internal system function calls. Through analysis system call sequences of malware, we extract the pattern of system call sequences while malware has abnormal behaviors. We have proposed an examination of system call sequences of virtual machines so that we could detect malware at hypervisor. We also investigate the relationship between the number of system calls that we extract and the performance overhead of virtual machines. This approach could be improved through reducing the performance overhead of virtual machines.

Cloud computing security, is the most important topic that attracts more and more researchers to this field, because of the rapid development of cloud computing. No only enterprises transform themselves to cloud to provide better services, but also private users move their data to the cloud because cloud computing makes their lives easier. Every step of the development of cloud computing, new security problems will arise which must be solved, Otherwise, they will involve fatal issues. After my graduation, the research on how to better improve the security of cloud computing environment which were addressed in this dissertation will continue. Both myself and others at UNC Charlotte will continue to improving cloud computing security environment, so that cloud computing providers and end customers can have a securer cloud computing experience.

REFERENCES

[1] Y. Aafer, W. Du, and H. Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*, pages 86–103. Springer, 2013.

[2] O. Agmon Ben-Yehuda, E. Posener, M. Ben-Yehuda, A. Schuster, and A. Mu'alem. Ginseng: Market-driven memory allocation. In *ACM SIGPLAN Notices*, volume 49, pages 41–52. ACM, 2014.

[3] I. Ahmed, A. Zoranic, S. Javaid, and G. Richard. Modchecker: Kernel module integrity checking in the cloud environment. In *International Conference on Parallel Processing Workshops (ICPPW)*, pages 306–313, 2012.

[4] A. Alasiri, M. Alzaidi, D. Lindskog, P. Zavarsky, R. Ruhl, and S. Alassmi. Comparative analysis of operational malware dynamic link library (dll) injection live response vs. memory image. In *International Conference on Computing, Communication System and Informatics Management (ICCCSIM)*, 2012.

[5] M. Ali, S. U. Khan, and A. V. Vasilakos. Security in cloud computing: Opportunities and challenges. *Information Sciences*, 305:357–383, 2015.

[6] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in computer virology*, 7(4):247–258, 2011.

[7] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Linux Symposium*, pages 19–28, 2009.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[9] F. Baiardi and D. Sgandurra. Building trustworthy intrusion detection through vm introspection. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on*, pages 209–214. IEEE, 2007.

[10] M. Baig, C. Fitzsimons, S. Balasubramanian, R. Sion, and D. Porter. Cloudflow: Cloud-wide policy enforcement using fast vm introspection. In *IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–164, 2014.

[11] I. Banerjee, F. Guo, K. Tati, and R. Venkatasubramanian. Memory overcommitment in the esx server. Technical report, VMWare Inc., VMWare Technical Journal, 2013.

[12] 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. In *Proceedings of ACM SOSP*, pages 164–177, 2003.

[13] M. Belaoued and S. Mazouzi. A real-time pe-malware detection system based on chi-square test and pe-file features. In *IFIP International Conference on Computer Science and its Applications_x000D_*, pages 416–425. Springer, 2015.

[14] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, N. Tawbi, et al. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001(184-189):79, 2001.

[15] S. J. Berman, L. Kesterson Townes, A. Marshall, and R. Srivathsa. How cloud computing enables process and business model innovation. *Strategy & Leadership*, 40(4):27–35, 2012.

[16] K. Bernsmed, M. Jaatun, P. Meland, and A. Undheim. Security slas for federated cloud services. In *Sixth International Conference on Availability, Reliability and Security (ARES)*, pages 202–209, Aug 2011.

[17] B. Blunden. *The Rootkit arsenal: Escape and evasion in the dark corners of the system*. Jones & Bartlett Publishers, 2012.

[18] J.-M. Borello and L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[19] I. Brandic, V. C. Emeakaroha, M. Maurer, S. Dustdar, S. Acs, A. Kertesz, and G. Kecskemeti. Laysi: A layered approach for sla-violation propagation in self-manageable cloud infrastructures. In *Proceedings of the IEEE Annual Computer Software and Applications Conference Workshops*, pages 365–370, 2010.

[20] V. Casola, A. Mazzeo, N. Mazzocca, and M. Rak. A sla evaluation methodology in service oriented architectures. In D. Gollmann, F. Massacci, and A. Yautsiukhin, editors, *Quality of Protection*, volume 23 of *Advances in Information Security*, pages 119–130. Springer US, 2006.

[21] P. M. Chen and B. D. Noble. When virtual is better than real. In *Hot Topics in Operating Systems VIII*, 2001.

[22] M. Cohen. Rekall memory forensics framework. *DFIR Prague*, 2014.

[23] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour. Establishing and monitoring slas in complex service based systems. In *IEEE International Conference on Web Services (ICWS)*, pages 783–790, 2009.

