Security Implications of Memory Deduplication in a Virtualized Environment

Jidong Xiao*, Zhang Xu*, Hai Huang†, Haining Wang*

*The College of William and Mary, Williamsburg, Virginia, USA
†IBM T. J. Watson Research Center, Hawthorne, New York, USA

***Abstract*— Memory deduplication has been widely used in various commodity hypervisors. By merging identical memory contents, it allows more virtual machines to run concurrently on top of a hypervisor. However, while this technique improves memory efficiency, it has a large impact on system security. In particular, memory deduplication is usually implemented using a variant of copy-on-write techniques, for which, writing to a shared page would incur a longer access time than those non-shared. In this paper, we investigate the security implication of memory deduplication from the perspectives of both attackers and defenders. On one hand, using the artifact above, we demonstrate two new attacks to create a covert channel and detect virtualization, respectively. On the other hand, we also show that memory deduplication can be leveraged to safeguard Linux kernel integrity.**

I. Introduction

Memory deduplication is a technique implemented in various commercial and open source hypervisors, including VMWare ESX, Xen, and Linux KVM. The key idea is, if multiple memory pages have the same content, then the hypervisor only needs to keep one copy for these so-called deduplicated pages. And later, if one of the deduplicated pages is modified, a copy-on-write (COW) technique is used. In other words, the page will be copied, and the write operation will take effect on the copied page. This allows a significant amount of memory space to be saved when there are many identical pages.

However, this technique can also expose unexpected security vulnerabilities. Due to the extra copy operation, a write to a deduplicated page and a normal page (non-deduplicated page) will incur different access times. Therefore, in a virtualized environment where both an attacker's virtual machine (VM) and a victim's VM might co-exist on the same host machine, the attacker can obtain memory information from its neighboring VM [1], [2]. For example, the attacker can detect whether a given page is located in the memory of a neighboring VM by loading the same page into its own memory, waiting for some time until the memory deduplication takes effect, and then writing to that page, i.e., the one that has been loaded into its own memory. If the page is deduplicated, writing to it would take longer than writing to a normal page. Consequently, the attacker is able to know whether the page also exists in a neighboring VM, resulting in information leakage.

In this paper, exploiting this artifact of memory deduplication, we demonstrate two new attacks to create a covert channel in a virtualized environment and detect virtualization, respectively. Meanwhile, we observe that memory deduplication can also be used for defense purposes. More specifically, we leverage memory deduplication to detect the existence of kernel rootkits in a timely manner with negligible overhead. The major contributions of this work are summarized as follows:

- From an attacker's perspective, we develop a reliable covert channel to transfer information between two virtual machines. We validate the feasibility and effectiveness of using this covert channel for information leakage. Our experimental results show that the new covert channel can reach nearly 100% accuracy, and even in a system under high computation and memory pressure, it can still achieve a reasonable transfer rate. Furthermore, following the same approach, we also present a virtualization detection technique that does not depend on any specific instruction or guest OS, and it is applicable to detect various virtual machine monitors such as VMWare ESX server, Xen, and Linux KVM.
- From a defender's perspective, we propose a simple and novel technique to ensure kernel runtime integrity. Our technique, as a virtualization-based defense solution, works in almost real time, and more importantly, it does not require any understanding of the target OS kernel. Thus, it mitigates the "*semantic gap*" issue that commonly exists on most virtual machine introspection tools [3], [4], [5]. Simply monitoring a memory statistic provided by Linux kernel, it can detect any modification against the read-only data of Linux kernel. Our experimental results show that the defense technique is able to effectively detect those rootkits that modify the system call table.

We run our experiments on top of Linux/KVM with Kernel Samepage Merging (KSM) implemented as a loadable

kernel module[1]. KSM is originally designed to combine identical virtual machine memory pages into one copy, but now it is able to merge any identical application memory pages. Although our experiments are performed on Linux/KVM with KSM, we believe that the security implications being identified would also apply to other mainstream commercial hypervisors, including VMWare ESX server, which has implemented a content-based memory sharing scheme [6], and Xen, which has incorporated memory page sharing as an important feature in Xen 4.0 [7]. In this paper, we use the term memory deduplication and page sharing interchangeably.

The remainder of the paper is organized as follows. Section II details the design of covert channel and virtualization detection. Section III describes our solution to detect kernel read-only data modification. Section IV presents our experiment results. Section V discusses some related issues. Section VI surveys related work. Finally, Section VII concludes the paper.

II. Offense Techniques

In this section, we first describe the construction of the covert channel in a virtualized environment, and then we present how to detect virtualization based on memory deduplication.

A. Covert Channel Construction

To achieve high resource utilization, public cloud providers such as RackSpace and Amazon EC2 often co-locate users and have them share resources. Inevitably, virtual machines belonging to different users are often placed on the same physical machine with users having very little control over the placement decisions. As Ristenpart et al. [8] have demonstrated, such physical co-location gives attackers a vantage point to perform malicious activities, such as covert channel attacks.

1) Working Mechanism: Preventing unauthorized disclosure of information is one of the major tasks in computer security. While it might be relatively easy to prevent information disclosure in open channels by using encryption, it is hard to detect and eliminate covert channels. Covert channels were first defined by Lampson in 1973 as a communication channel designed for information leakage [9]. Typically, a covert channel allows two isolated entities to communicate with each other. It consists of a sender and a receiver. In a cloud environment, both entities are virtual machines running on the same physical machine. The sender (of information) is generally the victim of the attack, and we assume that the sender has been compromised by the attacker, therefore, it is under the attacker's control. To bypass traditional monitoring techniques and remain hidden as long as possible, the sender encodes the information the attacker is interested in and injects "signals" onto a covert channel. The receiver is a virtual machine launched by the attacker to be a co-resident with the sender VM. The receiver will probe the covert channel and decode the "signals". In this way, the information on the victim machine can be leaked while minimizing the chance of the attacker being disclosed. Existing works have exploited various shared hardware resources to build a covert channel in virtualization environments, such as L2 cache [10]. However, the existing techniques are not robust against environment noises.

Here we develop a covert channel based on memory deduplication, which can achieve strong resistance to environment noises and still ensure a high bit rate. Figure 1 illustrates the framework of our covert channel. In the first step, the sender and receiver load a certain amount of memory with identical content. This can be easily done by having both the sender and receiver opening and reading the same file. Next, the sender encodes the information, e.g., writing to certain pages so that the contents of these pages are different from those launched by the receiver. Once the pages are modified, the sender and receiver sleep and wait for the system to merge these pages. Finally, the receiver should write to all memory pages used by the covert channel and record the write access time. The copy-on-write mechanism makes writing to shared pages cost more time than those that are not. Thus, the receiver can easily identify those modified pages. With the knowledge of which pages are modified, the receiver can decode the information. This is basically how the sender and receiver communicate via the memory deduplication covert channel.

