PAPER Special Section on Cryptography and Information Security

Implementation of a Memory Disclosure Attack on Memory Deduplication of Virtual Machines

Kuniyasu SUZAKI[†a)], *Member*, Kengo IIJIMA[†], Toshiki YAGI[†], *and* Cyrille ARTHO[†], *Nonmembers*

SUMMARY Memory deduplication improves the utilization of physical memory by sharing identical blocks of data. Although memory deduplication is most effective when many virtual machines with same operating systems run on a CPU, cross-user memory deduplication is a covert channel and causes serious memory disclosure attack. It reveals the existence of an application or file on another virtual machine. The covert channel is a difference in write access time on deduplicated memory pages that are re-created by Copy-On-Write, but it has some interferences caused by execution environments. This paper indicates that the attack includes implementation issues caused by memory alignment, self-reflection between page cache and heap, and run-time modification (swap-out, anonymous pages, ASLR, preloading mechanism, and self-modification code). However, these problems are avoidable with some techniques. In our experience on KSM (kernel samepage merging) with the KVM virtual machine, the attack could detect the security level of attacked operating systems, find vulnerable applications, and confirm the status of attacked applications.

key words: *memory disclosure attack, virtual machine, memory deduplication*

1. Introduction

IaaS (Infrastructure as a Service) type cloud computing uses a tremendous number of virtual machines. Even though data centers offer vast resources, the efficiency of a virtual machine is very important, because it is directly linked to the cost of cloud computing. To improve efficiency, memory deduplication [1], [2], [7], [9], [16] reduces the consumption of physical memory. Memory deduplication merges same-content memory pages on a physical machine, allowing more virtual machines to run on limited resources.

However, memory deduplication is subject to memory disclosure attacks. A merged page has to be re-created when a write access is issued to that page, which is called COW (Copy-On-Write). The attack exploits this covert channel of COW, which is known exploit to leak information [9], [11], [17]. While the sequence of COW is logically valid and behaves consistently, the write access time is different between deduplicated and non-deduplicated pages. An attacker can use the time difference in a memory disclosure attack. This does not violate any restriction of SLA (Service Level Agreements) of cloud computing. It only measures the write access time of its own memory, and guesses memory contents of other virtual machines. The attack uses a characteristic of the shared resource of a virtual machine, making it a kind of cross-VM side channel attack [13].

The attack seems to be easy, but it includes some implementation issues. The attack is limited to exact matches on memory pages, which are aligned on a certain boundary. An attacker has to care about page caching caused by his own binary file. Memory pages are modified more than we expect and the attack take risks of false-negative and false-positive. For examples, memory pages are swap-out on victim's VM and attacker's VM. Furthermore, the difference of write accesses depends on the environment. This paper describes how to deal with these issues; the successes of a real attack on Linux and Windows Guest OS.

The attack is prevented by some improvements on the VMM (virtual machine monitor) or guest OS. However, the countermeasures decrease the performance of the virtual machine in general. Conversely, when the attack technique is used for live memory forensics, it can increase security. For example, an administrator may detect a prohibited application or illegal file. We describe that as future work.

This paper is organized as follows. Related work is reviewed in Section 2 and the detail of memory deduplication is described in Sect. 3. The issues on memory disclosure attack on memory deduplication are described in Sect. 4. Section 5 reports the results of the attacks to detect security level of attacked operating systems, to find vulnerable applications, and to confirm the status of attacked applications. Section 6 describes countermeasures and Sect. 7 notes the application of this technique. Section 8 discusses related issues and Sect. 9 summarizes our conclusions.

2. Related Work

Cloud storage deduplication has a same vulnerability [8], which is exhibited on Dropbox, MozyHome, and Melpole. The attack takes advantage of the elimination of traffic, when the same file is already uploaded. Using this phenomenon, the paper shows the vulnerability to leak some information, for example PIN code and salary. This attack seems to be applicable to memory deduplication, but several constrains are different. The big difference between storage and memory deduplication is that the former is static while the latter is dynamic. Storage deduplication is fixed when data is written to storage, and its status does not change. On the other hand, memory deduplication is dynamic, because memory contents change frequently at run-time. Even if the contents are same, the status in an operating system is

Manuscript received March 23, 2012.
Manuscript revised July 24, 2012.

[†]The authors are with the National Institute of Advanced Industrial Science and Technology (AIST), Tsukuba-shi, 350-8568 Japan.
a) E-mail: k.suzaki@aist.go.jp

DOI: 10.1587/transfun.E96.A.215

Copyright © 2013 The Institute of Electronics, Information and Communication Engineers

changed For example, swap-out operation saves the memory image on a disk but the status is not change. Anonymous pages are not used by an application, but still exist on the memory. The attack on memory deduplcaition has to consider these situations.

Vulnerability on memory deduplication is described in paper [11]. However, the paper shows the possibility of information leak and does not show the difficulty of the attack caused by the execution environment; for example, alignment mismatching and page cache. This paper clarifies the challenges for the attacker.

The IEEE 1394 (FireWire) attack is a physical side channel attack for memory disclosure. IEEE 1394 enables us to read from and write to physical memory by accessing a DMA controller, while an operating system owns the memory. The IEEE 1394 attack is very critical attack, but it requires physical cable connection with IEEE 1394. On the other hand, the memory deduplication attack does not require physical equipment. It requires a measurement of write access times on its memory. It is a critical attack in a multi-tenant cloud computing environment, because it does not violate any SLA (Service Level Agreements).