[24] A. V. Dastjerdi, S. G. H. Tabatabaei, and R. Buyya. A dependency-aware ontology-based approach for deploying service level agreement monitoring services in cloud. *Softw. Pract. Exper.*, 42(4):501–518, Apr. 2012.

[25] S. de Chaves, C. Westphall, and F. Lamin. Sla perspective in security management for cloud computing. In *International Conference on Networking and Services (ICNS)*, pages 212–217, March 2010.

[26] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[27] A. A. E. Elhadi, M. A. Maarof, and A. H. Osman. Malware detection based on hybrid signature behavior application programming interface call graph. *American Journal of Applied Sciences*, 9(3):283–288, 2012.

[28] V. Emeakaroha, I. Brandic, M. Maurer, and S. Dustdar. Low level metrics to high level slas - lom2his framework: Bridging the gap between monitored metrics and sla parameters in cloud environments. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 48–54, 2010.

[29] V. Emeakaroha, T. Ferreto, M. Netto, I. Brandic, and C. De Rose. Casvid: Application level monitoring for sla violation detection in clouds. In *IEEE Annual Computer Software and Applications Conference (COMPSAC)*, pages 499–508, 2012.

[30] V. C. Emeakaroha, M. A. S. Netto, R. N. Calheiros, I. Brandic, R. Buyya, and C. A. F. De Rose. Towards autonomic detection of sla violations in cloud infrastructures. *Future Gener. Comput. Syst.*, 28(7):1017–1029, July 2012.

[31] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[32] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[33] A. Gordon, M. R. Hines, D. S. da, M. Ben-Yehuda, M. Silva, and G. Lizarraga. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *RESoLVE: Runtime Environments/Systems, Layering, and Virtualized Environments Workshop (co-located with ASPLOS)*, 2011.

[34] S. R. Group. Cloud market growing at 25 percent annually in 2016, 2017.

[35] D. Gullasch, E. Bangerter, and S. Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 490–505. IEEE, 2011.

[36] F. Guo, S. Kim, Y. Baskakov, and I. Banerjee. Proactively breaking large pages to improve memory overcommitment performance in vmware esxi. In *ACM SIGPLAN Notices*, volume 50, pages 39–51. ACM, 2015.

[37] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, 2010.

[38] A. Haeberlen. A case for the accountable cloud. *SIGOPS Oper. Syst. Rev.*, 44(2):52–57, April 2010.

[39] I. U. Haq, I. Brandic, and E. Schikuta. Sla validation in layered cloud infrastructures. In *Proceedings of the 7th International Conference on Economics of Grids, Clouds, Systems, and Services*, pages 153–164, 2010.

[40] R. R. Henning. Security service level agreements: Quantifiable security for the enterprise? In *Proceedings of the Workshop on New Security Paradigms*, pages 54–60, 1999.

[41] J. Heo, X. Zhu, P. Padala, and Z. Wang. Memory overbooking and dynamic control of xen virtual machines in consolidated environments. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 630–637, 2009.

[42] M. Hines, A. Gordon, M. Silva, D. D. Silva, K. D. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with ginkgo. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 130–137, 2011.

[43] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with osck. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 279–290. ACM, 2011.

[44] T. Hwang, Y. Shin, K. Son, and H. Park. Design of a hypervisor-based rootkit detection method for virtualized systems in cloud computing environments. In *AASRI Winter International Conference on Engineering and Technology (AASRI-WIET)*, pages 27–32. Citeseer, 2013.

[45] M. S. Inci, B. Gülmezoglu, G. I. Apecechea, T. Eisenbarth, and B. Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[46] W. Iqbal, M. Dailey, and D. Carrera. Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud. In *IEEE International Conference on Cloud Computing*, pages 243–253. Springer, 2009.

[47] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes. In *IEEE Symposium on Security and Privacy (SP)*, pages 591–604, 2015.

[48] M. Jakobsson and Z. Ramzan. Evolution of rootkits. In *Crimeware: Understanding New Attacks and Defenses*. Addison-Wesley, 2008.

[49] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based 'out-of-the-box' semantic view reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 128–138, 2007.

[50] S. Jin, J. Ahn, S. Cha, and J. Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 272–283, 2011.

[51] A. Joshi, S. King, G. Dunplap, and P. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, 2005.

[52] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim. Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, 2015.

[53] S. Katopodis and G. Spanoudakis. Towards hybrid cloud service certification models. In *IEEE International Conference on Services Computing*, 2014.

[54] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE communications surveys & tutorials*, 16(2):898–924, 2014.

[55] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4):233–245, 2011.

[56] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the linux virtual machine monitor. In *Linux Symposium*, pages 225–230, 2007.