The encoding mechanism is further detailed as follows. Since we can detect memory deduplication at the granularity of a page, we make each page represent one bit of information. At the sender side, an unmodified page indicates a 0 and a modified page denotes a 1. For instance, as shown in Figure 1, if we want to transmit 001011 through the covert channel, after the sender and receiver both read six identical pages, the sender should modify the 3rd, 5th, and 6th pages. After sleeping for a period of time, the receiver will write to these six pages and record access time. Since pages 1, 2, and 4 remain unchanged, the memory deduplication mechanism should have merged them with their counterparts. By contrast, pages 3, 5 and 6 have been modified by the sender, and hence they are not deduplicated. Thus, the receiver can find that it takes much more time (according to our observation, at least six times more) to write to pages 1, 2, and 4 than to pages 3, 5, and 6. At the receiver side, a long access time indicates a 0 and a short access time denotes a 1. Therefore, in our example the receiver can easily infer that the sender is sending 001011.

2) Design Space: To make the covert channel practical and maximize the transfer rate, we need to address the following two questions: (1) how to identify an optimal waiting time for memory deduplication mechanism to kick

[1]Linux has implemented kernel samepage merging(KSM) since kernel 2.6.32.

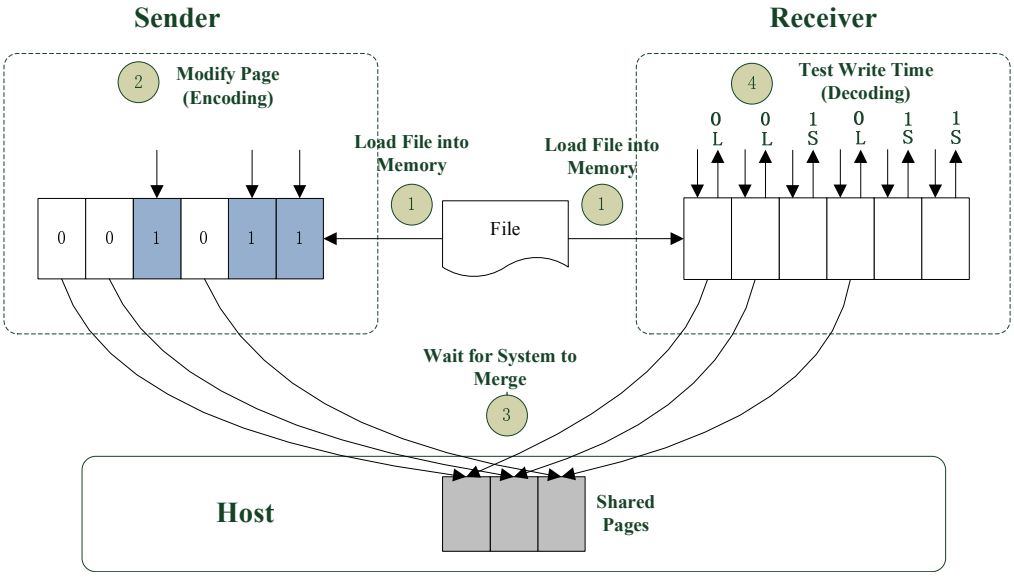

Figure 1. Covert Channel Overview

in? and (2) how to select an appropriate memory size to build the covert channel?

When the sender and receiver load contents into memory, the system does not immediately merge identical pages because it is usually done as a background task. In our covert channel, we run two C programs on the sender side and the receiver side, respectively. After page loading and information encoding, the programs on both sides should sleep for a while so that the memory deduplication mechanism has sufficient time to complete its work. The length of this sleeping time depends on the current load of the system as well as the memory size used for building the covert channel. Obviously, the heavier the load, the more slowly memory deduplication would complete.

Meanwhile, increasing the size of the memory used to build the covert channel has dual effect. On one hand, a larger piece of memory can carry more information, i.e., the covert channel can transfer more information during one transmission. On the other hand, more memory usage implies that KSM has more memory pages to scan and merge, which can prolong the required time to complete one transmission. Moreover, if the used memory is too large, the reliability of the covert channel starts to degrade and the covert channel could be easily detected. Note that the bit rate of the covert channel is the ratio between the information of being delivered during one transmission and the time to complete the transmission. The impact of the memory size on the bit rate is detailed in Section 4.1.

One advantage of our covert channel is that it is robust against environment noises. In contrast, the existing L2 cache based covert channels are sensitive to environment noises. When other users access the cache, the contents in the cache might be replaced. Thus, the protocol between the sender and the receiver is interfered. Even worse, some hypervisors (e.g., Xen) have the mechanism of core migration. In other words, VMs will be periodically migrated to different physical processors, which leads to a complete flush of the L2 cache and a failure of the covert channel. On the contrary, our covert channel will not be "coincidentally" destroyed. Although theoretically it is possible that there are other pages in the system happen to be identical to ours, by properly designing the contents to be loaded into memory, we can ensure that the likelihood of such interference is astronomically small. Thus, only if we modify these pages, will the deduplication be canceled. Moreover, our channel is based on write access rather than read access. Compared with read access, write access is affected far less by cache mechanisms. Thus, our channel is even less sensitive to environment perturbations. Furthermore, before writing a page, we can always read it first, which can help us eliminate the influence of swapping.

B. Virtualization Detection

In recent years, a large number of virtualization-based defense mechanisms have been proposed. Virtualization is leveraged by various techniques to monitor, analyze, and thwart malicious attacks. Therefore, from an attacker's perspective, it is crucial to detect the use of virtualization in the target environment as it would allow the attacker to potentially circumvent such defense mechanisms by dynamically adapting the behavior of malicious code.

We found that the artifacts of memory deduplication can be used for this purpose. Here is an example illustrating how this can be done. First, we load a file into memory, write to all the pages, and record write access time (*t1*). Next, we load the same file into memory twice, in other words,

load the same file into two memory regions. This would result in every page of the file having an exact replica page in the memory. If this is a virtualized environment where memory deduplication works at the hypervisor level, the memory deduplication mechanism will take effect on these pages after some time, at which time, we can write to these deduplicated pages and record their access time (*t2*), and we should notice that *t2* is clearly larger than *t1* due to COW. By contrast, in a non-virtualized environment without memory deduplication enabled, there should not be any discernible differences between *t1* and *t2*.