3. Memory Deduplication

Memory deduplication is a technique to merge same-content memory pages and reduce the consumption of physical memory. It is popular on a virtual machine monitor, because the memory images of virtual machines include many same-content pages, especially when the same guest OS runs on several virtual machines. Therefore, current virtual machine monitors are equipped with memory deduplication.

3.1 Type of Memory Deduplication

The techniques of memory deduplication are divided into two types; content-aware type and memory scan type. Content-aware deduplication is used on Disco's Transparent Page Sharing (TPS) [2] and Satori [9] on Xen. TPS reads page data from a special copy-on-write disk and checks whether the same page data is already present in main memory. If the pages match, TPS creates a shared mapping to the existing page. Satori has a similar policy for duplicate detection, although it does not use a special copy-on-write disk. Satori is implemented as para-virtualization on the Xen hypervisor and requires customization of the guest OS.

Memory scan type deduplication is used in Content-based Page Sharing of VMWare ESX [16], the Differential Engine [6] of Xen, and KSM (Kernel Samepage Merging) [1] of the Linux kernel. Content-based Page Sharing scans the VM's memory periodically and records fingerprints of each page. When the same fingerprint is found, it compares the contents of the relevant two pages, and merges them if they are identical. Differential Engine features not only memory-scan type deduplication, but also patching and compressing. When almost identical pages are found, the small difference is taken as a patch and the nearly identical pages are merged. Compression is used when a page is not active for a long time. KSM (Kernel Samepage Merging) included from Linux kernel 2.6.32 onward, is a general memory deduplication. It was developed for its virtual machine (KVM), but it is not limited to a virtual machine. In this paper we use KSM for memory deduplication.

3.2 KSM (Kernel Samepage Merging)

Most implementations of memory-scan type deduplication use the hash value of a page to check the similarity between pages. The initial implementation of KSM used the same technique, but it was re-implemented with another method to avoid a patent problem. KSM uses a simple 32-bit checksum for rough scanning. After the scanning, the exact similarity is computed by memcmp().

KSM manages memory pages with two red-black trees; one is for candidate pages of deduplication (called unstable tree), and the other one is for duplicated pages (called stable tree). Pages are identified by their 32-bit checksum in the trees. When the same content of a candidate page is found in the stable tree, the candidate page is merged with the stable tree. When the same content of a candidate page is found in the unstable tree, the two pages (candidate page and page in unstable tree) move to the stable tree.

Pages are scanned at intervals, which is defined at /sys/kernel/mm/ks m/sleep_millisecs. The default period is 20 msec. The time is the interval of the kernel daemon called "ksmd". The maximum number of pages that ksmd can use is limited (the default is 25% of the available memory). Therefore, not all pages are scanned at a time.

A merged page in the stable tree is re-created when a write access is issued to the page; this technique is called Copy-On-Write (COW). The write access is reflected in the new page. When the old same-contents page has no other more buddy pages, it is removed from stable tree.

4. Attack on Memory Deduplication

Memory deduplication may be subject to a memory disclosure attack from the attacker's VM to the victim's VM. Memory deduplication merges memory pages which have the same contents. When data is written to a deduplicated page, the page is re-created with a copy of its contents (Copy-On-Write). This causes the write access time to be slower than normal, because it includes the overhead to re-create the same page. An attacker can use this phenomenon as a memory disclosure attack in order to guess a process or an opened file on a victim's VM.

The preliminary idea of memory disclosure attack is as follows. The attacker allocates same-content pages of a process or file in the memory of the attacker's VM and waits for these pages to be deduplicaed. After that, the attacker issues one-byte write access to the pages. If the pages are deduplicated, the write access time is longer than normal. In order to distinguish the write access time difference, attacker must know the time difference between deduplicated

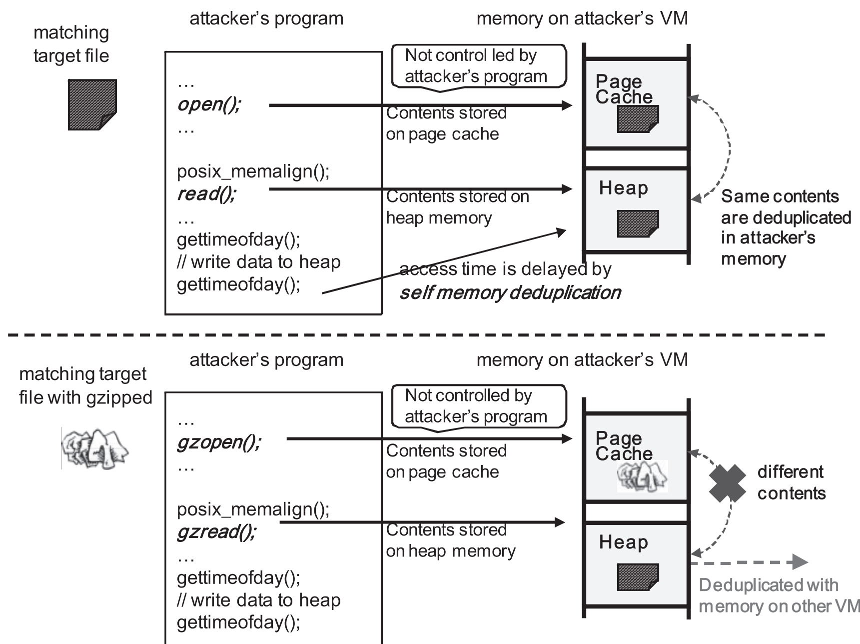


Fig. 1 The upper figure shows the self-reflection case. Contents of the page cache and heap are the same and deduplicated on a virtual machine. The lower figure shows the counter measure. A target file is gzipped and the contents on the page cache and heap are different.

and non-deduplicated page in advance. The attacker measures the write access time on zero-cleared pages which are deduplicated by themselves, and random data pages which are not deduplicated with other pages.

