# Detection of Service Level Agreement (SLA) Violation in Memory Management in Virtual Machines

Xiongwei Xie
Dept. of Software and Information System
UNC Charlotte
Charlotte, NC 28223
Email: xxie2@uncc.edu

Weichao Wang
Dept. of Software and Information System
UNC Charlotte
Charlotte, NC 28223
Email: wwang22@uncc.edu

Tuanfa Qin
School of Computer and Electronic Information,
Guangxi University,
Nanning, Guangxi, China
Email: tfqin@gxu.edu.cn

***Abstract*—In cloud computing, quality of services is often enforced through Service Level Agreement (SLA) between end users and cloud providers. While SLAs on hardware resources such as CPU cycles or bandwidth can be monitored by low layer sensors, the enforcement of security SLAs stays a very challenging problem. Several high level architectures for security SLAs have been proposed. However, details still need to be filled before they can be deployed.**

**In this paper, we propose to design mechanisms to detect violations of security SLAs. Specifically, we focus on unauthorized accesses to memory pages of a virtual machine and violation of the memory deduplication policies. Through measuring the accumulated memory access latency, we try to derive out whether or not the memory pages have been swapped out and the order of accesses to them. These events will then be compared to access commands issued by the local VM. In this way, unauthorized memory accesses or violation of deduplication policies can be detected. Compared to existing approaches, our mechanisms do not need explicit help from the hypervisor or third parties. Therefore, it can detect SLA violations even when they are initiated by the hypervisor. We implement our approaches under VMWare with Windows virtual machines. Our experiment results show that the VM can effectively detect the violations with small increases in overhead.**

## I. 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 [1], 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 [2], [3], network downtime [4], and bandwidth [2]. Several multi-layer monitoring structures [5], [4], [6], [7] 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 [8] 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 [9]. 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 paper, 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 mainframe virtual machine hypervisors such as VMware ESX and ESXi [10], Extended Xen [11], and KSM (Kernel Samepage Merging) [12] 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 [13], [14], [15], many end users ask the hypervisor to disable memory deduplication for their VMs. 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 VMs under their management and reconstruct their internal views [16], [17]. 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.

To detect such violations, we propose to design mechanisms based on memory access latency. When we revisit the two types of violations described above, we can find out that both attacks involve unauthorized memory access. For the attack on memory deduplication, although the other VM is accessing only its own pages, the victim VM is impacted because of the shared memory. For the attack of sneak peek, the attacker violates the security SLA and reads the memory of the victim VM. Under both cases, the order of accesses to the VM's memory pages is changed. Our detection mechanisms will capture such changes.

According to the documents released by major hypervisor companies such as VMWare and research results of other investigators [18], [19], Least Recently Used (LRU) memory pages are still the top choices during memory reclaiming. Therefore, unauthorized accesses to the memory pages of the victim VM 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 detect memory accesses that are not initiated by our running programs. Since the reference pages are hidden within the real data and program pages, it is very hard for the attacker to identify them. We have implemented the approaches in virtual machines under VMWare and tested them. Our experiment results show that the approaches can effectively detect unauthorized memory access operations with small increases in overhead.

The contributions of the paper 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, we choose unauthorized access to memory pages of a virtual machine as an example of security SLA violations. 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 paper is organized as follows. In Section II 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 III we present the details of our approaches. In Section IV we present the experimental results in real systems. In Section V we discuss the safety of the approaches. Finally, Section VI concludes the paper.

## II. Related Work

### A. Information Leakage among Virtual Machines

With the proliferation of cloud computing, more and more companies start to use it. One big security concern in 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 [20], [21] to steal information about key stroke or Internet surfing histories. Research in [22] 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. An implementation of the key extraction attack was presented in [23]. The delay caused by separation of deduplicated memory pages in virtual machines has been used to identify guest OS types [13] or derive out memory page contents [24].

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 [25] the processor is responsible for updating the memory page mapping and the page table. Szefer *et al.* [26] proposed to use memory level isolation or encryption to protect guest VMs from a compromised hypervisor. The software based approaches adopt more diverse mechanisms. For example, in [27] the cache that is used by security related processes is labeled with different colors to prevent side channel attacks. In both [28] and [29], the hypervisor monitors the memory access procedures to prevent cross-VM information flow. In [30], researchers tried to establish a very small compartment to allow VMs to run in an isolated state.