III. Kernel Runtime Integrity Monitoring

Although we have demonstrated that the memory deduplication mechanism can be exploited by attackers, we found that it can also be used from a defender's stand point. In [2], the authors mentioned that page deduplication can be leveraged to detect illegal applications and files on virtual machines. Here we present one more scenario: using it to monitor kernel read-only data's integrity. Kernel read-only data refers to the data that are not supposed to be changed during kernel runtime, for example, system call table, which is the target of many existing kernel rootkits.

As virtualization technologies have become more mature, some defense mechanisms are shifted from the OS layer to the hypervisor layer. By deploying defense tools at the hypervisor level, we can detect rootkits that subvert the guest OS kernel. However, existing virtual machine monitor (VMM) based detectors have to cope with the semantic gap between the low level information and the high level OS abstractions. In other words, in order to monitor the guest OS, the detailed knowledge of the guest OS kernel is required. Over the past few years, although researchers have proposed different approaches to overcoming this semantic gap, the tools they have developed are still far from generic. As OS kernels are upgrading in a relatively high frequency, tools developed for one kernel version might not fit for another kernel version. This property severely inhibits the development and deployment of virtualization based security products. By contrast, the technique we are going to present is more generic, and is applicable to multiple, if not all, Linux kernel versions.

In modern Linux systems, when a kernel image is generated, read-only data are stored in the *.rodata* section of the kernel executable and linkable format (ELF) binary. Therefore, we can extract this section from a kernel image, which is the same as the one used by the running guest OS, and store the extracted content in a file. Then, we write a C program to load this file into memory, creating two copies of the read-only data stored in memory. One is loaded with the guest OS kernel, the other is loaded by the C program. Since these two copies are identical, we can expect their pages to be merged on a system where memory deduplication is being used. One might think that, once all the pages corresponding to the file are merged, we can write to each page of that file, and we record the write access time to each page, we can do the above "load ->wait ->write ->record" procedure periodically, once the write access time decreases dramatically, it implies the corresponding pages become non-deduplicated pages. Therefore, we know that the kernel integrity might be broken and thus the system might be compromised.

However, this solution can be easily bypassed by attackers. If attackers are aware of the existence of the above integrity checking system, they can modify the system call table or anything else inside the kernel's read-only data section, and at the same time, load an unmodified copy of the *.rodata* section into another memory region. In this way, even if the original *.rodata* section has been changed, very likely, the integrity checking system will fail to detect the rootkit, because the additional copy might have been merged with our reference copy before we write to each page. Consequently, the write access time will not change.

We propose a new technique to monitor kernel read-only data integrity. Unlike some existing techniques, our approach does not require any changes in the kernel or the hypervisor; and thus it is more practical to use. Our technique mainly relies on a particular memory statistic term (related to memory deduplication) maintained by the Linux kernel. It is called PSS, which stands for "proportional set size". The PSS of a process describes the number of pages the process has in memory weighted by how the pages are deduplicated. For instance, if a process has 100 unique pages and another 100 pages shared and deduplicated with another process, its PSS should be 150, i.e., 100 + 100/2. The Linux kernel exports this memory statistic in the file of `/proc/$pid/smaps`. Basically, this file is used to show memory consumption for each of the process's mappings. Here is an example:

00400000-00495000 r-xp 00000000 08:01 2097157 /bin/bash
Size: 596 kB
Rss: 496 kB
Pss: 38 kB
Shared_Clean: 496 kB
Shared_Dirty: 0 kB
Private_Clean: 0 kB
Private_Dirty: 0 kB
Referenced: 496 kB
Anonymous: 0 kB
Swap: 0 kB
KernelPageSize: 4 kB
MMUPageSize: 4 kB

We observe that by monitoring PSS value, we can detect kernel rootkits that modify kernel read-only data such as the

Table I
SYSTEM CONFIGURATION

Components	Specification
Host CPU	Intel Xeon 3.07GHz, Quad-Core
Host Memory	4GB
Host OS	openSuSE 11.4
Host Kernel	2.6.37.6-0.11-desktop x86_64
Qemu	0.14.0_rc1-1.4.2.x86_64
KVM	0.14.0.0-1.14.2.x86_64
Guest CPU	QEMU Virtual CPU version 0.14.0
Guest Memory	1GB
Guest OS	Fedora 16
Guest Kernel	3.1.0-7.fc16.i686

system call table.[2] The idea is, we load the aforementioned file (which just includes kernel read-only data section) into the host memory and wait briefly until memory deduplication takes effect on the pages corresponding to that file, and then we turn off KSM, thereby preventing further memory deduplication from happening. Thus, the PSS should be a fixed value and never be changed, unless the read only data section of the guest OS is changed, which would result in the shared pages become unshared. Therefore, by periodically monitoring the PSS value, we can infer whether or not the kernel read-only data is modified. To achieve this, just a simple shell script would suffice.[3]

Overall, our implementation just includes a simple C program and a simple shell script. The C program, being responsible for loading the *rodata* file into host memory, consists of less than one hundred lines of C code, and the shell script, being in charge of checking PSS periodically, is less than ten lines. Due to its simplicity, the overhead of our approach is negligible. For the same reason, we can expect an average system administrator being able to write a similar program to load the kernel *.rodata* section and a similar shell script to monitor the corresponding PSS, thus monitoring the kernel runtime integrity.

IV. EXPERIMENTAL EVALUATION

Our experiments can be divided into three parts, with respect to covert channel construction, virtualization detection, and kernel integrity monitoring, respectively. The system configuration of our test machine is listed in Table 1.

A. Covert Channel Construction

To evaluate the feasibility of the memory deduplication based covert channel, we first verify that such a covert channel does work well in a virtualized environment. We boot two virtual machines on the same hypervisor. One virtual machine is selected as the sender and the other is set as the receiver. We load a file of size 1088KB (i.e., 272 4KB pages) into memory. In order to ensure the reliability of deduplication, we set the sleeping time to 250 seconds. We set up four sets of experiments. In each experiment, the sender modifies different pages to transfer different messages to the receiver, and we record all the write access time to these 272 pages at the receiver side to see whether information has been correctly delivered.

Figure 2 shows the experimental results. In the first experiment, the sender wants to send a 272-bit data beginning with 00111 as the first 5 bits followed by all 0s. Between the sender and receiver, if a modified page denotes an 1 and an unmodified page denotes a 0, the sender would need to modify the 3rd, 4th and 5th pages to encode such data. For the other three experiments, the sender modifies pages 260, 261 and 262 in the second experiment, pages 205, 206 and 207 in the third experiment, and pages 69, 70 and 71 in the last experiment. From Figure 2, we can see although different experiments demonstrate different write access spikes, the sender-modified pages always incur much less write access time. This is because once the sender modifies a page, the page will become a non-deduplicated page, and hence less time is required for the following write access issued by the receiver. We also observe that, the time of writing to a deduplicated page is always at least 6 times longer than writing to a non-deduplicated page. This strong signal-to-noise ratio implies that the covert channel can be reliably established.