The preliminary idea is simple. However, the memory disclosure attack on memory deduplication includes some implementation issues, which is caused by memory alignment, self-reflection, run-time modification, and the timing problem. These implementation issues cause false-negatives and false-positives.

4.1 Alignment Problem

The memory disclosure attack requires an exact match on memory pages as well as the aligned address of the pages. When a process is created, a binary file is loaded to memory pages by an interpreter. On Linux, the ELF binary interpreter "ld-linux" is called to create a process. The contents of a binary file are loaded to aligned memory pages. An attacker has to prepare an identically aligned memory region to guess the same contents. It means that it is important to know the address of the alignment. Fortunately, attacker can use the POSIX posix_memalign() function to get an aligned memory region in heap memory.

On the other hand, the address is arbitrarily assigned, when a file is loaded by an application. The address is not identified by an attacker. However, the contents are stored in the page cache by the OS kernel, when a file is opened. The page cache is aligned on memory pages, allowing an attacker to know the exact matching pages.

The page cache is used to detect an opened file by the attacker. However, the page cache causes the self-reflection problem when a binary file is cached on the attacker's VM.

4.2 Self-Reflection Problem

Memory deduplication deals with same-content memory pages on a virtual machine. If an operating system and applications create same-content memory pages on a single VM, memory deduplication works in the same way. The feature makes a memory disclosure attack difficult, this issue is called the "self-reflection problem". The upper half of Fig. 1 shows the problem.

The self-reflection problem is caused by different memory management on the page cache and heap. When attacker program opens a target file using function open(), the contents of opened file are stored in its page cache memory by the OS kernel. After that, the attacker program loads the matching contents to its heap memory with alignment using memalign() and read() functions. At that time, the contents of the heap memory are deduplicated to the page cache memory contents. In this situation, the write access time to the contents on heap memory is always delayed with self memory deduplication. This leads to spurious matchings and causes false-positives.

To prevent self-reflection, the contents in heap and page cache must be different. The lower half of Fig. 1 shows this solution. The target file is compressed with gzip and expanded at run time. The contents on page cache are gzipped image. The contents of the heap memory are decompressed by function gzread(). The contents of the heap memory are not deduplicated with the page cache; this fact is used to detect same-content memory pages on other VMs.

The self-reflection is caused when a target file is moved

and copied, because the contents are stored in the page cache. The attacker has to re-create a VM to prevent self-reflection, because memory is not cleared perfectly by simple rebooting [4], [5]. The re-creation of a VM, however, is the best way for clearing memory contents with zero, because the pseudo-physical memory is zero-cleared for memory isolation. An attacker can use this security countermeasure to his advantage.

4.3 Run-Time Modification Problem

Memory pages are modified more often than we expect. We call this phenomenon "run-time modification", which includes memory page swap-out, anonymous pages, ASLR (Address Space Layout Randomization), self-modifying code, and pre-loading. They cause false-negatives and false-positives on memory disclosure attack.

The most common modification is memory page swap-out on a victim's VM. At that time, memory deduplication is dissolved, and attacker cannot detect. However, the targeted process or file still exists on the process list (which is shown by command "ps" on Linux) or the list of open files (which is shown by command "lsof" on Linux) on a victim's VM. This causes a false-negative.

On the other hand, the attacker's memory is also swapped-out. In this case, the access time is delayed by the swap-in operation on the attacker's VM and causes a false-positive. Fortunately, this is solved by using the no-swap setting on the attacker's VM.

Anonymous pages do not change the contents of memory, but the status of a process or file is changed. Even if a process has terminated or a file becomes unused, the memory contents still exist as anonymous pages for a long time [4], [5]. Memory deduplication merges anonymous pages and causes a false-positive on the memory disclosure attack. It means that the attacker cannot know the status of application and file in general.

ASLR (Address Space Layout Randomization) is a computer security technique to make it difficult for an attacker to predict target addresses, which used on shellcode injection attacks. It changes the position of the base of code, libraries, heap, and stack for each process. Current operating systems have this mechanism as a default. The Linux kernel started to include ASLR in version 2.6.12, released June 2005. Even though it seems to decrease the effect of memory deduplication, the contents on most pages are unchanged by ASLR. An attacker need not care about ASLR.

Some operating systems have a pre-load mechanism in order to start applications quickly. Linux has a "readahead" system call, which populates the page cache with contents of a file before it executes. The target process and file is not listed on victim's VM but the attacker detects the memory contents. This can be considered as a false-positive, but the function is usually used just before an application is used. An attacker need not care about pre-loading.

Self-modifying code causes a false-negative, because the code alters its own instructions while it is executing. In general, attacker does not know the status of self-modifying code and cannot successfully carry out a memory disclosure attack. However, self-modifying code is not used in normal applications. An attacker need not care about it.