[57] P. F. Klemperer. *Efficient Hypervisor Based Malware Detection*. PhD thesis, CMU, May 2015.

[58] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Annual Computer Security Applications Conference*, 2004.

[59] U. Lampe, M. Kieselmann, A. Miede, S. Zöller, and R. Steinmetz. A tale of millis and nanos: Time measurements in virtual and physical machines. In K.-K. Lau, W. Lamersdorf, and E. Pimentel, editors, *Service-Oriented and Cloud Computing*, volume 8135 of *Lecture Notes in Computer Science*, pages 172–179. Springer Berlin Heidelberg, 2013.

[60] A. Lanzi, M. Sharif, and W. Lee. K-tracer: A system for extracting kernel malware behavior. In *NDSS*, 2009.

[61] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *Euromicro Conference, 2004. Proceedings. 30th*, pages 520–525. IEEE, 2004.

[62] M. Laureano, C. Maziero, and E. Jamhour. Protecting host-based intrusion detectors through virtual machines. *Computer Networks*, 51(5):1275–1283, 2007.

[63] C. Lee, C.-H. Hong, S. Yoo, and C. Yoo. Compressed and shared swap to extend available memory in virtualized consumer electronics. *IEEE Transactions on Consumer Electronics*, 60(4):628–635, Nov 2014.

[64] C. Li, A. Raghunathan, and N. K. Jha. Secure virtual machine execution under an untrusted management os. In *Proceedings of the IEEE International Conference on Cloud Computing*, pages 172–179, 2010.

[65] M. Li, W. Zang, K. Bai, M. Yu, and P. Liu. Mycloud – supporting user-configured privacy protection in cloud computing. In *Annual Computer Security Applications Conference*, 2013.

[66] M. H. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, 2014.

[67] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 605–622. IEEE, 2015.

[68] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. *SIGOPS Oper. Syst. Rev.*, 41(3):327–340, 2007.

[69] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[70] I. E. MARKETING. 2016 idg enterprise cloud computing survey, 2016.

[71] McAfee. Rootkits, part 1 of 3, the growing threat, 2006.

[72] S. Mei, Z. Wang, Y. Cheng, J. Rena, J. Wu, and J. Zhou. Trusted bytecode virtual machine module: A novel method for dynamic remote attestation in cloud computing. *International Journal of Computational Intelligence Systems*, 5(5):924–932, 2012.

[73] A. B. M. Moniruzzaman. Analysis of memory ballooning technique for dynamic memory management of virtual machines (vms). arxiv.org document 1411.7344, 2014.

[74] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[75] Y. Mundada, A. Ramachandran, and N. Feamster. Silverline: Data and network isolation for cloud services. In *USENIX Workshop on Hot Topics in Cloud Computing*, 2011.

[76] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot: a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[77] N. L. Petroni Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *ACM CCS*, pages 103–115, 2007.

[78] M. Narouei. Mining modules dependencies for malware detection, 2012.

[79] M. Narouei, M. Ahmadi, G. Giacinto, H. Takabi, and A. Sami. Dllminer: structural mining for malware detection. *Security and Communication Networks*, 8(18):3311–3322, 2015.

[80] P. O'Kane, S. Sezer, and K. McLaughlin. Obfuscation: The hidden malware. *IEEE Security & Privacy*, 9(5):41–47, 2011.

[81] R. Owens and W. Wang. Fingerprinting large data sets through memory deduplication technique in virtual machines. In *IEEE Military Communications Conference (MILCOM)*, 2011.

[82] R. Owens and W. Wang. Non-interactive os fingerprinting through memory deduplication technique in virtual machines. In *IEEE International Performance Computing and Communications Conference (IPCCC)*, 2011.

[83] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. Soules, G. R. Goodson, and G. R. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *USENIX Security*, 2003.

[84] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[85] Y. Rahulamathavan, P. S. Pawar, P. Burnap, M. Rajarajan, O. F. Rana, and G. Spanoudakis. Analysing security requirements in cloud-based service level agreements. In *Proceedings of the International Conference on Security of Information and Networks*, pages 73–76, 2014.

[86] H. Raj, D. Robinson, T. B. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted computing for guest vms with a commodity hypervisor. Technical report, Microsoft Research, MSR-TR-2011-130, Redmond, WA, USA, 2011.

[87] R. Riley, X. Jiang, and D. Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 47–60. ACM, 2009.

[88] 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 (CCS)*, pages 199–212, 2009.

[89] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.

[90] A. Salah, M. Shouman, and H. Faheem. Surviving cyber warfare with a hybrid multiagent-base intrusion prevention system. *IEEE Potentials*, 29(1):32–40, 2010.

[91] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1020–1025, 2010.

[92] A.-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE, 2009.