Next we evaluate the channel's robustness and transfer speed to see whether it is practical. As the covert channel is based on memory deduplication, it is important to figure out the relationship between the memory size used to build the covert channel and the bit rate we can achieve. The bit rate (R) is determined by the time needed to complete one transmission (T), and the volume (V) of information that can be delivered in one transmission: R=V/T. Moreover, since the transmission time is dominated by the sleeping time (i.e., the transmission time is almost equivalent to the sleeping time), we can use the sleeping time as the time required to complete one transmission without losing accuracy. In this set of experiments, we also set up two virtual machines as the sender and the receiver, respectively. In each round, the sender and the receiver load a file with a certain size into memory to build the channel. We gradually increase the sleeping time until it reaches a threshold such that the information can be transferred through the covert channel without any errors. We call such a threshold stable sleeping time. Then, we gradually increase the size of the loaded file and record the change of the stable sleeping time. After obtaining the stable sleeping time for each different memory size, we can calculate the bit rate using the formula of R=V/T.

[2]Note that this file is exported by the host OS, not by the guest OS, and therefore, attackers who might have compromised the guest OS cannot directly control or modify this file.

[3]Ideally, we can detect the modifications on the read only data section regardless of turning off KSM or not, however, in reality, if KSM keeps running, there might be some false positives; for example, it is possible that some pages just happen to be the same as a read only data page, therefore, they will be merged, and this might result in PSS change. In contrast, by turning off KSM at this point, we can fully eliminate false positives.

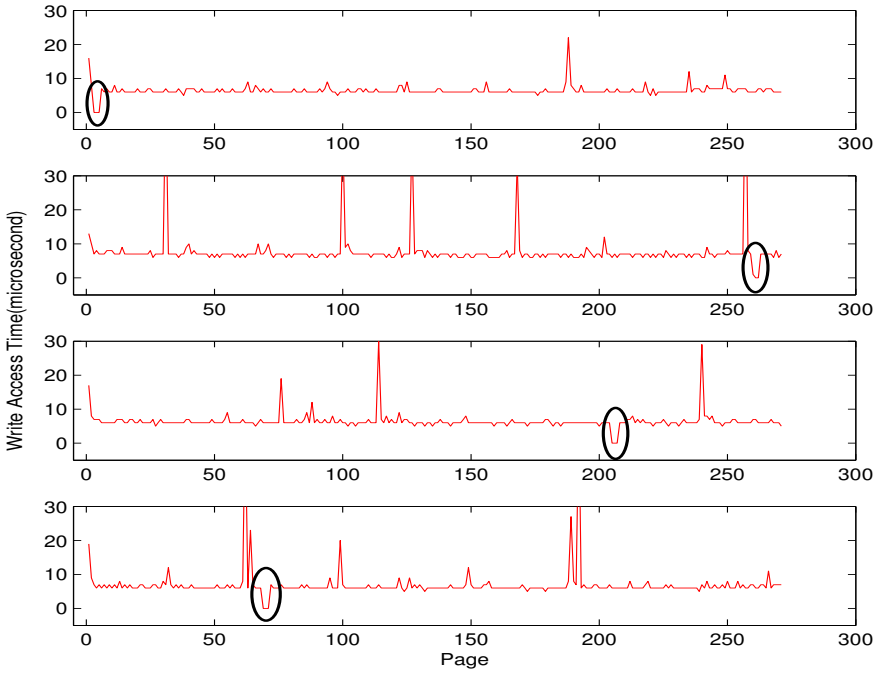

Figure 2. Information Transferring in Covert Channel

Figure 3 illustrates the dynamics of achieved bit rate with the change of memory size, in which the stable sleeping time in each case is also marked. When the memory size increases from 1MB to 100MB, we can easily observe that the bit rate significantly increases but the stable sleeping time only modestly increases. When the memory size is 1MB, or 256 4K pages, it takes our channel 205s to complete the transmission. The bit rate under this case is merely 256b/205s=1.24bps. However, when the file size reaches 100MB, namely 25,600 pages, the covert channel only takes around 280s to complete the transmission. In this way, the bit rate can surge to above 90bps. This means that we should select relatively large memory size for the covert channel construction. Unfortunately, we cannot keep increasing memory size indefinitely. In one experiment, we load a 300MB file into memory to build the channel. However, the result is, even when we increase the sleeping time to 3,600s, the covert channel still cannot achieve 100% correctness. This indicates that when the memory size reaches a certain threshold, the covert channel will lose reliability. Moreover, using too much memory might easily expose our covert channel. Thus, a memory size in the range of 80MB to 100MB can be a good choice.

The performance of the covert channel is also dependent upon the system workload. Given a certain memory size, we run a set of experiments to study how the bit rate changes when the system workload increases. In the experiments, we first boot two virtual machines as the sender and receiver, and then we gradually increase the system workload by launching more virtual machines and running CPU and memory intensive benchmarks on the virtual machines.

Figure 4 illustrates the dynamics of the bit rate under different system workloads, which can be divided into six cases. For the first five cases, the size of the leveraged memory is fixed at 10MB. Initially, only two virtual machines are launched and they collude with each other to construct the covert channel. In this first case, the stable sleeping time is 145s and the bit rate is 17.66bps. In the second case, we boot another two virtual machines (we call them the *irrelevant VMs*) with the same configuration and keep them idle. Now the stable sleeping time is increased to 180s and thus the bit rate is reduced to 14.44bps. To further increase system workload, in the third case, we run CPU intensive benchmark Cuadro [11] on the two *irrelevant VMs*. Again, the bit rate drops to 11.64bps. Then we also run the benchmark on the sender, i.e. the victim VM. This fourth scenario is close to a real world one: the victim machine and other co-resident machines are busy running services while the attacker-controlled receiver can keep idle. Such a system configuration in the fourth case yields the stable sleeping time of 225s with the bit rate of 11.38bps. In the fifth case, when we also run the benchmark on the receiver, the bit rate drops a little again to 11.13bps. These results agree with our assumption: increasing system workload will result in a longer stable sleeping time and a lower bit rate. This is due to the competition between KSM and other processes of the

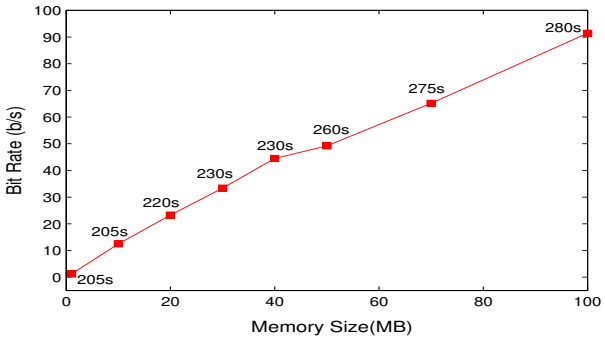

Figure 3. The relationship between memory size and bit rate. The stable sleeping time is also marked.