### B. 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 [2], [3], [4]. Group two push the approaches one step further through formalization of both service capabilities and business process requirements [5]. In this way, a language can be used for direct communication between the two layers. Dastjerdi *et al.* [31] 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 [32] raised the question whether security can be adequately expressed in an SLA. Casola *et al.* [33] proposed a methodology to evaluate and compare security SLAs in web services. Chaves *et al.* [34] explored security management through SLAs in cloud computing. An infrastructure to enforce security SLA in cloud services is described in [9]. Haeberlen [8] proposed an approach to differentiate the responsibility of a user from that of the service provider when some security breach happens.

## III. The Proposed Approaches

In this section, we will present the details of the proposed approaches. We first introduce the technique of memory deduplication and how it impacts memory access latency. 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.

## A. 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 1. 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 VMs. If memory de-duplication is enabled, we need to store only one copy of multiple identical pages. Therefore, the two VMs 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 VMs and the performance penalty caused by memory access miss.

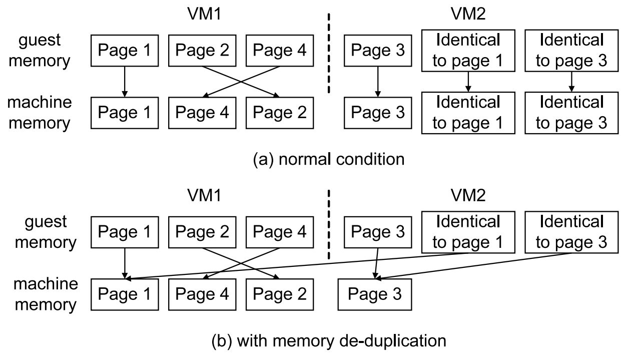

Fig. 1. Memory de-duplication reduces the memory footprint size.

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 VM 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 VM from changing another VM'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 [35]. Since the memory pages have the size of 4KB, this technique will not impact the results of memory deuplication.

## B. System Assumptions

In the investigated scenarios, we assume that the security SLA signed between the cloud provider and customers includes the following two requirements: (1) the provider cannot apply memory deduplication technique to the memory pages of the guest VMs; and (2) without a customer's permission, the provider cannot peek at the memory pages of her VM. 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 VM. 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 VM can consume. Without losing generality, we assume that VMs use $4KB$ memory pages.

We assume that the cloud service provider is not malicious but very curious. At the same time, he wants to squeeze as many virtual machines as possible into a single physical box. From this point of view, he has the motivation to enable memory deduplication. 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. He does not have the capability to alter the source code of the hypervisor. However, he may configure the software to enable memory deduplication. He may also read the memory page contents of other VMs through the virtual machine manager. All such operations can be conducted without the permission of end users.

## C. 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 hard disk.

Now let us reexamine the two violations of interest. When the hypervisor or an attacker takes a sneak peek at memory pages of a guest VM, 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 a record of the 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 VM, we can confirm that some unauthorized access has happened.

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 VM memory. If the hypervisor does not enable memory deduplication, the files will occupy different chunks of memory. On the contrary, 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 [13]. 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.

## D. Details of Implementation

Although the basic idea of the detection mechanisms is straightforward, we face many 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.

*1) Choice of the 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 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 two types of violations. The selection of memory pages for the detection of sneak peek is very tricky. Theoretically, attackers or the hypervisor could choose any memory pages of the guest VM to read. It is impossible for the guest VM to know beforehand which pages to examine. In real world, however, the selection range is much smaller. The 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 file, the hypervisor and attackers will not be able to identify them. Therefore, when they conduct unauthorized memory access to the guest VM, 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 VM can read from/write to these pages to measure the access delay. We propose to construct data files and actively load them into our VM's memory. Since deduplication is conducted at the page level, the positions 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 VMs. Under this case, we can use methods in [20] to make sure that these VMs are located in the same physical box so that deduplication can be conducted.

*2) Measurement of Access Time:* To successfully detect the SLA violations, we must accurately measure the information 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 [36], tickless timekeeping can keep time at a finer granularity. Therefore, we choose the Windows API $QueryPerformanceCounter$ to measure the duration. Previous research [37] has also shown that the time measurement accuracy may be impacted by the workload on the physical box. We can use the lightweight toolset $TiMeAcE.KOM$ [37] to assess and fix the measurement results.

*3) Verification of Memory Access Order:* As explained in Section III.C, the detection of unauthorized memory access depends on the verification of one fact: some memory pages that should have been swapped out are still in memory. The reason that these pages are not swapped out 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 VM 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 [13] 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 when he randomly selects guest VM pages to read. In this way, the reference pages can effectively serve their purposes.

