A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines

Jens Lindemann and Mathias Fischer
Security and Privacy Group
Department of Computer Science
University of Hamburg
{lindemann,mfischer}@informatik.uni-hamburg.de

ABSTRACT

Virtualization offers the possibility of hosting services of multiple customers on shared hardware. When more than one Virtual Machine (VM) run on the same host, memory deduplication can save physical memory by merging identical pages of the VMs. However, this comes at the cost of leaking information between VMs. Based on that, we propose a novel timing-based side-channel attack that allows to identify software versions running in co-resident VMs or on the host. Our attack tests for the existence of memory pages in co-resident VMs that are unique among all versions of the respective software. Our evaluation results indicate that with few repetitions of our attack we can precisely identify software versions within reasonable time frames.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Virtualization and security**; *Vulnerability scanners*; • **Computer systems organization → Cloud computing**;

KEYWORDS

security, side-channel attack, virtualization, cloud computing

ACM Reference Format:
Jens Lindemann and Mathias Fischer. 2018. A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines. In *SAC 2018: SAC 2018: Symposium on Applied Computing , April 9–13, 2018, Pau, France.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3167132.3167151

1 INTRODUCTION

Our society relies more and more on the availability of Internet services. These services are increasingly provided by virtualised servers operated by cloud providers in their server infrastructures.

Attacks on cloud providers take place all the time. However, only few of them cause real damage. Mostly, such successful attacks are prepared by extensive reconnaissance in which attackers actively scan their targets. Obtaining information about the software configuration of other VMs, other users on the same VM, or the host operating system allows them to specifically exploit security vulnerabilities known to exist in the identified software versions. Conducting vulnerability scans in a provider's infrastructure from virtual machines is forbidden by the acceptable use policies of many cloud service providers [9]. Furthermore, when such scans are conducted via the network, they can be easily detected, e.g. by an Intrusion Detection System (IDS). However, so-called side-channel attacks, which do not use standard communication paths, can reveal information on a target system by evading standard detection mechanisms at the same time.

Such side-channel attacks are especially a danger in virtualised environments (e. g. [6, 7]), in which multiple virtual machines share the same hardware. Suzaki et al. [15] have shown that it is possible to detect a software running within another VM on the same host by writing a copy of the binary into memory. This copy will then be deduplicated by the hypervisor. When this deduplicated copy of the binary is overwritten, this will take longer than overwriting random and non-deduplicated data. Thus, this gives an attacker the information that another copy of the binary is present on the host. A vulnerable software version identified in another VM does not lead to a direct attack path, as the IP address will normally be unknown and would have to be obtained using another method. If the IP address is known to the attacker, however, knowing the software version being executed enables the attacker to launch an attack specifically targeted at vulnerabilities in this version. Also, a vulnerable hypervisor version or a vulnerable software version being executed by another user on the same VM will be directly attackable.

The main contribution of this paper is a novel side-channel attack based on memory deduplication that allows a curious attacker controlling a VM to gain information about the software configuration of (a) other co-located VMs, (b) other users of the same VM or (c) the host operating system. Our attack is based on identifying memory pages of a software version that are unique across all other versions of that software. Once such signature pages have been identified, their existence in co-resident VMs can be easily tested by loading just these pages into the memory of the attacker VM. Thus and contrary to related work, our attack does not presume to load a software binary completely.

Our evaluation results indicate that we can distinguish software versions to the precision of the distribution patch level. Our attack is faster than other attacks that test whether individual pages have been deduplicated as timing differences. Our attack requires fewer measurements to detect versions that have a large number of unique pages. For example, to achieve an accuracy of more than 95 % in the detection of the software version, at least ten measurements with a

SAC 2018, April 9–13, 2018, Pau, France

minimal signature size of four pages are necessary. However, such an attack would take at least 1.8 hours.

As previous work did not analyse the impact of differences in software versions and the underlying operating system, we present an analysis of the effectiveness of such an attack across different software and operating system versions. We found that software binaries of the same upstream release share almost no common pages across different Linux distributions. Furthermore, we have analysed the potential for memory savings by deduplicating memory pages containing executable code across different OS and software versions.

The remainder of this paper is structured as follows: In Section 2 we discuss background information and related work. Section 3 describes our side-channel attack. In Section 4, we evaluate effectiveness and efficiency of the attack. Section 5 presents potential countermeasures and Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

In this section we first explain the concept of memory deduplication as well as its implementation in popular hypervisors. Then, the attacker model is presented. Finally, we will discuss related work.

2.1 Memory Deduplication

Memory deduplication is a technique for saving physical memory on a computer. It is often deployed on hosts for virtual machines as a cost-saving measure. The memory of a computer is organised by the operating system as a set of memory pages $\mathcal{M}$. Typically, the size of a memory page $p_i \in \mathcal{M}$ is 4 096 bytes. Every memory page resident in physical memory will consume these 4 096 bytes. Memory deduplication takes advantage of the fact that there are often sets D_i of multiple identical pages $p_i = p_j \in \mathcal{M}$. To save memory, all but one page $p_m \in D_i$ will be removed from the physical memory and all memory mappings $\forall p_i \in \mathcal{M} : p_i = p_m$ updated to point to p_m instead. Subsequently, when a page $p_i \in D_i$ is to be changed, the deduplication mechanism copies it to a different memory region so that it can be modified without affecting the other copies of the page.

Note that only pages actually resident in memory can be deduplicated, whereas pages that are swapped out cannot be deduplicated. This implies that it may not be possible to detect some pages of a file by means of a deduplication side-channel attack, despite the file being loaded into the virtual memory of the host.

In the following, we will describe the memory deduplication mechanisms of the popular KVM, Xen and VMWare ESXi hypervisors.

KVM. The KVM hypervisor is built into the Linux kernel and uses the “Kernel Samepage Merging” (KSM) technique [1] for memory deduplication. KSM automatically scans the memory pages of virtual machines for identical pages, which are then deduplicated. The guest OSs do not have to be modified for KSM. Deduplication is performed periodically dependent on memory load on the host. The interval between scans and the number of pages scanned per interval can be configured. The configuration can automatically be tuned according to the current memory usage using the ksmtuned daemon.

Xen. The Xen hypervisor provides a mechanism for sharing memory pages between VMs [4]. However, no mechanism for automatically identifying such pages is provided as part of the hypervisor, so that this feature is of little use in practice.