The mentioned run-time modifications make a memory disclosure attack difficult. However, it is a matter of possibility and the vulnerability still exists. Furthermore, the success of an attack also depends on the number of deduplicated pages. When the attacker gets many deduplicated pages, the possibility of successful attack increases. A case study is presented in Sect. 5.2.

4.4 Timing Problem

A memory scan type deduplication takes time to be carried out, because candidate pages are examined on being identical during a certain interval. Therefore, an attacker has to wait for a period of time. This period depends on the environment and the size of matching memory. If the period is too short, the prepared pages are not deduplicated by target pages on the victim's VM. If the period is too long, the attack leads to a false-negative or -positive, due to some noise caused by swap-out, anonymous-page, and so on. Usually, attacker has to decide on the suitable period in a real environment. This requires many attack trials and rebooting his VM. However, here trials are easy because rebooting VM is commonly-performed to save CPU power and the risk to be noticed is very low.

5. Experiments

In our experiments, the target of our memory disclosure attack consists of applications on a VM running Linux or Windows. We reveal that the attack can detect the security level of attacked operating systems, find of vulnerable applications, and confirm the status of attacked applications.

The memory disclosure attack uses the technique for the alignment problem, mentioned in Sect. 4.1, and self-reflection problem, mentioned in Sect. 4.2. The runtime modification problem mentioned in Sect. 4.3 occurs occasionally. We refer to the runtime modification problem in each case in this section. The timing problem is confirmed in Sect. 5.1.

We ran the experiments on a machine with an Intel Core2Quad 3.0 GHz processor and 8 GB of memory. The host OS was Debian squeeze with the standard Linux kernel 2.6.32 being augmented with KSM. The attacker's and victim's VMs were running on KVM (0.12.5). The guest OS is also same Debian squeeze.

5.1 Waiting Time for Deduplication

The memory disclosure attack must wait for memory 4 KB pages to be deduplicated. The period depends on the target environment and we measured it with several trials.

After one byte of data was written on a target page,

Table 1 Average write access time on 100 random 4 KB pages (400 KB) and 100 zero-cleared 4 KB pages (400 KB) (microseconds).

	After 1 min	After 3 min	After 5 min
Non-dedup memory	0.58	0.48	0.57
Dedup memory	10.61	12.76	17.1

the write access time was measured, because the page is re-created with first write request by COW, when the page is deduplicated. We compared the write access time of pages containing random data and zero-cleared data. Pages with random data are unique and are not merged with the stable tree of KSM. Their write access time is normal. Pages with zero-cleared data exist many times and are merged to stable tree of KSM. The write access time is delayed by COW.

Table 1 shows the average write access time on 100 random 4 KB pages (400 KB) and 100 zero-cleared 4 KB pages (400 KB) after certain period (1, 3, and 5 minutes) the pages are loaded on memory. The results show that a 1-minute wait time is enough. However, the access time on each page is not so clear, as shown in Fig. 2.

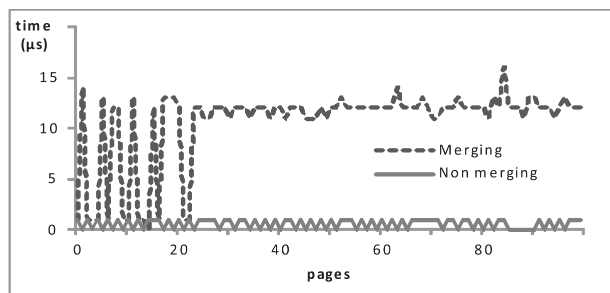
Although the write access time on non-deduplicated page is stable at all time periods, the deduplicated pages are different. Figure 2(a) shows that many deduplicated pages are not delayed. This is caused by non merged pages, which is confirmed by the status information on /sys/kernel/mm/ksm. The status of most pages becomes stable after 3 minutes. However, this occasionally changes depending on the load of the CPU. In this paper, we use 5 minutes wait time to prevent false-negatives on each trials.

5.2 Detected Security-Level

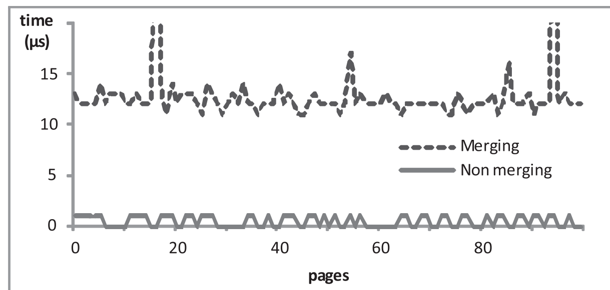
At the beginning of an attack, the attacker wants to know the security level on a target operating system. An attack is easy when the attacker knows that there are no running security tools. We tried to disclose the existence of security tools on a victim's VM. In our experiments, these applications are Snort, which is an IDS (Intrusion Detection System) on Linux, and Symantec Anti Virus on Windows. The security tools are executed at boot time.

The size of the Snort ELF file is 708,780 bytes, and the size of Symantec Anti Virus PE file is 1,777,248 bytes. The matching target pages of Snort and Symantec Anti Virus are 177 and 444 pages, respectively. The last page which is less than 4 KB is not the target of matching, because the tail of 4 KB is arbitrary contents. We measured the write access time of same-content pages on the attacker's heap memory, before and after the applications were invoked.