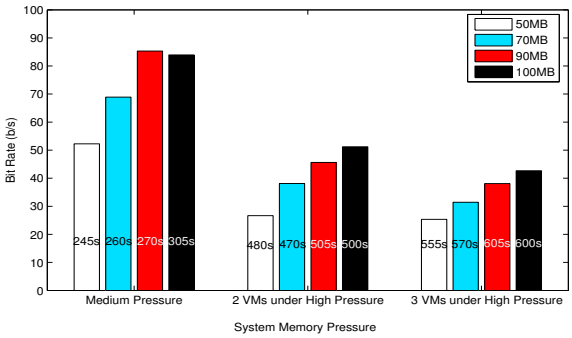

Figure 5. The relationship between system memory pressure and bit rate. The stable sleeping time is also marked.

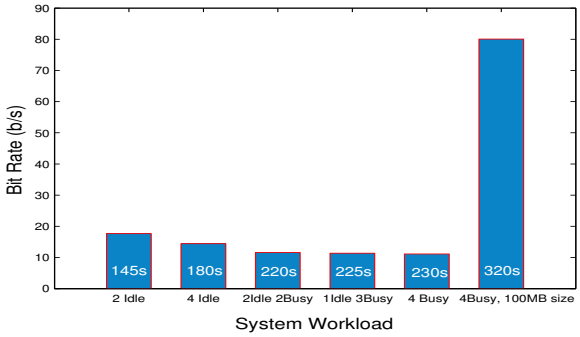

Figure 4. The relationship between computation workload and bit rate. The stable sleeping time is also marked.

system on CPU resources. The more CPU resources used by guests, the fewer CPU cycles left for KSM to perform deduplication.

However, from Figure 4, we can also see that the performance of the covert channel is not severely influenced by system workload. When the system workload increases, the bit rate only drops slightly. Moreover, in the experiments of the sixth case, all the virtual machines are busy running benchmarks and we leverage 100MB memory to construct the covert channel. Since the stable sleeping time is no more than 320s, we can still achieve a bit rate above 80bps. This indicates that even in a close-to-real-world scenario where multiple users run different services on different VMs above a single physical machine, our covert channel is still able to transfer information in a decent rate. According to the analysis of [10], a real world L2 cache channel can only achieve a bit rate around 11bps with some errors. Thus, it is clear that our memory deduplication covert channels outperform L2 cache covert channels.

Finally, we evaluate the performance of our covert channel under high memory pressure. Figure 5 illustrates the dynamics of the bit rate under different system memory pressure. In this series of experiments, we boot 4 VMs and all of them run the CPU benchmark except for the receiver. We perform the experiments with different memory pressure. At the beginning, the memory usage of all virtual machines remain medium. From Figure 5 we can see under medium memory pressure, the achievable bit rate is decent. Then, we launch a memory intensive benchmark LMBench [12] on those two *irrelevant VMs*. The benchmark makes the virtual machines under high memory pressure. Since our system is configured as over-committed, the entire system suffers from a high memory pressure. Due to the high memory pressure, the bit rate of the channel decreases. After we run the benchmark on the sender side as well, the stable sleeping time rises and the bit rate drops again. Such results are expected: under high memory pressure KSM has many more pages to scan, which takes more time. Moreover, over-committing memory could cause swapping, which might affect the deduplication as well. However, even in the extreme scenario where the system is under a memory pressure as high as in our experiments, a 100MB memory deduplication covert channel can still achieve a bit rate above 40bps, which is acceptable.

B. Virtualization Detection

To evaluate the effectiveness of the proposed virtualization detection method, we conduct four groups of experiments to cover the following four scenarios: (1) run the experiment on the guest OS, while memory deduplication is enabled on the host OS; (2) run the experiment on the guest OS, while memory deduplication is disabled on the host OS; (3) run the experiment on the host OS, while memory deduplication is enabled on the host OS; and (4) run the experiment on the host OS, while memory deduplication is disabled on the host OS. On current Linux systems, we can enable KSM by writing 1 to `/sys/kernel/mm/ksm/run` and disable it by writing 0 to the same file. The KSM in the guest OS is always turned off so as not to interfere with the KSM at the host level.

For each group, the experiment is conducted in two separate rounds. First, we load the ELF binary for *apache* into memory, write to each page, and record the access time for each page. Second, we load the same ELF binary

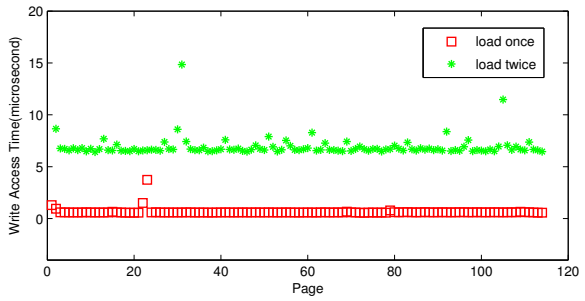

Figure 6. Write access to apache pages on guest OS with memory deduplication enabled on host

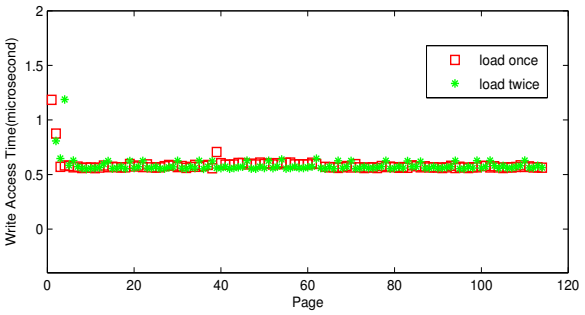

Figure 7. Write access to apache pages on guest OS with memory deduplication disabled on host

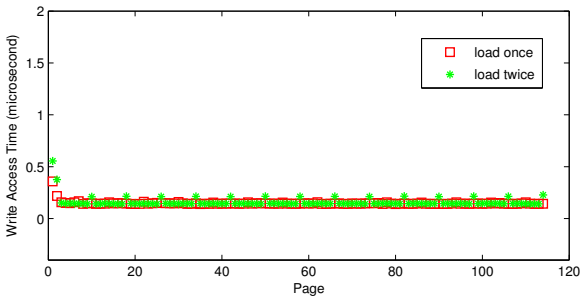

Figure 8. Write access to apache pages on host OS with memory deduplication enabled

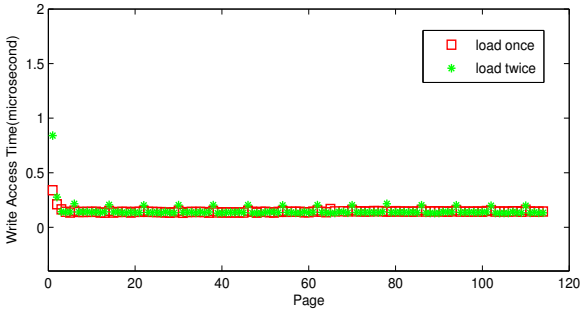

Figure 9. Write access to apache pages on host OS with memory deduplication disabled

into two memory regions, wait for five minutes, allowing memory deduplication to take effect, and then for one of the two regions, we write to each page and record the access time.