However, mechanisms that enable live deduplication in Xen have been developed by researchers. One such mechanism is the Difference Engine proposed by Gupta et al. [5]. Similar to KSM, it periodically scans the memory for shareable pages. It supports deduplication of similar, but not necessarily identical, pages, leading to higher memory savings compared to techniques that work on identical pages only.

Another mechanism is Satori [11]. Instead of periodically scanning the full VM memory for shareable pages, it checks whether a page can be deduplicated when it is being loaded. However, it requires modifications to the guest OS.

VMWare ESXi. VMWare ESXi uses its own deduplication mechanism, which has been described by Waldspurger [16]. Similar to the KSM mechanism used by KVM, the memory of guest VMs is regularly scanned for duplicate pages to deduplicate these. Modifications to the guest OS or the disk images used by the VM are not necessary.

2.2 Attacker Model

Our assumptions about the attacker’s capabilities are as follows: A memory deduplication side-channel attack takes place on a host h that hosts a set of virtual machines M. We denote the set of all versions of an application i as A_i. Individual versions are denoted as $a_i^v \in A_i$, where v is used as a version identifier. Each virtual machine $m_k \in M$ runs a set of application versions, which are returned by $apps(m_k)$. The attacker controls at least one virtual machine $m_a \in M$. The attacker can only observe the network traffic of m_a, not that of h or any other VM $m \in M \setminus \mathrm{m_a}$. The attacker’s intention is to determine a specific version $a_i^v \in A_i$ running outside their scope of control.

There are three possible attack scenarios:

- **Inter-VM.** The attacker is trying to determine a_i^v of an application A_i running on another virtual machine $m_v \in (M \setminus m_a)$
- **Intra-VM.** The attacker does not have root access to $\mathrm{m_a}$ and is trying to determine a_i^v of an application A_i being executed on m_a by another user.
- **VM-to-host.** The attacker is trying to determine a_i^v of an application A_i running on the host operating system of H.

For sake of clarity, we will concentrate on describing inter-VM attacks in the following. The mechanisms of intra-VM and VM-to-host attacks are identical. Intra-VM attacks will work on all hosts where inter-VM attacks are possible. Whether a VM-to-host attack can be performed on a host depends on whether the deduplication mechanism deduplicates pages of the host OS in addition to those of the VMs.

While the operator of a VM host trying to exploit a security vulnerability in software of a guest VM is the worst-case attacker, we do not consider Host-to-VM attacks. As the host has full access to the memory of all VMs on the host, its operator has a much easier attack path than a memory deduplication side-channel attack. Furthermore, they would also know how to communicate with an affected VM without having to find out the IP address using a separate side-channel.

2.3 Related Work

Data deduplication is similar to memory deduplication, but aims to save disk space by deduplicating copies of identical data in storage. It can be very effective (savings of 70 to 80 percent) when applied to images of similarly configured VM images [8, 10]. However, its effectiveness is reduced for heterogeneous software configurations on the VMs [8]. Timing side-channels also exist in data deduplication. They can reveal whether a file (or even a part of it) is already present on a storage service through timing differences caused by copy-on-write [6] or non-uploading of file contents [12]. Researchers have proposed Message-Locked Encryption as a countermeasure [2, 14].

Bosman et al. [3] demonstrate a JavaScript-based **memory deduplication side-channel attack** on the Microsoft Edge browser running on Windows 8.1 and 10, which use memory deduplication by default. Their attack does not require a virtualised environment, but targets end-user computers. The authors show that it is feasible to read arbitrary data from the target computer's memory.

Irazoqui et al. [7] describe an approach to detect the version of a cryptography library executed on a co-resident VM. They make use of a Flush+Reload attack on functions characteristic to a library. This leads to a difference in reload time if the function has been called in another VM after the attacker has flushed it from the cache. For the attack to work, the page containing the attacked function needs to be deduplicated between the attacker VM and the victim VM. While their attack has a similar aim as ours, it uses a different technique that requires manual analysis of the attacked libraries to find a suitable function. Furthermore, as their attack targets a single function in the library, it will be unable to distinguish versions in which the analysed function identical. This implies that different functions may have to manually be found to distinguish different pairs of versions.

Xiao et al. [17] show that memory deduplication can be used to establish a covert channel for communication between two (collaborating) co-resident virtual machines. Furthermore, they show that memory deduplication can be used to monitor the integrity of a VM's kernel from the outside.

Suzaki et al. [15] first described a side-channel attack exploiting timing differences caused by the KSM deduplication mechanism used in KVM. They demonstrate that it is possible to detect applications running in a co-resident VM. However, they only analyse a single version of each tested application. The authors do not analyse whether it is possible to tell different versions of an application apart. They used the full binary as a signature, ignoring whether pages may also be present in other versions of the applications or even other parts of the system.

Owens and Wang [13] describe an approach to detect the operating system running inside another virtual machine hosted on the same VMWare ESXi host through a memory deduplication side-channel attack. They generate their signatures by setting up the targeted OS versions, capturing memory images of the running system and then filtering out the memory pages unique to that OS version. However, their approach was only tested on four different major releases of Windows and two of Ubuntu Linux. The impact of the frequently published patches for these operating systems on the accuracy of their detection mechanism was not evaluated.

In **summary**, most other side-channel attacks on memory deduplication concentrate on either revealing data in the memory of another VM or on establishing a covert communications channel between two VMs. While some approaches are concerned with detecting the presence of applications, they do not thoroughly study detecting specific versions. The work of Owens and Wang [13], who aim to detect versions of operating systems, is the closest to ours.

3 MEMORY SIDE-CHANNEL ATTACK

Our memory deduplication side-channel attack is based on timing measurements and can reveal whether pages characteristic for a software version have been deduplicated. In the following, we will describe the general approach an attacker would take to identify software versions running in other VMs. We will also describe how to find characteristic memory pages that can serve as a signature for a specific software version.

3.1 Attack Procedure

Memory deduplication opens up a timing side-channel that can reveal to an attacker that a memory page holding a certain content is present on the host, e. g. within another virtual machine. A deduplicated page needs to be copied before it can be modified. Thus, there is an additional delay in modifying such a page compared to modifying a non-deduplicated page. This delay can be used to detect the presence of applications [15] or other data [3] in other VMs on a host. Note, however, that it will not allow an attacker to find out in which particular other VM the application is running.