## E. 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 two SLA violations.

Figure 2.a illustrates the detection of unauthorized memory accesses. The guest VM 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 3, 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 3 for different groups of reference pages increase exponentially. Assuming that at time $7t$ the VM 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 switched out first. Then the 4th group of reference pages are also switched out. At time $8t$ when the predetermined interval for group 4 expires, we will read this group of reference pages. Since they have been switched 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.

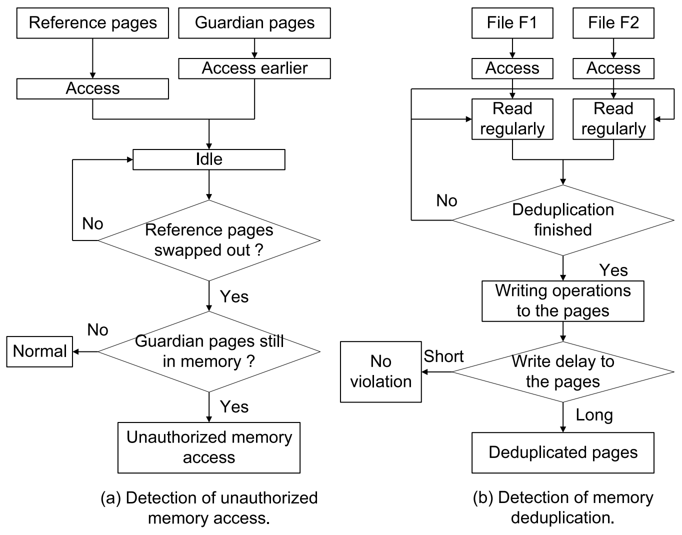

Fig. 2. Detection procedures of the SLA violations.

Figure 2.b illustrates detection of the violations of deduplication policies. Here two applications in the guest VM will read the files $F1$ and $F2$ into its memory. The two files contain identical memory pages. Should deduplication is enabled in the guest VM, 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 [13], 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.

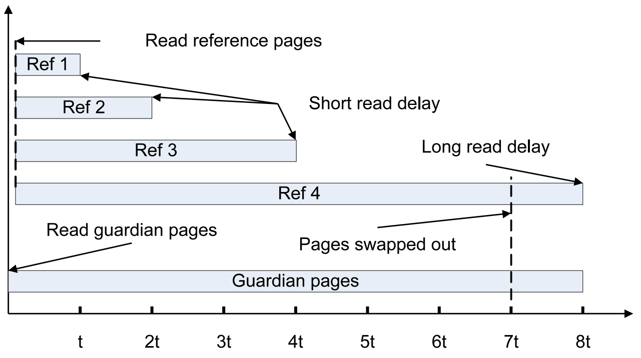

Fig. 3. Using reference pages to detect memory swap operations.

## IV. Experimental Results

### A. 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, 512MB RAM, and 8GB hard disk. Our experiments show that when there are more than 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 pages.

### B. Experiments and Results

We conduct two groups of experiments to evaluate the detection of unauthorized memory access and violation of deduplication policies respectively. Below we will first present the results of the baseline experiments. The detection capabilities and overhead of our approaches in more complicated scenarios will be discussed later.

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 VM that runs our detection algorithms, and an attacker's VM that stealthily accesses the memory of the victim. Instead of locating some malware to penetrate VMWare and get access to the guest VM's memory, we propose to use the Virtual Machine Communication Interface (VMCI) [38] to simulate such an attack. Specifically, in the guest VM, we create a block of shared memory so that the attacker's VM can access the guardian pages remotely. The guest VM will load both reference pages and guardian pages into its memory, as described in Section III.E. 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 VM remotely accesses the guardian pages, they will be kept in the memory. On the contrary, the reference pages will be swapped out. When the guest VM measures the access delay to the guardian pages, it will figure out that they are still in the memory and detect the unauthorized access.

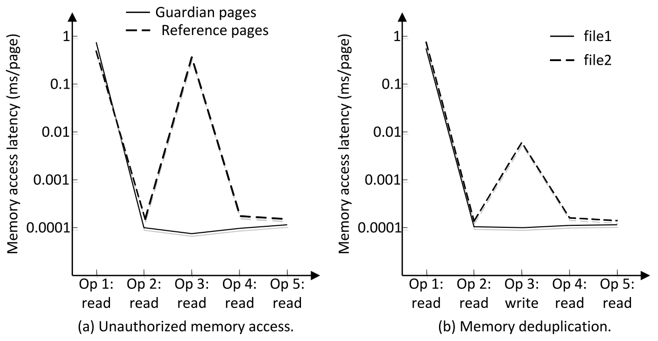

Fig. 4. Detection capabilities of the approaches in baseline scenarios.