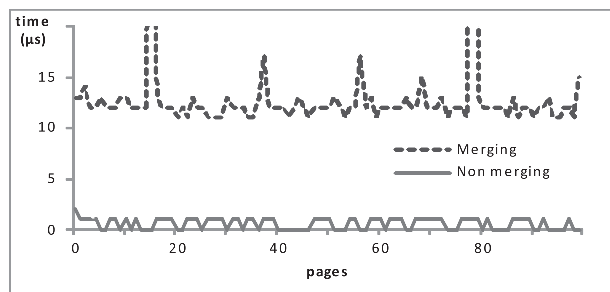
Table 2 shows the average access time of the same memory image of target applications on the attacker's memory. These results indicate that thresholds exist but they depend on the situation. Figure 3 shows the write access time to each page. Figures 3(a) and (b) show the results of Snort



(a) Write access time after 1 minute



(b) Write access time after 3 minutes



(c) Write access time after 5 minutes

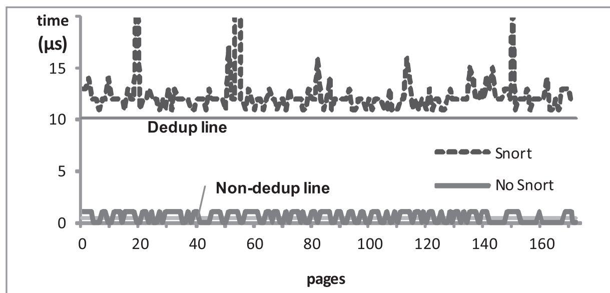
Fig. 2 Write access time on random and zero-cleared pages a certain period (1, 3, and 5 minutes) after the data are loaded.

Table 2 Average write access time of contests of Snort and Symantec Anti Virus (microseconds).

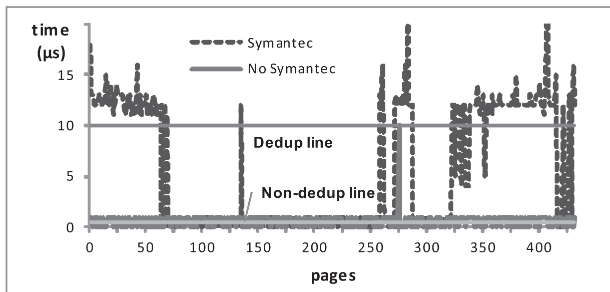
	Snort	Symantec Anti Virus
Not running	0.54	0.63
Running	14.16	5.59

and Symantec Anti Virus, respectively. The result of Snort is easy to distinguish. The write access time is clearly separated into running and not-running cases. Although there are some spikes in the case of running, the noise is little and negligible.

However, the result of running Symantec Anti Virus (Fig. 4(b)) shows many drops on page 71–259 and 262–272, which causes a false-negative on the attack. We guess these pages are not loaded from disk, because the binary of Symantec Anti Virus is very large and some parts of the binary are normally not used at run time. On the other hand, the result of not running Symantec Anti Virus shows the

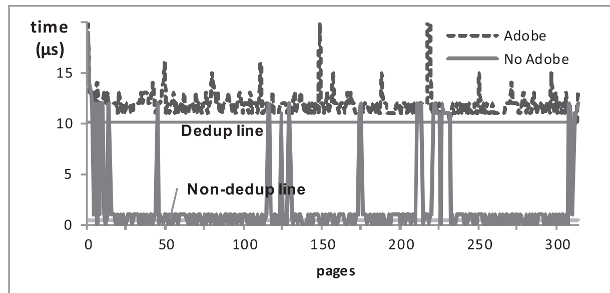


(a) Snort is running or not running.

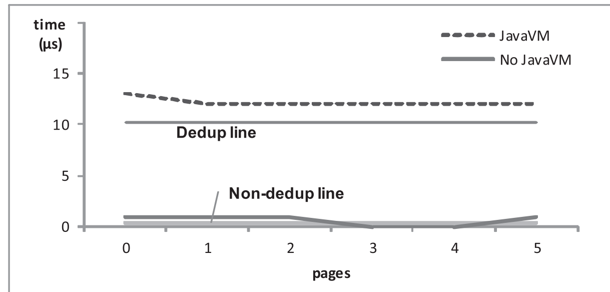


(b) Symantec Anti Virus is running or not running.

Fig. 3 Write access time on attacker's VM. (a) shows the result of Snort on Linux and (b) shows the resut of Symantec Anti Virus on Windows.



(a) Adobe Reader is running or not running.



(b) Java VM is running or not running.

Fig. 4 Write access time on attacker's VM. (a) shows the result of Adobe Reader on Windows and (b) shows the result of Java-VM on Linux.

Table 3 Average write access time of contests of Adobe Reader and Java-VM (microseconds).

	AdobReader	Java VM
Not running	1.79	0.67
Running	12.19	12.17

spikes on page 260–262, which causes a false-positive on the attack. Fortunately, the size of Symantec Anti Virus is large and the noise is compensated when the average is taken.

5.3 Detected Vulnerable Applications

After detecting the security level on OS, an attacker will search for vulnerable applications. We tried to detect Adobe Reader 10.0 on Windows and Java VM Version 6 Update 27 on Linux. The Adobe Reader has vulnerability (CVE-2011-061) to allow remote attackers to execute arbitrary code. The Java VM has vulnerabilities (CVE-2011-3548, CVE-2011-3521, CVE-2011-3554, and CVE-2011-3544) to be exploited through untrusted Java applets. The size of the Java VM is 25,634 bytes (6 pages), and the size of Adobe Reader is 1,289,624 bytes (332 pages). Java VM is constructed many libraries and the size of main binary is small.