We define $pages(a_i^v)$ to return all pages of a_i^v excluding duplicate pages within the binary and pages containing only zero or one bits. Each virtual machine $m_j \in M$ is running a set of applications R_j. An attacker is interested in whether an application version is present in another VM, i. e. $a_i^v \in apps(M \setminus m_a)$. We define $pages(m_j)$ as the set of all memory pages of a VM m_j, i. e.

$$pages(m_j) \supseteq \bigcup_{a_i^v \in R_j} pages(a_i^v) \qquad (1)$$

The attacker first needs to establish a deduplication and a non-deduplication baseline. To obtain the non-deduplication baseline, the attacker fills a number of memory pages equal to the number of pages they wish to test with random data, so that

$$pages(m_a) \cap \{\cup_{m \in M \setminus (m_a)} pages(m)\} = \emptyset \qquad (2)$$

It can be assumed that randomly-generated pages do not get deduplicated as it is extremely unlikely that an identical copy is present on the host or in another VM. The attacker then measures the time it takes to overwrite these pages as a baseline for non-deduplicated pages.

The assumption is that we can identify a particular application version based on a subset of pages of that application version a_i^v that are unique across all different versions of it. We refer to this subset of pages as a signature $sig(a_i^v)$ (cf. Sect. 3.2 for details on signature derivation). The attacker writes the signature of an application they believe to be present in another VM to the memory of their

VM m_a. If another VM m_v is executing a_i^v, this implies $\{\mathcal{M}_a \cap \mathcal{M}_v\} \supseteq sig(a_i^v)$, which means that these pages can be deduplicated. The attacker then needs to wait for deduplication to take place. Afterwards, the attacker modifies the pages that serve as signature and measures the time needed for overwriting exactly these pages.

This measurement can be compared to the baselines. A threshold for classifying measurements into deduplicated and non-deduplicated needs to be determined. If the measurement is significantly higher than the non-deduplicated baseline and close to the deduplicated baseline, the attacker can infer that the pages were most likely deduplicated, so that another copy of them as part of application version a_i^v is present in another VM. However, if the measurement is very close to the non-deduplicated baseline, the pages have not been duplicated and have been modified directly. This could mean that another copy of the pages was indeed not present on the host, but there is a small probability that a copy of the pages *is* present on the host, but has not been scanned by the deduplication mechanism yet, e. g. due to the deduplication mechanism being configured to only activate itself when the memory of the host is almost full. An easy and naive classification rule would be to use the mean of the two baselines as a threshold, which works well enough if multiple pages are being measured at once (cf. Sect. 4.7).

To use this side-channel to detect the presence of application version a_i^v on a host, an attacker would act as follows:

(1) Establish *baselines* by writing $length(sig(a_i^v))$ pages containing random information to the memory of m_a and measuring the time it takes to overwrite this random information (non-deduplicated baseline). Furthermore, write two copies of randomly generated pages into the memory of m_a and overwrite one of the copies (deduplicated baseline). The baselines should be based on multiple measurements.
(2) Determine the *classification threshold* based on the baselines obtained in the previous step.
(3) *Write* $sig(a_i^v)$ into the memory m_a.
(4) *Wait for deduplication* to happen. The correct waiting time depends on the configuration of the host's deduplication mechanism.
(5) *Overwrite* the signature, while *measuring the time* this operation takes to complete.
(6) *Repeat* steps 2 to 4 until a sufficient number of measurements has been taken.
(7) Calculate the mean of the measurements taken and *compare* it to the classification threshold.

If the attacker is not interested in particular pages, but in identifying pages that are unique to an application version (aka signature), the full set should be written at once. The timing differences observed between overwriting deduplicated and non-deduplicated pages will be more pronounced if multiple pages are being checked at the same time. Thus, an attacker can identify a program running on another VM with fewer measurements. This implies that the signatures for a software version should consist of as many pages unique to this version as possible.

As it is necessary to repeat the measurements several times, and each measurement comes with a delay, the attack will be relatively slow. However, if the signatures to be checked are disjunct, i. e. they do not contain any pages that are also present in other signatures, multiple signatures can be checked in parallel.

The configuration of the deduplication mechanism will have an impact on the effectiveness of any memory deduplication side-channel attacks: If the interval between scans is long, potential attackers would be slowed down at the cost of decreased memory savings. If, on the other hand, the activation threshold is set relatively high, attacks will not be possible at all as long as the host's memory load remains beneath the threshold.

3.2 Signatures for the Timing Side-Channel

To be able to reliably detect a software version, we need to build signatures for each version. A signature should contain only pages unique to the respective version, as including pages that can also be found in other versions may lead to false classifications. This also holds true for pages of completely different applications. Due to the size of a page, it is however very unlikely that an identical page can also be found in a different application.

To derive a signature for a version of an application binary, we start with all its pages and then remove the following types of pages:

(1) **Internal duplicates** because a duplicate page within the signature itself would be sufficient to trigger deduplication of these pages.
(2) **Pages containing only zeroes or ones**, as another copy of these is very likely to be present on the host even if the surveyed application is not being executed in another VM at all.
(3) **Pages present in other versions**, as these are unsuitable for distinguishing the version.

Any of these pages can be deduplicated without the probed version being present in another VM. Therefore, they must be removed to avoid false positives. In summary, the signature for an application version a_i^v is generated as follows:

$$sig(a_i^v) = pages(a_i^v) - \bigcup_{a \in \{A_i \setminus a_i^v\}} a \qquad (3)$$

Our approach for deriving signatures is similar to the one for detecting operating systems by Owens and Wang [13]. In their approach, they capture memory images of different OSs while executing them. Then, they derive signatures from these that contain pages unique among their OS dataset. Similar to their work, we aim to find memory pages that are unique to an application instead of an OS version to use them as signature. However, we consider the pages of the application binary only and can ignore all pages containing application data pertaining to the runtime state. Thus, any pages that do not contain executable code, such as data pages, are ignored. These pages may differ between two instances of an identical binary, e. g. due to a different runtime state, or may be identical for two different versions of a binary, e. g. due to a similar runtime state saved in a memory structure that has not been changed between versions. Thus, they are not well-suited for detecting the application version being executed.