Figure 4.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 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 VM, it is unauthorized access.

The second group of experiments assess the detection of the violation of 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 2.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 [13], [24], [14], we can estimate the time that the algorithm needs to merge the pages. When the estimated delay expires, we will issue the "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 4.b illustrates the detection results. Reading Operation 1 has 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 process 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 VM 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 5.

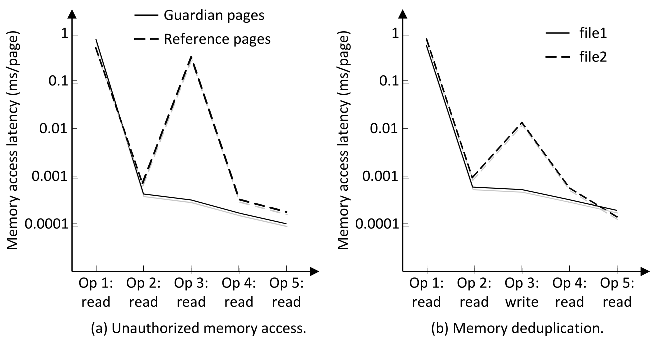

Fig. 5. Detection capabilities of the approaches under intense CPU demand.

From the figure, 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.

## C. Impacts on System Performance

The proposed violation detection mechanisms will occupy some memory. They will also incur CPU operations during the detection procedures. Therefore, they may impact the overall system performance. In the following groups of experiments, we try to assess such impacts. Since the detection algorithm for unauthorized memory access demands more CPU and memory resources than the approach for deduplication policies, we choose it as the benchmark for evaluation.

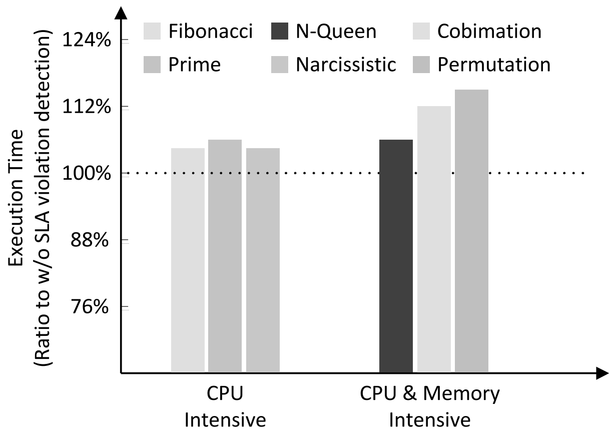

Fig. 6. Impacts on system performance.

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 computes the fibonacci sequence; (2) the $Prime$ benchmark calculates the prime numbers; and (3) the $Narcissistic$ benchmark generates the narcissistic numbers. Each of these software packages is running in parallel with the detection algorithm for unauthorized memory access. We measure the execution time of the software since this is the most intuitive parameter that end users use 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 computes solutions to the N-Queens problem in chess and stores the results in memory; (2) the $Combination$ benchmark computes all possible combinations of the input numbers; and (3) the $Permutation$ benchmark 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 6, 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 the worst case since we execute the detection algorithm continuously. In real worlds, end users can reduce the detection frequency (e.g. once every 30 minutes). Under that case, the increased execution time is smaller than 2%.

## V. Discussion

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 potential vulnerabilities of the approaches and our mitigation mechanisms. We will also discuss the schemes to reduce false alarms.

Several factors may impact the detection capabilities of the mechanisms. First, since we need to measure the memory access latency to determine whether or not the pages are in memory, the hypervisor or malicious attackers can manipulate the returned clock results to impact the detection. Based on the analysis in [36], tickless timekeeping can keep time at a finer granularity. Therefore, we choose the Windows API $QueryPerformanceCounter$ to measure the duration. Previous research [37] has also shown that the time measurement accuracy may also be impacted by the workload on the physical box. We can use the lightweight $TiMeAcE.KOM$ [37] to assess and fix the measured time.

Since the access latency to a single memory page is too short to be accurately measured, the detection of both 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 VM to reduce the chance of detection. For example, the hypervisor can select memory pages of the guest VM randomly to read. In this way, when our detection algorithm measures the access delay, only a small percentage of the guardian pages are still in memory. This scheme, however, will also hurt the information stealing procedures. For example, a virtual machine with 2GB memory has 512,000 memory pages. It may take the hypervisor hours to conduct a complete scan of the guest VM memory if it does not want the detection algorithm to raise an alarm. 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.

