On the Detection of Applications in Co-Resident Virtual Machines via a Memory Deduplication Side-Channel

Jens Lindemann
Security and Privacy Group
Department of Computer Science
University of Hamburg, Germany
lindemann@informatik.uni-hamburg.de

Mathias Fischer
Security and Privacy Group
Department of Computer Science
University of Hamburg, Germany
mfischer@informatik.uni-hamburg.de

ABSTRACT

Nowadays, hosting services of multiple customers on the same hardware via virtualiation techniques is very common. Memory deduplication allows to save physical memory by merging identical memory pages of multiple Virtual Machines (VMs) running on the same host. However, this mechanism can leak information on memory pages to other. In this paper, we propose a timing-based side-channel to identify software versions running in co-resident VMs. The attack tests whether pages that are unique to a specific software version are present in co-resident VMs. We evaluate the attack in a setting without background load and in a more realistic setting with significant background load on the host memory. Our results indicate that, with few repetitions of our attack, we can precisely identify software versions within reasonable time frames and nearly independent of the background load. Finally, we discuss potential countermeasures against the presented side-channel attack.

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

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 [13]. 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 a danger especially in virtualised environments (e. g. [9, 11]), in which multiple virtual machines share the same hardware. One type of side-channel attacks in virtualised environments is based on the memory deduplication mechanism, which identifies and merges identical memory pages. This can save large amounts of physical memory [4, 27]. However, this mechanism can adversely affect the confidentiality of data in virtual machines. Suzaki et al. [24] 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 has already been published by us as a conference paper [14]. The attack 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. It requires fewer measurements to detect versions that have a large number of unique pages. Furthermore, memory activity has an impact on the number of measurements required: without load, fewer measurements are required to achieve a certain level of accuracy. For example, to achieve an accuracy of more than 99.9 % in the detection of the software version with background activity on the host, at least three measurements with a minimal signature size of five pages are necessary. Such an attack would take at least 32 minutes.

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.

Compared to our former work [14], we have improved our attack code by eliminating some of the noise observed in the measurements. As a result, we require fewer measurements to achieve the same level of accuracy in detecting application versions. We also evaluate the attack under more realistic conditions, i. e. with background load on the host. Furthermore, we more extensively discuss countermeasures against the identified side-channel attack.

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, before we discuss how executable files are loaded on Linux. 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.

Figure 1 shows an example of memory deduplication. Let us assume two VMs running on a host. Each VM is assigned four pges of virtual memory. Without deduplication (Figure 1a) every page in the virtual memory of the two VMs are mapped to exactly one distinct page in physical memory. When deduplication is activated (Figure 1b), it will scan the memory and find that there are two pages of identical content. These pages are then merged into one physical memory page, resulting in the two VMs sharing a physical memory page.

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

KVM.

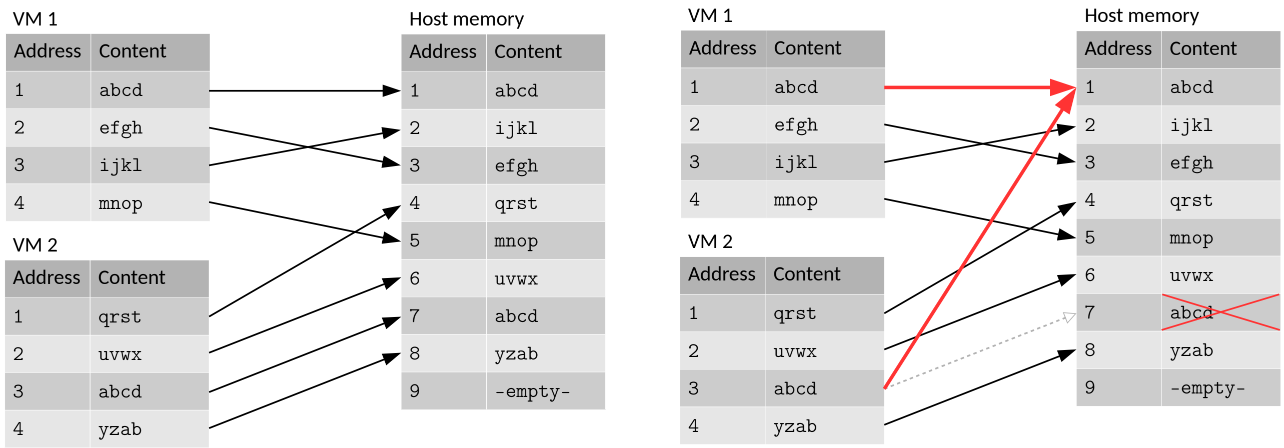
The KVM hypervisor is built into the Linux kernel and uses the "Kernel Samepage Merging" (KSM) technique [1] for memory deduplication. The guest OSs do not have to be modified for KSM. KSM runs in the background ans scans the memory pages of virtual machines for identical pages, which are then deduplicated. It will scan in specified intervals. When an interval has passed, a specific number of memory pages is scanned and, if appropriate, deduplicated.