Focusing on only the pages of the application binary renders our technique much more efficient compared to the work of Owens and Wang. These binary pages are the only pages that can safely be assumed to reside in memory on all systems executing it. Moreover, binaries need not be executed to generate signatures in the form of unique memory pages. Thus, in the following, we can use these signatures to specifically test via our side-channel attack described in Section 3.1 if there is an application running that matches the signature, i. e. contains the signature pages.

In the next section, we summarise our findings in evaluating the proposed side-channel attack.

4 EVALUATION

In this section, we first present the tools and datasets that we have created for our experiments. Also, we describe experiments that indicate that a timing side-channel exists that can reveal whether pages of an application are present within another VM. We then present experiments that indicate that versions of the same application are different enough from each other to detect them with this method. Furthermore, our experiments indicate that releases of the same upstream version from different distributions can easily be distinguished, too. We also analyse the impact of changing the page size on the deduplication of pages containing executable code. Finally, we present an analysis on the complexity of our attack.

4.1 Signature Generation and Measurements

To derive signatures and enable the experiments described in the reminder of this paper, tools have been developed that allow the automatic comparison of a large number of versions of a binary[1].

To analyse a software, we first need to obtain its different versions. Then, the main binary has to be extracted from the downloaded packages. To this end, shell scripts have been developed that can extract the binaries from RPM and deb packages. The scripts can easily be adapted to each application and distribution and will then process a large range of versions, as the location and name of a binary within the package rarely changes. The scripts will place the extracted binaries into a directory structure that can be processed by our analysis tool.

The main analysis tool is written in Java and can process ELF binaries, but can easily be extended to other executable formats as well. It supports two modes of analysis: First, all versions of a binary can be **compared** with each other to determine the number of matching pages between each pair of versions. Results will be output as a csv file. Second, the software can output the **signature** for each version, which will contain all pages unique to that version (cf. Sect. 3.2). Statistics about the signatures will also be created and saved in a csv file. These include the signature sizes, the number of internal duplicates, the number of pages that can also be found in other versions, and the number of pages containing only zeroes or ones. The page size used by the tool can be configured freely.

Furthermore, two C programs have been developed to perform timing measurements. The first one loads signature pages into memory and measures the time it takes to overwrite them after a specified amount of time has passed. Measurements will be output to the console and logged into a file. The second one loads signature pages into memory aligned to page boundaries to enable experiments that do not use a running executable.

[1]The code of our tools is available at https://github.com/jl3/memdedup-app-detection.

4.2 Datasets

We created three datasets for our experiments: The first one contains all Apache web server releases for Debian on the *x86_64* platform and the second one all SSH daemon (*sshd*) releases. The third dataset contains releases of *sshd 7.5p1* for different distributions. Our datasets consist of the following application versions:

- The **Apache-Debian-x86_64** dataset consists of all 131 Debian releases of *Apache* available for the x86-64 platform and includes versions from 2.2.11-5 to 2.4.25-3.
- The **sshd-Debian-x86_64** dataset consists of all 185 Debian releases of *sshd* available for the x86-64 platform and includes versions from 4.2p1-7 to 7.5p1-5.
- The **sshd-crossdist** dataset consists of 11 package versions of *sshd 7.5p1* from Arch Linux, Debian, Fedora, OpenMandriva and Ubuntu. Multiple revisions are included for Debian (5) and Fedora (3).

Only the main executable of each application (*httpd* for Apache, *sshd* for the SSH daemon) is contained in our datasets. For the surveyed applications, these are typically the only executables that will be running as a daemon at all times. While both applications include additional executables (e. g. *ssh-keygen* for generating SSH keypairs), these would normally not be running long enough for the described attack to be possible. In case of packages containing multiple executables to be run constantly (e. g. as a daemon), all these executables should be included when generating signatures.

The packages for the Debian-based datasets were obtained from the snapshot archive[2], which provides historic package versions. A similar repository of old package versions is available for Fedora[3], which retains old versions of binary packages and keeps them publicly available.

Unfortunately, most distributions, among them openSUSE, OpenMandriva, Ubuntu and Arch, provide only very recent versions of the binary packages. This makes it hard to create a dataset that can be applied to other distributions as well. For the cross-distribution dataset, due to the lack of older versions of binary packages for many distributions, we had to use the recent upstream version 7.5p1 of sshd, which was available for download for a variety of distributions at the time of dataset creation.

4.3 Feasibility of the Side-Channel

In the following, we will show that a timing side-channel in memory deduplication exists that can be used to reveal the presence of memory pages in another VM or on the host.

For the experiments described in this section, two virtual machines m_a and m_v are used. The host is an Intel Core i7-4790 with 16 GiB RAM running KVM and KSM on Fedora 26. First, a number of pages is loaded into the memory of m_v. Then, the same pages are loaded into m_a. After that, we wait for deduplication to take place and overwrite the pages in the memory of m_a, measuring the time this takes.

Figure 1 shows the write times to sets of non-deduplicated and deduplicated pages depending on the number of pages in the respective application. All results in the figure are averaged over 2 000

[2]http://snapshot.debian.org

[3]https://koji.fedoraproject.org/koji/

measurements each. In the non-deduplication case, the pages on m_a and m_v are of identical size, but have different contents, so that no deduplication can take place. In the deduplication case, the pages on m_a and m_v are identical, so they can be deduplicated. It can be seen that write times to deduplicated pages are higher

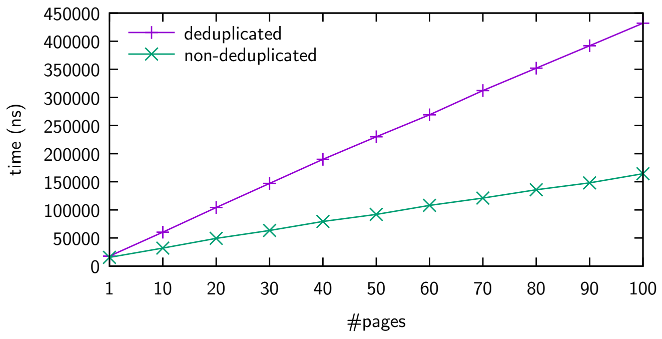


Figure 1: Average write times (nanoseconds) to n deduplicated/non-deduplicated pages

than to non-deduplicated pages. For both types of pages, write time increases linearly with the number of pages overwritten. The gap in write times between non-deduplicated and fully deduplicated sets of pages increases when writing to a larger number of pages. This implies that when we measure the time to overwrite a larger number of pages at once, it will be easier to determine whether these pages have been deduplicated previously.