## VI. Conclusion

In this paper we propose mechanisms to detect violations of the SLA on memory management in virtual machines between cloud users and providers. Instead of proposing a generic security SLA enforcement architecture, we use the examples of unauthorized memory access and memory deduplication to the detection procedures. We have implemented both approaches under VMWare and tested them. The results show that they can effectively detect the violations without introducing much overhead. At the same time, the two approaches share some common operations and can work together to secure memory management of virtual machines.

Immediate extensions to our approach 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 mechanism. Second, we want to study the relationship between our approaches and existing security SLA enforcement architectures. If we can integrate them into the architecture, we will have a solid platform for future extension. The research will provide new information to strengthen the protection to virtual machines and end users of cloud.

## References

[1] S. J. Berman, L. KestersonTownes, A. Marshall, and R. Srivathsa, "How cloud computing enables process and business model innovation," *Strategy & Leadership*, vol. 40, no. 4, pp. 27–35, 2012.

[2] 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*, 2010, pp. 365–370.

[3] 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)*, 2012, pp. 499–508.

[4] 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)*, 2010, pp. 48–54.

[5] 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)*, 2009, pp. 783–790.

[6] 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.*, vol. 28, no. 7, pp. 1017–1029, July 2012.

[7] 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*, 2010, pp. 153–164.

[8] A. Haeberlen, "A case for the accountable cloud," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 52–57, April 2010.

[9] 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)*, Aug 2011, pp. 202–209.

[10] VMWare, "esxi configuration guide," VMware vSphere 4.1 Documentation, 2010.

[11] 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*, vol. 53, no. 10, pp. 85–93, 2010.

[12] A. Arcangeli, I. Eidus, and C. Wright, “increasing memory density by using ksm,” in *Linux Symposium*, 2009, pp. 19–28.

[13] R. Owens and W. Wang, “Non-interactive os fingerprinting through memory de-duplication technique in virtual machines,” in *IEEE International Performance Computing and Communications Conference (IPCCC)*, 2011.

[14] 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.

[15] 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)*, 2013, pp. 1–12.

[16] 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*, 2007, pp. 128–138.

[17] 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.

[18] C. Lee, C.-H. Hong, S. Yoo, and C. Yoo, “Compressed and shared swap to extend available memory in virtualized consumer electronics,” *Consumer Electronics, IEEE Transactions on*, vol. 60, no. 4, pp. 628–635, Nov 2014.

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

[20] 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)*, ser. CCS’09. New York, NY, USA: ACM, 2009, pp. 199–212.

[21] E. Tromer, D. A. Osvik, and A. Shamir, “Efficient cache attacks on aes, and countermeasures,” *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, 2010.

[22] 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*, ser. CCSW’11. New York, NY, USA: ACM, 2011, pp. 29–40.

[23] 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*, ser. CCS’12. New York, NY, USA: ACM, 2012, pp. 305–316.

[24] R. Owens and W. Wang, “Fingerprinting large data sets through memory de-duplication technique in virtual machines,” in *IEEE Military Communications Conference (MILCOM)*, 2011.

[25] 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*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 272–283.

[26] 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*, ser. ICDCSW’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 248–252.

[27] 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*, ser. DSN-W’11, 2011, pp. 194–199.

[28] 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*, 2010, pp. 172–179.

[29] 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.

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

[31] 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.*, vol. 42, no. 4, pp. 501–518, Apr. 2012.

[32] R. R. Henning, “Security service level agreements: Quantifiable security for the enterprise?” in *Proceedings of the Workshop on New Security Paradigms*, 1999, pp. 54–60.

[33] V. Casola, A. Mazzeo, N. Mazzocca, and M. Rak, “A sla evaluation methodology in service oriented architectures,” in *Quality of Protection*, ser. Advances in Information Security, D. Gollmann, F. Massacci, and A. Yautsiukhin, Eds. Springer US, 2006, vol. 23, pp. 119–130.

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

[35] Symantec, “An analysis of address space layout randomization on windows vista,” http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf, Tech. Rep., 2007.

[36] VMware, “Timekeeping in vmware virtual machines,” http://www.vmware.com/files/pdf/techpaper/Timekeeping-In-VirtualMachines.pdf, 2011.

[37] U. Lampe, M. Kieselmann, A. Miede, S. Zller, and R. Steinmetz, “A tale of millis and nanos: Time measurements in virtual and physical machines,” in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science, K.-K. Lau, W. Lamersdorf, and E. Pimentel, Eds. Springer Berlin Heidelberg, 2013, vol. 8135, pp. 172–179.

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