Instead of using fixed values for interval and number of pages to scan per interval, the ksmtuned daemon can be used [22]. It will tune the KSM configuration according to the current memory usage on the host: The higher the memory usage is, the more pages will be scanned per interval. When memory usage decreases, the number of pages scanned per interval will also be decreased. Ksmtuned also supports turning off KSM when memory deduplication does not exceed a specified threshold.

Xen.

The Xen hypervisor provides a mechanism for sharing memory pages between VMs [5]. 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. [8]. Similar to KSM, it periodically scans the memory for shareable pages. Besides deduplicating identical pages, it also supports sharing similar pages. This is achieved by storing the difference between the deduplicated pages and patching the appropriate page when it is being accessed. Note that, unlike deduplicating identical pages, this also causes



(a) no deduplication **(b) deduplication active**

Figure 1: Memory deduplication

an overhead when pages are merely being read. In addition, it compresses unshareable pages that are not being used frequently. Compared with techniques that work on identical pages only, Difference Engine achieves higher memory savings. While the researchers originally published their code online, the project now seems dead and the repository is no longer available.

Another mechanism is Satori [18]. Unlike most other deduplication mechanisms, it requires modifications to the guest OS. Instead of periodically scanning the full VM memory for shareable pages, it checks whether a page can be deduplicated when it is being loaded. This means that even pages that only remain in memory for a short period of time can be deduplicated. However, pages that are changed to become identical to another page after they have initially been loaded into memory will not be detected. Satori allows a guest VM to specify pages that may not be shared. This implies disabling deduplication for specific memory areas and will eliminate both memory savings as well as memory timing side-channels in respect to these pages. We were unable to find a publicly available implementation of Satori.

VMWare ESXi.

VMWare ESXi uses its own deduplication mechanism, which has been described by Waldspurger [27]. 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 Loading of Executables in Linux

As described in Sect. 2.1, the content of two memory pages must be identical for them to be deduplicated. However, the position of the page in memory is irrelevant. Thus, we need to know the content of an executing program's memory pages, but not their position in memory.

Linux uses the Executable and Linkable Format (ELF) for executable files. The current version of the standard is 1.2 [25]. ELF is also used by many other modern Unix operating systems, such as FreeBSD and Solaris.

The data in an ELF file is organised in sections and segments. A file contains one or more segments. A segment contains one or more sections. For executing a program, only segments are relevant. An executable contains a program header table describing the segments contained in the file. For each segment, it contains information such as the type of the segment, its position and size in the file, the virtual memory address that it shall be loaded to and alignment requirements.

In the following, we will describe how ELF executables are placed into memory pages by the Linux kernel. The loading mechanism for ELF files can be found in the source file fs/binfmt_elf.c. To load an ELF object file, the function load_elf_binary is called. The function iterates over all entries in the program header table. It checks whether the corresponding segment is a loadable (PT_LOAD) segment. If it is, it calls the elf_map function.

The elf_map function then maps the specified segment into memory. It maps full memory pages, even if the virtual memory address specified in the program header table points to a position within a page. If this is the case, the bytes directly preceding the segment are loaded until the memory page is filled. A similar approach is taken if the segment does not end on a page boundary: The bytes directly succeeding the segment are loaded until the page is filled.