4.4 Cross-version similarities

We now present our analysis on cross-version similarities in the *Apache-Debian-x86_64* and *sshd-Debian-x86_64* datasets.

For that, we directly compare each version to every other version available. This direct comparison shows how many pages are identical among two specific versions. The more pages are identical, the harder it will be for an attacker to distinguish these versions from each other using our side-channel attack. However, a larger number of identical pages also implies that deduplication can save more memory.

Furthermore, we determine the number of pages that can be used as signature for each application version in our datasets. We also analyse how many pages have not been used for deriving signatures (cf. Section 3.2).

Figure 2a shows the number of matching pages between all versions for the *Apache-Debian-x86_64* dataset. Figure 3a shows the number of matching pages between all versions in the *sshd-Debian-x86_64* dataset. Each row and column corresponds to one version, namely from old versions on the top left to new versions on the bottom right. The colour at the intersection of the row corresponding to version v_r and the column corresponding to version v_c represents the number of pages in the binary of v_r that are also present in the binary of v_c. The bright diagonal line running from the top left to the bottom right represents the comparison of a version with itself ($v_r = v_c$) and shows the size of the respective version's binary in pages. It can be seen that newer versions of the binaries are larger than older ones.

The results indicate that some versions form clusters, whose binaries are relatively similar to each other. For the *Apache* dataset, these clusters correspond to multiple Debian revisions of an upstream version. An example of a range of versions whose binaries are very similar to each other is 2.2.16-1 to 2.2.16-6. However, the backport revision 2.2.16-4-bpo50+1 is completely different from neighbouring versions. Further examples of similar binary versions are 2.2.22-6 to 2.2.22-10 and 2.4.10-2 to 2.4.10-6. The latter cluster is also contained within the larger cluster of versions 2.4.10-2 to 2.4.10-11, which all share at least seven pages with each other (with the exception of 2.4.10-10+deb8u8). Figure 2b gives a detailed view of this cluster and shows how many of the pages in version 2.4.10-10+deb8u5 can also be found in each neighbouring version.

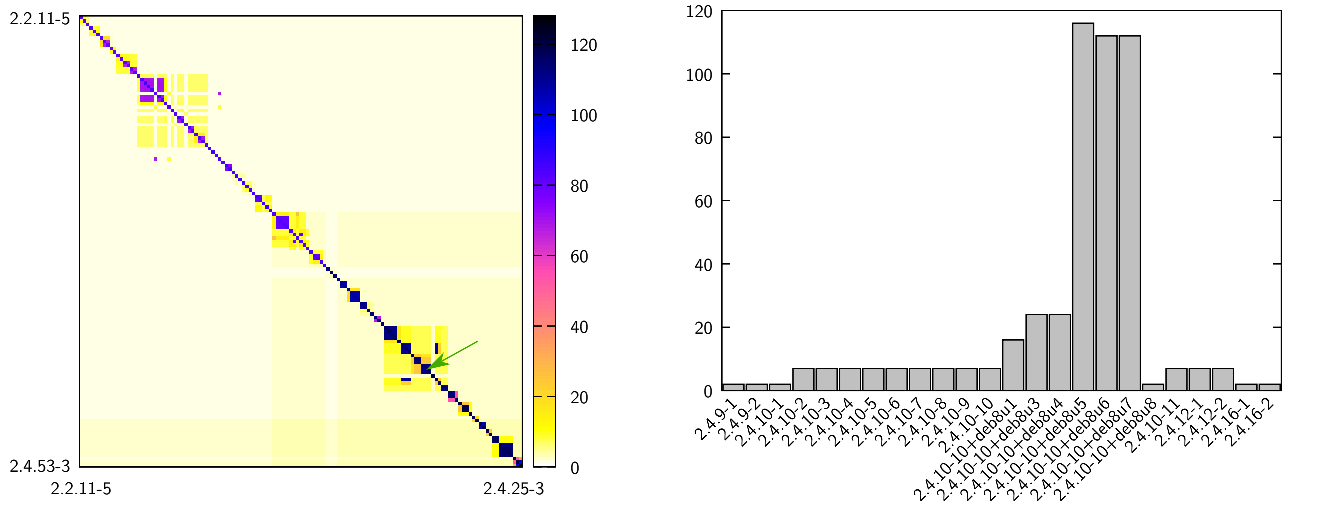
The clusters in the *sshd* dataset are similar in size and are also restricted to versions that have been released close to each other. Interestingly, while most of these clusters also contain only different Debian revisions of the same upstream software version, the *sshd* dataset – unlike the *Apache* dataset – shows some examples of clusters stretching across different upstream versions. The two most notable examples of this were: First, the cluster comprising versions 5.4p1-1 to 5.5p1-6 (detailed view shown in Figure 3b). Second, the cluster comprising versions 6.6p1-1 to 6.6p1-3 as well as 6.9p1-2 and 6.9p1-3, which are more similar to each other in terms of memory pages than to the versions between.

The results indicate that almost no similarities exist between binaries of packages across different upstream versions. The results also indicate that memory savings by means of deduplicating binaries of *Apache* on Debian x86-64 can only be achieved if multiple instances of the same upstream version and ideally the same or a very close Debian revision are being executed on the host. For *sshd*, limited sharing potential exists between releases of some close upstream versions.

Figure 4 shows how many memory pages can be used in a signature for a binary of the *Apache-Debian-x86_64* dataset. The values are calculated on the assumption that signatures shall be used to identify not only the upstream version, but also the exact Debian patch level of the binary. The figure also shows how many pages of the binary are contained more than once within the binary or contain only zeroes or ones. It is also shown how many of the remaining pages are also contained in other versions of the binary. The remaining pages can be used as a signature.

The results show that the size of the signature is large for many of the versions surveyed as they contain many unique pages. These versions can be precisely identified using our attack. Results for the *sshd-Debian-x86_64* dataset were similar.