Figures 6, 7, 8, and 9 show the results for the four groups of experiments, respectively. In all the four figures, the red squares represent the write access time when we load the binary once, while the green asterisks represent the write access time when we load the same binary twice into memory (at two different memory regions). The size of the *apache* ELF file is 468,560 bytes, indicating that it occupies about 114 pages in memory.

In the first scenario, the write access in the second round should take longer time than the first round, because of the extra copy operation. As shown in Figure 6, the green asterisks are clearly on top of the red squares, indicating that our experimental results match with the analysis above. In the other three scenarios, the write accesses of the two rounds should have no significant difference. From Figures 7, 8, and 9, we can see that in these scenarios, there is no significant difference between the red squares and the green asterisks, indicating that the write access times at different rounds are no longer distinguishable.

The third scenario deserves more explanation. We run our program on the host OS, and Linux KSM is enabled on the host OS. However, Linux KSM has no performance impact upon write access, as shown in Figure 8. This is because in the current Linux KSM implementation, page sharing is not transparent to applications. The applications that want to benefit from Linux KSM have to explicitly invoke a system call function *madvise()* to inform the kernel that their memory can be shared. But madvise() is not called in our experiment, and thus Linux KSM will not merge the pages owned by the application. In other words, although Linux KSM is on, it is not utilized by the application and thereby producing zero impact on the application's write accesses.

C. Kernel Integrity Monitoring

The purpose of this experiment is to demonstrate how we utilize memory deduplication and *smaps* to detect rootkits that modify kernel read-only data. We run our monitoring program on the physical machine (host OS) and we monitor the runtime integrity of a guest OS that is running Fedora 16.

On Linux systems, after kernel compilation, an ELF file called vmlinux will be generated. Generally, this file includes *.text* section, *.data* section, *.rodata* section, *.bss* section, etc. Among these sections, the *.rodata* section is the one that stores read only data such as the system call table. In fact, the system call table is a commonly modified data structure targeted by kernel module rootkits [13], [14], [15], [16]. By redirecting system calls from standard system call code to malicious code, attackers can hide files, processes, and

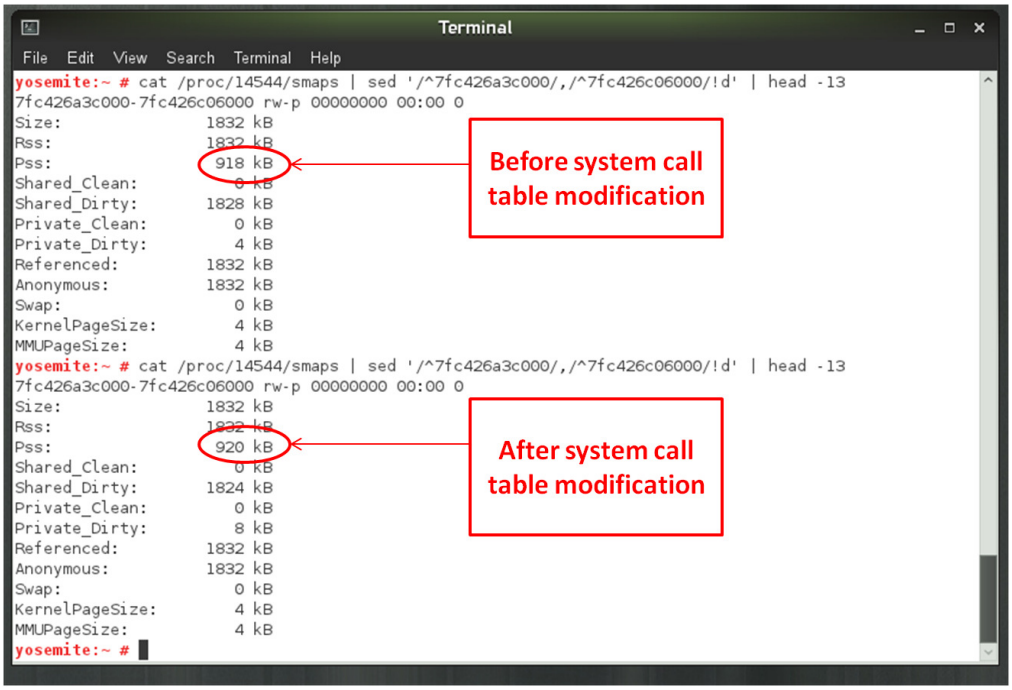

Figure 10. Kernel Read-Only Data Pages Monitoring

network connections. So, it is critical to ensure the integrity of the system call table. We achieve this by monitoring the *.rodata* section.

First, we use *objcopy* command to dump the *.rodata* section from the vmlinux (of Fedora 16) into a small file. We call this file rodata file below. In our experiments, the size of the vmlinux is 151MB; while the size of the rodata file is 1.8MB, including 457 pages plus 1 incomplete page in our system whose page size is 4KB. And then we launch the virtual machine. On the host OS, we also load the rodata file into memory (using the C program we mentioned before). Since the rodata file is relatively small (1.8MB), it usually takes less than one minute to merge its identical pages. After that, we read the PSS value corresponding to the file. As Figure 10 shows, the PSS is 918KB, i.e., PSS=(457/2+1)*4KB=918KB, where '1' means the incomplete page that will not be shared.

Later on, we port a well-known real world rootkit called *override* [17], [18], [19] that hijacks several system calls by modifying the system call table. The system calls being hijacked include sys_getuid(), sys_geteuid(), sys_getdents64(), sys_chdir(), and sys_read(). After we successfully load the rootkit as a loadable kernel module into the target Linux kernel, the page corresponding to the system call table should become unshared [4]. From Figure 10, we can see that PSS is now 920KB, i.e., PSS=(456/2+1+1)*4KB=920KB, and the number of shared pages decreases from 457 to 456. Therefore, we can infer the kernel read only data has been modified.