Table 3 shows the average access time. The results show a clear difference between the case of not running and running. Figure 4 shows the write access time of each page. Figure 4(a) shows that there are sporadic spikes in the case of Adobe Reader not running. The Adobe Reader uses many pages and we can distinguish them from noise. On the other hand, Fig. 4(b) shows that the Java VM uses few pages, but they are clearly separated by the write access time. This case is easily distinguished, but the case of a few pages may cause a false-positive or false-negative. The attacker should use as large memory contents as possible.

5.4 Detection of the Status of Applications

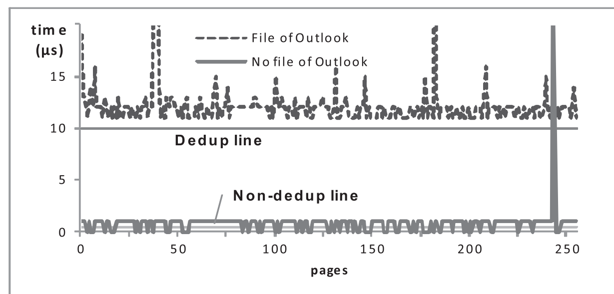
An attacker wants to know the status of an application in order to carry out an the attack. The memory disclosure attack is used in such a case, because it is not limited to executable binaries. For example, it detects whether an attached file is read from a mail reader and whether a browser visits a target home page on Linux. These applications open a file when they carry out some actions and the memory disclosure attack searches the file.

We try to detect whether Outlook Express reads a 1MB attached file and whether FireFox visits a given home page. The memory disclosure attack could detect a file when Outlook Express saves the file. When outlook accepts an email with an attached file and reads the email with the file name, the detection does not work well. It means that Outlook Express does not touch the contents of file before the file is saved.

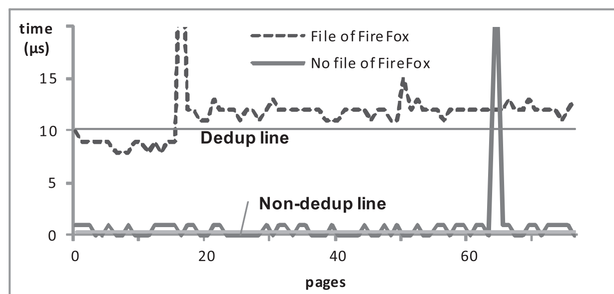
When FireFox visits a home page which includes a large file, the memory disclosure attack can detect the visit. However, currently popular home page does not use a large file. Even if a file is large, it is changed frequently. An accurate attack requires frequent update of the target file. An advanced point of this attack is that the attack does not care

Table 4 Average write access time of (microseconds).

	1MB attacked file	319KB file on HP
Not saved / Not downloaded	0.83	0.81
Saved / Downloaded	12.95	11.54



(a) Attached file in Outlook



(b) Downloaded file by FireFox

Fig. 5 Write access time on attacker's VM. (a) shows the result of Outlook on Windows and (b) shows the result on FireFox on Windows.

about network encryption. Even if the network is encrypted by TLS/SSL, the downloaded file is plain and detectable.

Table 4 shows the average write access time when a 1MB (250 pages) attached file on Outlook Express is saved and FireFox visits the home page of Gentoo Linux (http://www.gentoo.org/) and downloads a 318,659-byte (79 pages) figure file. The files are detected clearly. Figure 5 shows the write access time of each page. Both figures have spikes when there is no file. However the spike occurs only one time on each trial. It is easily recognized as noise.

6. Countermeasures

The attack has to issue a write access to the deduplicated pages. Even if the victim VM allocates memory image to read-only pages, the page is merged with the read-write page on the attacker's memory, because current memory deduplication does not care the permissions of pages. If memory deduplication is allowed only on read-only pages between virtual machines, the memory disclosure attack is prevented. However, in order to distinguish the permission of memory, memory deduplication has to know the information of page table which is managed by virtual machine monitor. Current memory deduplication does not inspect it. KSM is implemented as independent of virtual machine monitor and cannot get this information. Even if memory deduplication is integrated in virtual machine monitor, it is not easy to get the information of the page table because the page table for current virtual machine is managed by hardware; for examples, EPT (Extended Page Table) of Intel and NPT (Nested Page Table) of AMD. In order to know read-only permissions, a hardware mechanism to trace permission is required.

A covert channel of memory deduplication is mentioned in Satori [9]. Satori can prevent this attack because Satori is content-aware type deduplication and can recognize illegal matching. Satori also has a mechanism to refuse memory deduplication to prevent the exploit. These are strong defenses, but Satori requires para-virtualization on the guest OS and is not applicable to any OS. The mechanism also requires the OS and application to know the importance of data and manage the memory pages.

Overshadow [3] and SP3 [14] offer a mechanism to encrypt a VM's memory to prevent memory disclosure. The memory is encrypted by a different key on each VM. This completely prevents the attack, but ruins the effect of memory deduplication. If we use encryption to prevent a memory disclosure attack, the contents should be limited to important data and be made as small as possible.