[93] M. Schmidt, L. Baumgartner, P. Graubner, D. Bock, and B. Freisleben. Malware detection and kernel rootkit prevention in cloud computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 603–610. IEEE, 2011.

[94] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 335–350. ACM, 2007.

[95] M. Shafiq, S. Tabish, F. Mirza, and M. Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In E. Kirda, S. Jha, and D. Balzarotti, editors, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 121–141. Springer Berlin Heidelberg, 2009.

[96] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang. Detecting malware variants via function-call graph similarity. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 113–120. IEEE, 2010.

[97] P. Sharma and P. Kulkarni. Singleton: system-wide page deduplication in virtual environments. In *Proceedings of international symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 15–26, 2012.

[98] J. Shi, X. Song, H. Chen, and B. Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, pages 194–199, 2011.

[99] A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In S. Jha, R. Sommer, and C. Kreibich, editors, *Recent Advances in Intrusion Detection*, volume 6307 of *Lecture Notes in Computer Science*, pages 97–117. 2010.

[100] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory deduplication as a threat to the guest os. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC'11, pages 1:1–1:6, New York, NY, USA, 2011. ACM.

[101] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Software side channel attack on memory deduplication. *SOSP POSTER*, 2011.

[102] Symantec. An analysis of address space layout randomization on windows vista, 2007.

[103] Symantec. 2016 internet security threat report, 2016.

[104] J. Szefer and R. B. Lee. A case for hardware protection of guest vms from compromised hypervisors in cloud computing. In *Proceedings of the International Conference on Distributed Computing Systems Workshops*, pages 248–252, 2011.

[105] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, 2010.

[106] VMWare. Esxi configuration guide. VMware vSphere 4.1 Documentation, 2010.

[107] VMWare. Understanding memory resource management in vmware esx 4.1. VMware ESX 4.1 Documentation, EN-000411-00, 2010.

[108] VMware. Timekeeping in vmware virtual machines, 2011.

[109] VMWare. Configure the virtual machine communication interface in the vsphere web client. in vSphere Virtual Machine Administration Guide for ESXi 5.X, 2012.

[110] B. Walters. Vmware virtual platform. *Linux J.*, (63es), 1999.

[111] T.-E. Wei, Z.-W. Chen, C.-W. Tien, J.-S. Wu, H.-M. Lee, and A. Jeng. Repef: A system for restoring packed executable file for malware analysis. In *International Conference on Machine Learning and Cybernetics (ICMLC)*, volume 2, pages 519–527, 2011.

[112] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

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

[114] X. Xie and W. Wang. Rootkit detection on virtual machines through deep information extraction at hypervisor level. In *International Workshop on Security and Privacy in Cloud computing (SPCC), in conjunction with IEEE CNS*, 2013.

[115] X. Xie and W. Wang. Lightweight examination of dll environments in virtual machines to detect malware. In *Proceedings of the 4th ACM International Workshop on Security in Cloud Computing*, pages 10–16. ACM, 2016.

[116] X. Xie, W. Wang, and T. Qin. Detection of service level agreement (sla) violation in memory management in virtual machines. In *Computer Communication and Networks (ICCCN), 2015 24th International Conference on*, pages 1–8. IEEE, 2015.

[117] X. Xiong, D. Tian, P. Liu, et al. Practical protection of kernel integrity for commodity os from untrusted extensions. In *NDSS*, volume 11, 2011.

[118] M. Xu, X. Jiang, R. Sandhu, and X. Zhang. Towards a vmm-based usage control framework for os kernel integrity protection. In *SACMAT*, pages 71–80, 2007.

[119] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of ACM workshop on Cloud computing security workshop*, pages 29–40, 2011.

[120] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu. Combining file content and file relations for cloud based malware detection. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 222–230. ACM, 2011.

[121] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang. An intelligent pe-malware detection system based on association mining. *Journal in Computer Virology*, 4(4):323–334, 2008.

[122] H. Yin, Z. Liang, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. In *NDSS*, 2008.

[123] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Krida. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM CCS*, pages 116–127, 2007.

[124] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297–300. IEEE, 2010.

[125] A. Yu, Y. Qin, and D. Wang. Obtaining the integrity of your virtual machine in the cloud. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 213–222, 2011.

[126] L. Zhang, S. Shetty, P. Liu, and J. Jing. Rootkitdet: Practical end-to-end defense against kernel rootkits in a cloud environment. In *European Symposium on Research in Computer Security*, pages 475–493. Springer, 2014.

[127] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.

[128] 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 ACM conference on Computer and communications security*, pages 305–316, 2012.

[129] Z. Zhu, G. Lu, Y. Chen, Z. J. Fu, P. Roberts, and K. Han. Botnet research survey. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 967–972. IEEE, 2008.