Based on the experimental result above, it is evident that our solution can effectively detect the malicious modifications on the system call table. We believe that the same mechanism will work for any kernel pages that are not supposed to be changed during runtime.

V. Discussion

When page sharing mechanism is initially proposed, it is for the sake of saving memory. Thus, it is understandable that the developers might have overlooked some hidden security issues. Based on our study, we observe that current page sharing mechanism is practically not secure, and therefore we suggest to improve the page sharing mechanism for striking a better balance between performance and security. In the following, we first discuss potential countermeasures to thwart the disclosed attacks, and then discuss an issue related to the kernel integrity monitoring technique.

A. Potential Countermeasures

To defeat covert channels, we propose to use a random sharing scheme - merge identical pages randomly. In doing so, in the covert channel attack scenario, the receiver will not be able to reliably decide which pages are written by the sender, and hence it cannot decode the information being transferred.

In order to prevent virtualization detection, we can modify the page sharing system so that identical pages from the

[4]In standard Fedora 16 (32-bit) kernel, there are 347 entries in the system call table. Given the fact that one entry corresponds to 4 bytes, the whole table occupies less than one page.

same virtual machine or the same process are not merged, unless they are zero pages. In this manner, the attacker will not see the write access time difference between loading two copies of the same file into memory and just loading one copy, because neither can induce page sharing. One might argue that this strategy may reduce the benefit of page sharing, however, given the fact that there are already existing mechanisms for sharing same-content pages within a virtual machine, for example, shared libraries, we believe the opportunities lost due to not merging (non-zero) identical pages within a virtual machine would be minor.

B. KSM Off

As to our kernel integrity monitoring technique, one might argue that, to eliminate false positives, when the read only data pages have been merged, we need to turn off KSM, which might cause a performance loss. However, this is not necessarily true. First of all, we propose to turn off KSM only because, from security's perspective, it's the safest way to eliminate a false positive theoretically. In practice, whether or not turning off the KSM would not affect the detection results. In our experiments, we never encounter any false positives no matter whether KSM is turned off or not. Also, in reality, since the kernel read-only data just occupies one page, it's easy to determine if any PSS changes is caused by data modification or by false positives, because they would incur different PSS changes. Last but not least, although KSM reduces memory usage, it increases CPU usage. Whether KSM improves performance depends on the type of workload; and system administrators are expected to keep KSM on or turn it off based on their environments [20].

VI. Related Work

This section briefly summarizes previous work related to covert channel construction in a virtualized environment, virtualization detection, and kernel integrity monitoring, respectively.

A. Covert Channel

Ristenpart et. al [8] have succeeded in identifying co-resident virtual machines and then launching L2-cache based side channel attacks in a real cloud environment, i.e., Amazon EC2. In fact, L2 cache channel is one of the most widely studied covert channels. Compared with L2 cache channel, our memory deduplication channel can achieve higher reliability as well as higher bit rate. According to the work done by Xu et al. [10], the optimal bit rate of L2 cache covert channel can be around 262bps and the achievable bit rate in EC2 is around 11bps. Our memory deduplication covert channel can achieve 1,000bps in an ideal situation, where we set the entire system to be idle and leverage 400MB memory to build the channel. Meanwhile, in a realistic scenario, the bit rate of our covert channel can achieve more than 40bps. Moreover, our covert channel is more reliable than L2 cache channel, i.e., with the same bit rate our channel will introduce much fewer errors and by properly setting sleeping time our channel can reach 100% correctness. Wu et. al [21] proposed the memory bus based covert channel, and their transmission rate in Amazon EC2 cloud can achieve 100bps. However, their approach relies on the lower level features and therefore has some hardware limitations, e.g., the atomic instructions it exploits are platform dependent.

Milo's et al [22] mentioned the potential threat of covert channel attacks based on page sharing. However, in their context, by covert channel, they primarily meant information leakage rather than two guest OSes collude with each other to transmit information. Similarly, Suzaki et al. [1] also, in a few words, described the potential threat of side channel attacks leveraging memory deduplication. Again, they just mentioned that attackers can use the channel to search for another VM in the cloud. In contrast, we not only implemented a covert channel to transmit information but also evaluated it with experiments and conducted a comprehensive analysis to verify its feasibility and performance.

Besides memory deduplication, there is another type of deduplication, commonly known as data deduplication or storage deduplication [23], [24], [25]. Data deduplication means that when there are multiple copies of the same data, only one copy is stored. Data deduplication is a key technique used by online storage service vendors, such as Dropbox [26], one of the most popular cloud storage provider. By storing and transferring only one copy of redundant data, data deduplication can reduce disk space consumption significantly.

However, recent research efforts [27], [28] have demonstrated that data deduplication also offers new chances to attackers. By exploiting two crucial features of the current data deduplication mechanism: source-based deduplication and cross-user deduplication, attackers can reveal the content of other users' files; in addition, a similar covert channel can also be constructed by attackers.

B. Virtualization Detection

In the past several years, many solutions have been proposed to detect virtualization. However, most of them are not generic enough. For example, RedPill [29] and Scooby Doo [30] can only detect VMWare, while the LDT [31] based approach only works on Windows guest OSes, and the counter-based mechanism [32] requires a multi-core processor. In contrast, our approach is more generic, as it does not rely on any specific hardware or guest/hypervisor implementation, and the feature it leverages—memory deduplication—is commonly used in various mainstream hypervisors.

A variety of timing analysis based detection mechanisms have also been proposed [33], [34]. The basic idea is that, some instructions (e.g., RDMSR) will be intercepted by

the hypervisor, and hence the execution time will increase compared to running on a real machine. However, timing analysis suffers from the difficulty of obtaining accurate timing, since the hypervisor can fool the guest OS [35]. In contrast, our approach is more stealthy, and thus can hardly be cheated by the hypervisor.

C. Kernel Integrity

The ultimate goal of kernel integrity monitoring is to detect rootkits. Traditional rootkit detection tools running inside the system can be easily defeated when the system is compromised by attackers. Another type of rootkit detection tools, which rely on an external PCI card to acquire system memory, are considered to be more reliable [36], [37], but recent research [38] has demonstrated that they can still be fooled by attackers.