This attack is also prevented, if the victim's OS uses obfuscation code to change every runtime memory image. We think the mechanism can be implemented by a sandbox or a binary loader. Then, an attacker cannot prepare an identical page for a target application. Even if the change is only the shift of the start point of memory image, it becomes difficult to predict the existence of an application. For example, GNU debugger "gdb" shifts 8 bytes of starting point of a binary, and we struggled to find the reason of this for a long time.

It is not good idea to include a delay in write access, because that destroys the merit of deduplication. Since memory access affects the performance of applications severely, few users want to use a slow memory-access environment. Furthermore, it is not clear how much delay time is suitable and how one randomly inserts a delay.

Tool sHype [10] offers a security mechanism to prevent a cross-VM attack. It does not allow VMs to run simultaneously, when the VMs have a conflict of interest. Namely, the policy of sHype defines that VMs which have a conflict of interest are scheduled exclusively. Our technique is not affected by such a security mechanism, because it does not assume a running victim VM. Even if a victim VM is not running, the memory contents are shared by memory deduplication and our proposed technique is applicable.

Memory sanitization seems to prevent a memory disclosure attack. However, it also works the other way. Memory sanitization helps an attacker to know that an application is running on a victim's VM. Because the cache and anonymous page are cleaned up, an attacker knows the exact launch and termination time of an application.

7. Applications of This Technique

The technique of memory disclosure attack is not limited to disclose contents on memory. It can be used for secret communication on multi-tenant cloud computing environment and for education on security.

7.1 Secret Communication

Our memory disclosure attack is able to detect a VM on a multi-tenant cloud computing environment. A VM can set a secret marker in its memory in order to announce its existence to other VMs unseen by an administrator. The other VMs detect the secret marker using the memory disclosure attack. This secret channel is not easily detected by an administrator, because an attacker can cause spurious memory deduplication.

Such a marker must be used to prevent a conflict of interest between VMs on a processor. This implements a user-level exclusive allocation control, which is offered by sHype [10] as an administrator control.

A marker may also help to collect VMs on a processor. This is also a kind of user level optimization. When a new VM does not find the marker, the user terminates the VM and recreates it till the VM is allocated on the same processor running the target VM. As a result, the VMs can reduce communication overhead.

7.2 Education for Security

Papers [4], [5] mention that data life time in memory is longer than we expect. Most users do not realize that memory keeps important data after an application terminates. The technique of our memory deduplication attack can be used to warn about the remaining data on memory. This is especially useful for education because we confirm the effect of using virtual machines.

For an administrator of cloud computing, memory deduplication attack may be used for live memory forensics. It can detect prohibited applications and files on other VMs. For example, insecure applications, a P2P downloader, or illegal data can be detected. The benefit of our method is that it only measures the write access time on the attacker's own VM. If an administrator can use live migration on any VM, he may move suspicious VMs to another CPU to narrow down the search for a particular VM containing the prohibited application or file.

8. Discussions

The proposed techniques are closely related to cross-VM side channel attacks [13] and the functionality of a virtual machine monitor.

8.1 Comparison with Cross-VM Side Channel Attacks

A cross-VM side channel attack on Amazon EC2 instances has been proposed in related work [13]. That attack uses the IP address to guess that a victim VM runs on the same CPU. Then, it monitors the behavior of the shared cache on a hyper-threading or multi-core CPU, and detects an application running on a victim VM. The attack is based on the covert channel of a shared cache on a hyper-threading or multi-core CPU.

That cross-VM side channel attack can be replaced with our technique, if a victim VM runs on memory deduplication offered by VMM running on a CPU. Our technique can also detect whether a victim VM runs on a VMM or not, if a marker exists in the memory of a victim VM as mentioned in Sect. 7.1. An application running on a victim VM is also detected by the technique proposed in Sect. 4, which does not require the shared cache of a hyper-threading or multi-core CPU.

The difference between a cross-VM side channel attack and our technique is the assumption that a target victim VM runs simultaneously. The cross-VM side channel attack monitors behavior of the shared cache. Therefore, it can be protected by sHype [10], which does not allow VMs to run simultaneously, when the VMs have a conflict of interest. Our technique is applicable whether a victim VM is running or not, because the delay in the write access time on a deduplicated page is caused by COW, which occurs independently of the status of the victim VM.

The deficiencies of both techniques are that they cannot determine an exact VM instance when multiple VMs are running on a CPU or a VMM. Both techniques can determine only the existence of an application on a VM on a CPU. This is caused by a kind of semantic gap which is lack of knowledge of the software implementation. However, the information is still useful for an attacker, because the attacker may learn about the existence of a vulnerability.

8.2 Virtual Machine Monitor and Proposed Technique

Our proposed technique can be applied on a VMM, if memory deduplication covers the memory of a VMM. However, most implementations of memory deduplication do not cover the memory of a VMM, because the aim of memory deduplication is to reduce the memory consumption of a VM with identical pages. In general, the memory of a VMM is treated as another isolated layer. Furthermore, some implementations of memory deduplication require special treatment. For example, KSM requires madvise() to know the region of memory deduplication. This means that not all memory is target of memory deduplication. KVM is customized to use KSM with madvise(), and memory deduplication is applied on the memory of all VM instances.

On the other hand, most VMMs can peek into the memory of a guest OS (e.g., XenAccess on Xen). This means an attacker does not need to use our technique, if the attacker obtains control of a VMM. Therefore, the memory forensic tool "Volatility" [15] can be applied on a VMM to obtain information on the guest OS. It can designate a target VM accurately and tell which VM runs a targeted application.