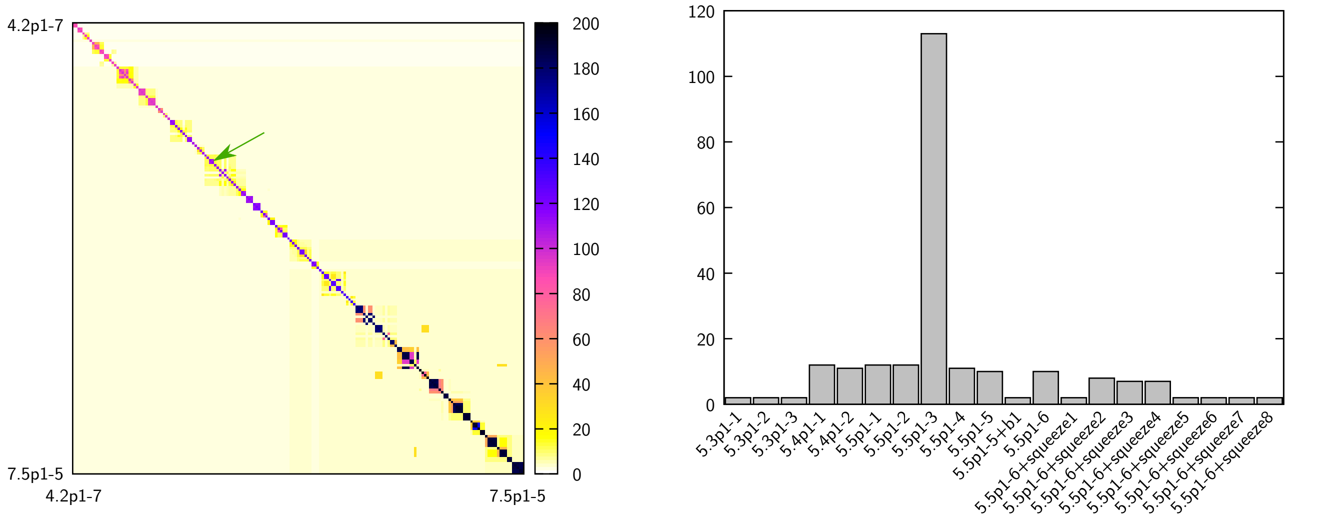
However, the size of the signature is small for many other versions. Due to the timing difference observed in a memory deduplication attack being far less pronounced for shorter signatures, it will be hard to identify these versions to the precision of a specific Debian revision. Signature size can be increased by grouping some of the affected versions with neighbouring versions and by creating a signature that describes the group. This reduces the precision of the version identification, but will make the memory deduplication attack easier to perform.



(a) Number of matching pages for all pairs of versions. The colour at the intersection of the line of version v_r with the column of version v_c indicates how many pages of v_r can also be found in v_c.

(b) Detail of selected cluster from (a) – Number of pages in Apache 2.4.10-10+deb8u5 (marked by the arrow in (a)) that can also be found in neighbouring versions.

Figure 2: Cross-version similarities – Apache-Debian-x86_64 dataset



(a) Number of matching pages for all pairs of versions. The colour at the intersection of the line of version v_r with the column of version v_c indicates how many pages of v_r can also be found in v_c.

(b) Detail of selected cluster from (a) – Number of pages in sshd 5.5p1-3 (marked by the arrow in (a)) that can also be found in neighbouring versions

Figure 3: Cross-version similarities – sshd-Debian-x86_64 dataset

4.5 Inter-distribution similarities

We now analyse whether signatures derived from the binaries of one Linux distribution can also be used to detect the version of the same software on another distribution. To this end, we compared binaries of the same software version from packages of several distributions in the same way as described in Sect. 4.4. Our experiments are based on the *sshd-crossdist* dataset.

Fig. 5 shows the number of duplicate pages between the different binaries. It can be seen that the binaries distributed by Debian are very similar to each other. Furthermore, while the Fedora releases 7.5p1-2 for Fedora 26 and 27 share 45 pages with each other, they are not similar to release 7.5p1-1 for Fedora 27. The Debian and Ubuntu releases share nine pages with each other. All other cross-distribution pairs of releases exhibit almost no similarities.

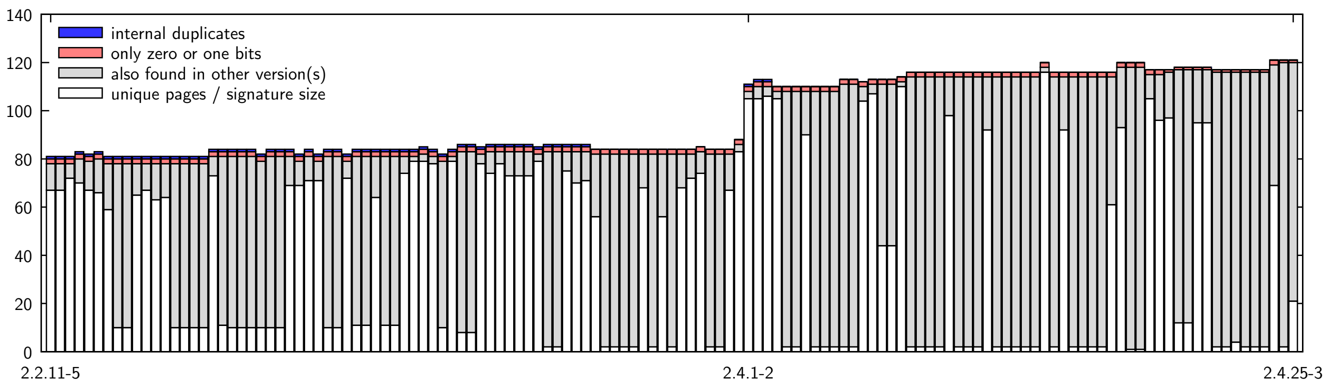


Figure 4: Number of unique pages that can be used as a signature for all versions of the Apache-Debian-x86_64 dataset (white) and reasons on the unsuitability of other pages.

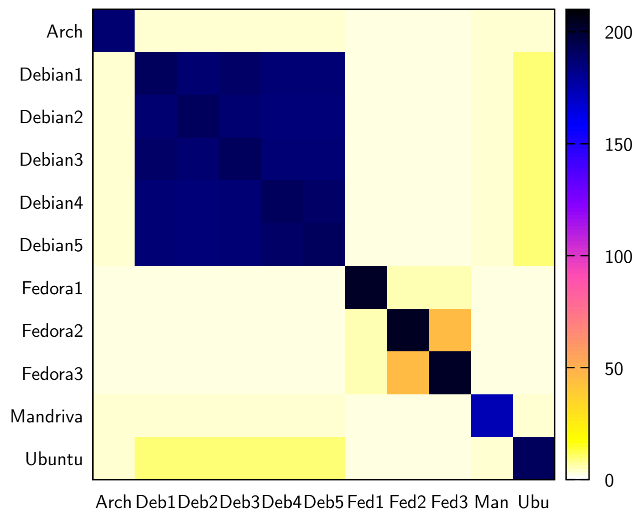


Figure 5: Cross-distribution similarities. Number of matching pages for all pairs of versions in the sshd-crossdist dataset. The colour at the intersection of the line of version v_r with the column of version v_c indicates how many pages of v_r can also be found in v_c.