A new type of rootkit detector is based on virtualization. Livewire [3], which presents the idea of virtual machine introspection, namely inspects a virtual machine from the hypervisor level. By marking the code section as well as some other critical data structures read-only, Livewire is able to ensure kernel code integrity. Inspired by Livewire, in recent years, more and more researchers have proposed to build defense tools using this virtual machine introspection methodology, such as VMwatcher [4] and HIMA [39]. As alluded to earlier, a common problem in these virtual machine introspection based tools is that, the semantic gap between the hypervisor and the guest OSes prevents the hypervisor from monitoring guest OSes efficiently. In other words, to monitor the guest OSes, significant efforts are required; for example, in VMwatcher, the authors wrote a Windows device driver to interpret Linux file systems, which are the root file system for the guest OS. Worse yet, these tools highly depend on the internal kernel data structures. When the internal kernel data structures change, these tools need to be changed accordingly. Thus, the maintenance cost of such tools are quite high. In contrast, although our solution is also based on virtualization technology, it does not require any understanding on the guest OS. As long as we have a clean Linux kernel image file (vmlinux), we can extract the read-only section and load it into memory, and then use the detection technique we described before to monitor kernel integrity. In other words, our solution is more generic since it is applicable to the vast majority of modern Linux kernel versions.

VII. Conclusion

Memory page sharing, or memory deduplication, is an important feature in modern hypervisors, and even in operating systems. Whereas this feature is originally designed for improving performance, it can be exploited for security purposes and induce unexpected implications on system security. In this paper, we have demonstrated that attackers can build a new covert channel and detect virtual machine monitors based on memory deduplication. Meanwhile, we have also showed that defenders can take advantage of this feature to monitor kernel integrity. We have developed prototypes of the proposed mechanisms and conducted a series of experiments on Linux/KVM to validate their feasibility and effectiveness. In our future work, we plan to investigate effective defense approaches to thwarting information leakage and other potential security threats posed by memory deduplication.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback. This work was partially supported by NSF grant 0901537 and ONR grant N00014-13-1-0088.

References

[1] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Software side channel attack on memory deduplication," *SOSP'11 POSTER Session*, 2011.

[2] K. Suzaki, K. Lijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest os," in *Proceedings of the Fourth European Workshop on System Security*, 2011, pp. 1–6.

[3] T. Garfinkel and M. Rosenblum, "A virtual machine introspection-based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003, pp. 191–206.

[4] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proceedings of the 14th ACM conference on Computer and communications security (CCS'07)*, 2007, pp. 128–138.

[5] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P'11)*, 2011, pp. 297–312.

[6] C. Waldspurger, "Memory resource management in vmware esx server," in *Proceedings of the 5th Symposium on Operating systems design and implementation (OSDI'02)*. USENIX, 2002, pp. 181–194.

[7] "New features of xen 4.0," http://wiki.xen.org/xenwiki/Xen4.0.

[8] 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'09)*, 2009, pp. 199–212.

[9] B. Lampson, "A note on the confinement problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973.

[10] 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 the 3rd ACM workshop on Cloud computing security*, 2011, pp. 29–40.

[11] "Cuadro cpu benchmark," http://sourceforge.net/projects/cuadrocpubenchm.

[12] "Lmbench," http://www.bitmover.com/lmbench/.

[13] Halflife, "Abuse of the linux kernel for fun and profit," http://biblio.l0t3k.net/rootkit/en/P50-05.txt.

[14] M. Jakobsson and Z. Ramzan, *Crimeware: understanding new attacks and defenses*. Addison-Wesley Professional, 2008, p. 261.

[15] Plaguez, "Weakening the linux kernel," http://www.phrack.org/issues.html?issue=52&id=18.

[16] Pragmatic, "Complete linux loadable kernel modules," http://www.thc.org/papers/LKM_HACKING.html.

[17] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing," in *Proceedings of the 11th Recent Advances in Intrusion Detection (RAID'08)*, 2008, pp. 1–20.

[18] Z. Lin, R. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2009, pp. 107–126.

[19] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proceedings of the 11th Recent Advances in Intrusion Detection (RAID'08)*, 2008, pp. 21–38.

[20] "How to improve kvm performance by adjusting ksm," http://searchenterpriselinux.techtarget.com/tip/How-to-improve-KVM-performance-by-adjusting-KSM.

[21] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 159–173.

[22] G. Miłós, D. Murray, S. Hand, and M. Fetterman, "Satori: Enlightened page sharing," in *Proceedings of the USENIX Annual Technical Conference (ATC'09)*, 2009, pp. 1–14.

[23] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage," in *Proccedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, 2009, pp. 197–210.

[24] L. You, K. Pollack, and D. Long, "Deep store: An archival storage system architecture," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE'05)*, 2005, pp. 804–815.

[25] K. Suzaki, T. Yagi, K. Iijima, N. Quynh, C. Artho, and Y. Watanebe, "Moving from logical sharing of guest os to physical sharing of deduplication on virtual machine," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Security (HotSec'10)*, 2010.

[26] "Dropbox," http://www.dropbox.com/.

[27] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P'10)*, 2010, pp. 40–47.

[28] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl, "Dark clouds on the horizon: Using cloud storage as attack vector and online slack space," in *Proceedings of the 20th USENIX Security Symposium*, 2011, pp. 65–75.

[29] J. Rutkowska, "Redpill," http://invisiblethings.org/papers/redpill.html.

[30] T. Klein, "Scooby doo," http://www.trapkit.de/research/vmm/scoopydoo/index.html.

[31] D. Quist, V. Smith, and O. Computing, "Detecting the presence of virtual machines using the local data table," http://www.offensivecomputing.net/files/active/0/vm.pdf, 2006.

[32] C. Thompson, M. Huntley, and C. Link, "Virtualization detection: New strategies and their effectiveness," http://www-users.cs.umn.edu/~cthomp/papers/vmm-detect-2010.pdf.

[33] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: Vmm detection myths and realities," in *Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, 2007, pp. 1–6.

[34] J. Franklin, M. Luk, M. Jonathan, A. Seshadri, A. Perrig, and L. van Doorn, "Towards sound detection of virtual machines," *Advances in Information Security, Botnet Detection: Countering the Largest Security Threat*, 2008.

[35] M. Athreya, "Subverting linux on-the-fly using hardware virtualization technology," Master's thesis, Georgia Institute of Technology, 2010.

[36] N. L.Petroni Jr., T. Fraser, J. Molina, and W. A.Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 179–194.

[37] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Transactions on Dependable and Secure Computing*, pp. 670–684, 2011.

[38] J. Rutkowska, "Beyond the cpu: Defeating hardware based ram acquisition," http://www.blackhat.com/presentations/bh-dc-07/Rutkowska/Presentation/bh-dc-07-Rutkowska-up.pdf.

[39] A. M.Azab, P. Ning, E. C.Sezer, and X. Zhang, "Hima: A hypervisor-based integrity measurement agent," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'09)*, 2009, pp. 461–470.