This manipulation, however, requires a privileged-level exploit. On the other hand, our proposed technique is implemented as a user level-application on an attacker's VM. It does not require special privileges on the VMM.

9. Conclusions

This paper describes a memory disclosure attack on another VM that uses memory deduplication. The attack measures the write access time on pages that are re-created by COW (Copy-On-Write) of memory deduplication.

The attack seems to be easy to implement but it has some implementation issues caused by memory alignment, self-reflection between page cache and heap, and run-time modification (swap-out, anonymous pages, ASLR, preloading mechanism, and self-modification code) problems. The problems cause false-positives and false-negatives, but most of them are avoidable with techniques presented in this paper.

In our experiments on KSM (kernel samepage merging) with the KVM virtual machine, the attack could detect the security level of attacked operating systems (Symantec Anti Virus on Windows and Snort on Linux), find vulnerable applications (Adobe Reader which has vulnerability of CVE-2011-061 and Java VM which has vulnerability of (CVE-2011-3548, CVE-2011-3521, CVE-2011-3554, and CVE-2011-3544), and confirm the status of attacked applications (saving of attacked file with Outlook Express on Windows and downloading of a file with Firefox on Linux). The attack does not care about network encryption because it attacks the downloaded and decrypted contents on memory. These results show the possibility of attacks on multitenant cloud computing environments.

Fortunately, the attack can be prevented by some countermeasures: restriction of deduplication to read-only pages which requires the virtual machine monitor to check the page table entries, or encryption of information data. These countermeasures, however, decrease the performance of deduplication. The analysis of the trade-off between security and performance constitutes future work.

Acknowledgments

We would like to thank to the members of the Research Institute for Secure Systems (RISEC) of the National Institute of Advanced Industrial Science and Technology (AIST) for their discussions.. Especially our group leader Professor Etsuya Shibayama and Dr. Yutaka Oiwa gave us a number of helpful comments.

References

[1] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," Linux Symposium, pp.19–28, 2009.
[2] E. Bugnion, S. Devine, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," Symposium on Operating Systems Principles (OSDI), pp.143–156, 1997.
[3] X. Chen, T. Garfinkel, E.C. Lewis, P. Subrahmanyam, C.A. Waldspurger, D. Boneh, J. Dwoskin, and D.R.K. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.2–13, 2008.
[4] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," USENIX Security, pp.22–22, 2005.
[5] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," USENIX Security, pp.321–336, 2004.
[6] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," Operating Systems Design and Implementation (OSDI), pp.309–322, 2008.
[7] J.A. Halderman, S.D. Schoen, N. Heninger, W. Clarkson, W. Paul, J.A. Calandrino, A.J. Feldman, J. Appelbaum, and E.W. Felten, "Lest we remember: Cold boot attacks on encryption keys," USENIX Security, pp.45–60, 2008.
[8] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channles in cloud service deduplication in cloud storage," IEEE Security & Privacy, vol.8, no.6, pp.40–47, Nov. 2010.
[9] G. Miło's, D. Murray, S. Hand, and M.A. Fetterman, "Satori: Enlightened page sharing," USENIX Annual Tech, 2009.
[10] "sHype: Secure hypervisor approach to trusted virtualized systems," IBM Research Report RC23511, 2005.
[11] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest OS," European Workshop on System Security (EuroSec), 2011.
[12] K. Suzaki, T. Yagi, K. Iijima, N.A. Quynh, C. Artho, and Y. Watanabe, "Moving from logical sharing of guest OS to physical sharing of deduplication on virtual machine," USENIX Workshop on Hot topics in Security (HotSec), 2010.
[13] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," Proc. 16th ACM conference on Computer and Communications Security, pp.199–212, 2009.
[14] J. Yang and K. Shin, "Using hypervisor to provide application data secrecy on a per-page basis," Conference on Virtual Execution Environments (VEE), 2008.
[15] Volatility, https://www.volatilesystems.com/default/volatility/
[16] C.A. Waldspurger, "Memory resource management in VMware ESX server," Symposium on Operating Systems Principles (OSDI), pp.181–194, 2002.
[17] A. Warner, Q. Li, T.F. Keefe, and S. Pal, "The impact of multilevel security on database buffer management," 4th European Symposium on Research in Computer Security (ESORICS), 1996.

Kuniyasu Suzaki is a senior researcher at the Research Institute for Secure Systems (RISEC) of the National Institute of Advanced Industrial Science and Technology (AIST). He got Master of Engineering from Tokyo University of Agriculture in 1990 and Technology, and received his Ph.D. in Computer Science from University of Tokyo.

Kengo Iijima graduated from University of Tsukuba in 1998 and joined the National Institute of Advanced Industrial Science and Technology (AIST) in 2002.

Toshiki Yagi graduated from University of Tsukuba in 1999 and joined the National Institute of Advanced Industrial Science and Technology (AIST) in 2005.

Cyrille Artho is a researcher at the Research Institute for Secure Systems (RISEC) of the National Institute of Advanced Industrial Science and Technology (AIST). He has completed his Ph.D. at ETH Zurich, Switzerland, in May 2005. From June 2005 to March 2007, he worked as a post-doctoral researcher at the National Institute of Informatics in Tokyo, Japan.