4.6 Influence of Page size

In the following, we will present an analysis on the influence of changing the page size on the effectiveness of our attack and the memory saving potential of deduplicating executable code. To analyse whether decreasing the page size from the standard of 4 096 bytes increases the proportion of binaries that can be deduplicated, we analyse the number of matching pages across versions of the *Apache-Debian-x86-64* dataset for non-standard page sizes in the same way as described in Sect. 4.4.

The results of the experiment are shown in Figure 6. We divide all pairs of versions into two categories: high-sharing pairs (≥ 5% of pages shareable) and low-sharing pairs (< 5% of pages shareable). The results indicate that reducing the page size increases the percentage of shareable pages for pairs of versions that were already similar at standard page size. However, sharing opportunities remain almost unchanged for lower page sizes – the percentage even drops slightly, which can be explained by pages that are only partly filled. When page size is halved, pages that are less than half-filled will not be split into two pages, but only one (smaller) page will remain that now makes up a larger proportion of the binary.

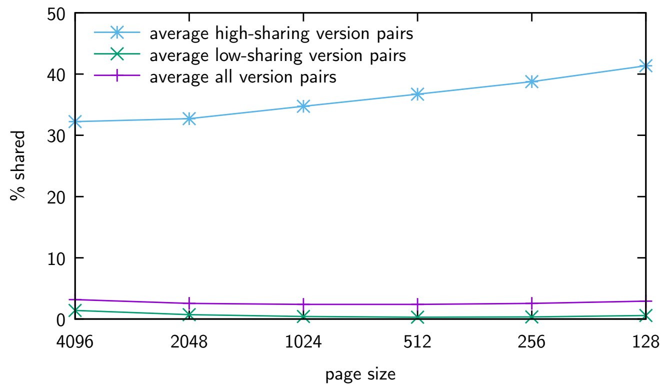


Figure 6: Influence of changing the page size on the proportion of shared pages. High-sharing version pairs are pairs of versions with ≥ 5% shared pages at standard page size

4.7 Attack Complexity

We will now analyse how long it takes to perform our attack. The duration for a successful run of our side-channel attack depends on the configuration of the deduplication mechanism and on the desired accuracy of the results.

How long it takes to perform a single measurement is defined by the time an attacker has to wait for deduplication to take place, which depends on how long the deduplication mechanism requires to scan the complete memory. In its standard configuration on Fedora 26 and RHEL 7.4, the ksmtuned daemon, which automatically configures KSM (cf. Sect. 2.1) according to memory usage, scans at least 1/65536 of the physical memory, i. e. it will take at most 655.36 seconds until all of a machine's memory has been scanned. For the remainder of this section, we will assume this as the time an attacker has to wait for deduplication to take place.

To probe a signature, multiple measurements should be performed to increase the accuracy of the results. The time this takes for one signature depends on the desired accuracy of the results. Figure 7 shows the impact of the number of measurements performed and the size of the signature on the accuracy of our version detection mechanism. To obtain the results, $1 \leq n \leq 20$ signature pages were loaded into the memory of m_a and m_v. As the concrete content of the pages is irrelevant for this experiment, pages were generated randomly. For the deduplication case, the pages were identical on the two VMs. For the non-deduplication case, the pages did not match. We performed 2 000 individual measurements for each case. We calculated the mean of the measurements for each test case to act as a baseline for classification (cf. Sect. 3.1). For different values of m, we then took 10 000 000 random samples of m measurements each from all our test cases and checked whether the mean of the sample was classified correctly. The accuracy value shown is aggregated over both the deduplicated and the non-deduplicated test case. As our classification rule is relatively simple and can probably be improved, the accuracy values can be considered a lower bound of what is possible.

It can be seen that measuring multiple pages at once increases the accuracy. Thus, measurements should be performed based on signatures that contain all pages unique among the different versions of that application. Also, accuracy increases with the number of measurements performed. However, even a single deduplicated page in a set of non-deduplicated page will increase the write time significantly, thus leading to false classifications. Therefore, signatures should be as large as possible, but not contain any pages that are also present in other versions. In the best case, every version has completely different pages, so that all of them can be used as a signature.

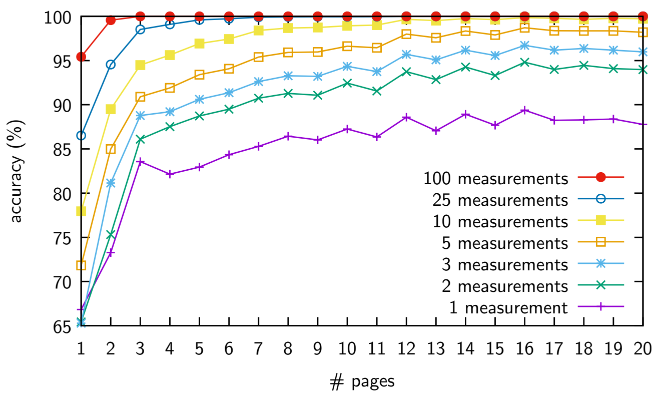


Figure 7: Influence of signature size and number of measurements on accuracy of the version detection mechanism

For signatures of four pages or more, 10 measurements have proved sufficient to achieve an accuracy of ≥ 95%. Smaller signatures require a higher number of measurements to achieve a similar accuracy: 12 measurements for three pages, 28 for two pages and 93 for one page.

For example, if ten measurements are desired, it takes about 1.8 hours to probe the signature pages. Increasing the number of measurements increases the time linearly. To probe all signatures of the *sshd-Debian-x86_64 dataset* consecutively takes about 10 days if 10 measurements are performed per signature. However, as our signature pages are disjunct in between different application versions, they can actually be probed in parallel if enough memory is available in the attack VM v_a. This reduces the time to about 1.8 hours, the same time it takes to probe a single signature. For that, all signatures are loaded into the memory of v_a at once. The attacker must then wait for deduplication to occur. Afterwards, the timing measurements can be performed consecutively. Each of them takes a fraction of a second. This process can then be repeated multiple times to achieve the desired number of measurements per signature.

5 COUNTERMEASURES

The easiest way of avoiding side-channel attacks by memory deduplication is to turn this feature off. However, this comes at the cost of eliminating all memory savings by deduplicating memory pages. Alternatively, this feature gets disabled only for pages belonging to executable binaries. According to our results, this will only require significantly more physical memory on systems hosting a large number of VMs that all run very similar software. However, modifications to the hypervisor (and possibly guest OS) would be necessary, so that they are aware whether a page actually belongs to a binary.

If a user who merely rents a VM on a host whose configuration they cannot control wants to prevent memory deduplication side-channel attacks on their VM, a possible solution would be to obfuscate the VM's memory. This could be achieved by deploying an Address Space Layout Randomization technique that – unlike the standard Linux implementation – does not only shuffle pages in memory, but randomises the memory contents on the sub-page level.

Another solution would be to slightly modify all executed binaries. Considering that the binaries released by different distributions are based on the same upstream version are highly different in their memory pages, it should be sufficient to compile the programs manually with some less commonly used compile options. Finally, the operator of a VM can also encrypt its memory. However, all of these techniques will make it impossible for the hypervisor to deduplicate memory pages, thus preventing any memory savings.

Instead of preventing the side-channel attack outright, it would also be possible to deceive attackers by placing pages of binaries that are not actually running on any VM or the host into memory, e. g. pages that are contained in our signatures. This can be done by either the operator of the host or anyone controlling a VM on the host. While an attacker would still be able to detect the presence of software versions that are being executed, this would come with a certain number of false positives. An effective defence by such an approach would require much more memory to load a signatures for many versions of many applications. It may, however, be suitable to prevent that an attacker gets to know the exact version of the host's hypervisor from memory deduplication attacks.

6 CONCLUSION

We have introduced a novel side-channel attack that is based on memory deduplication and that can detect software versions on co-resident VMs. We can even identify versions to the precision of a specific distribution patch level of an upstream release. This provides valuable knowledge to an attacker, who can perform attacks specifically targeting vulnerabilities in the software versions that were detected by the side-channel attack. No significant similarities were found between binaries from different distributions that were based on the same upstream release. This means that releases of the same upstream software version from different distributions can be easily distinguished. It also implies that the potential for memory savings by deduplicating executable code is limited for computers hosting VMs with homogeneous software configurations. Changes to the page size increase deduplication potential only for pairs of versions that already share a significant number of pages at standard page size, i. e. only for some pairs of releases of the same or neighbouring upstream versions by the same OS and distributions.

Our results indicate that we can detect the presence of a signature of four pages or more in another VM or on the host with a reasonable amount of 10 measurements with an accuracy of $\geq 95\%$. However, an actual attack takes time and for 10 measurements it will take 1.8 hours.

The side-channel can be prevented by disabling memory deduplication across multiple VMs – either completely or by modifying the deduplication mechanism to only not deduplicate executable code.

Possible future work includes studying a broader range of applications and extending the study to other operating systems, such as Windows. Furthermore, more advanced mitigation strategies should be developed to enable memory deduplication to take place without leaking information to other VMs.

REFERENCES

[1] Andrea Arcangeli, Izik Eidus, and Chris Wright. 2009. Increasing memory density by using KSM. In *Linux Symposium, Montreal, Canada*. 19–28.

[2] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. 2013. Message-Locked Encryption and Secure Deduplication. In *EUROCRYPT*. 296–312. https://doi.org/10.1007/978-3-642-38348-9_18

[3] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *IEEE Symposium on Security and Privacy*. 987–1004. https://doi.org/10.1109/SP.2016.63

[4] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. 2004. Safe hardware access with the Xen virtual machine monitor. In *Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS)*.

[5] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2010. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM* 53, 10 (2010), 85–93. https://doi.org/10.1145/1831407.1831429

[6] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. 2010. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security & Privacy* 8, 6 (2010), 40–47. https://doi.org/10.1109/MSP.2010.187

[7] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Know Thy Neighbor: Crypto Library Detection in Cloud. *PoPETs* 2015, 1 (2015), 25–40.

[8] Keren Jin and Ethan L. Miller. 2009. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference (SYSTOR)*. 7. https://doi.org/10.1145/1534530.1534540

[9] Jens Lindemann. 2015. Towards Abuse Detection and Prevention in IaaS Cloud Computing. In *International Conference on Availability, Reliability and Security (ARES)*. 211–217. https://doi.org/10.1109/ARES.2015.72

[10] Dutch T. Meyer and William J. Bolosky. 2012. A study of practical deduplication. *ACM Trans. Storage (TOS)* 7, 4 (2012), 14:1–14:20. https://doi.org/10.1145/2078861.2078864

[11] Grzegorz Milos, Derek Gordon Murray, Steven Hand, and Michael A. Fetterman. 2009. Satori: Enlightened Page Sharing. In *USENIX Annual Technical Conference*.

[12] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar R. Weippl. 2011. Dark Clouds on the Horizon: Using Cloud Storage as Attack Vector and Online Slack Space. In *USENIX Security*.

[13] Rodney Owens and Weichao Wang. 2011. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IEEE International Performance Computing and Communications Conference (IPCCC)*. 1–8. https://doi.org/10.1109/PCCC.2011.6108094

[14] Pasquale Puzio, Refik Molva, Melek Önen, and Sergio Loureiro. 2013. ClouDedup: Secure Deduplication with Encrypted Data for Cloud Storage. In *Cloud Computing Technology and Science (CloudCom)*. 363–370. https://doi.org/10.1109/CloudCom.2013.54

[15] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. 2011. Memory deduplication as a threat to the guest OS. In *European Workshop on System Security (EUROSEC)*. https://doi.org/10.1145/1972551.1972552

[16] Carl A. Waldspurger. 2002. Memory Resource Management in VMware ESX Server. In *Symposium on Operating System Design and Implementation (OSDI)*.

[17] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. 2013. Security implications of memory deduplication in a virtualized environment. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–12. https://doi.org/10.1109/DSN.2013.6575349