2.4 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 [12, 16]. However, its effectiveness is reduced for heterogeneous software configurations on the VMs [12]. 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 [9] or non-uploading of file contents [19]. Researchers have proposed Message-Locked Encryption as a countermeasure [2, 21].

Gruss et al. [6] demonstrate that it is possible to perform a *memory deduplication side-channel attack* from within a browser using JavaScript. Bosman et al. [3] apply this approach to the Microsoft Edge browser 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. [11] 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. Automatically generating signatures for this type of attack would be hard as the targeted function needs to be loaded into the cache by the victim, which would typically be triggered by its execution. Thus, signatures would need to take into account how likely a function is to be executed. If an automatic signature generation mechanism targeted a function that is unique for an application, but rarely executed (e. g. handling of an uncommon error), this would be of little use for detecting an application. Furthermore, as their attack targets a single function in the library, it will be unable to distinguish versions in which the analysed function is identical. This implies that different functions may have to manually be found to distinguish different pairs of versions.

Gulmezoglu et al. [7] describe a cache-based side-channel attack to detect applications in co-resident virtual machines. They use machine learning to train a classifier on the cache usage patterns of applications. While their attack has the advantage of not requiring memory deduplication to be active, it is unclear whether it can be used to exactly identify the executed version of an application.

Xiao et al. [28] 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. [24] 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 [20] 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 [20], 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 [24] 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.

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

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

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

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 takes a relatively long time. 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. To avoid measurements influencing each other, the overwriting operations should not overlap. It is not a problem to perform an overwriting measurement on one of the signatures while other signatures are in the waiting phase, though.

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:

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

Our approach for deriving signatures is similar to the one for detecting operating systems by Owens and Wang [20]. 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 before discussing the limitations of our approach.

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

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

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. This tool has been improved in comparison with the version used in our former work [14]. The old version was susceptible to noise from disk I/O, which has been removed by pre-caching the data that is to be loaded into memory when overwriting. The second one loads signature pages into memory aligned to page boundaries to enable experiments that do not use a running executable.

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 7.9p1* for different distributions. Our datasets consist of the following application versions:

- The **Apache-Debian-x86_64** dataset consists of all 160 Debian releases of *Apache* available for the x86-64 platform and includes versions from 2.2.11-5 to 2.4.37-1.
- The **sshd-Debian-x86_64** dataset consists of all 211 Debian releases of *sshd* available for the x86-64 platform and includes versions from 4.2p1-7 to 7.9p1-4.
- The **sshd-crossdist** dataset consists of 10 package versions of *sshd 7.9p1* from Arch Linux, Debian, Fedora, Mageia and Ubuntu. Multiple revisions are included for Debian (3) and Fedora (4).

Our datasets contain only the main executable of each application (*httpd* for Apache, *sshd* for the SSH daemon). 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 7.9p1 of sshd, which was available for download for a variety of distributions at the time of dataset creation.

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

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

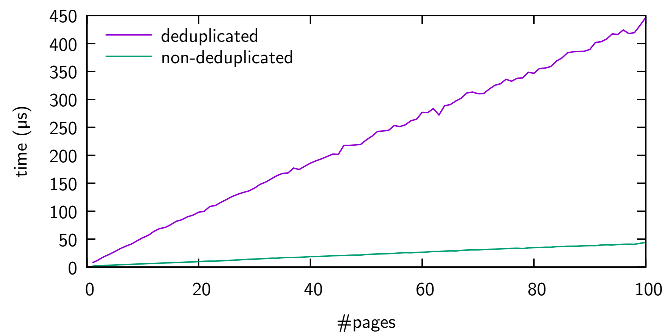


Figure 2: Average write times (in microseconds) to n deduplicated/non-deduplicated pages

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 2 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 1 000 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.

Write times to deduplicated pages are higher 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.

Figure 3a shows a histogram of 1 000 write times each for a single deduplicated or non-deduplicated page without background load on the system. As expected, the write times to non-deduplicated pages are typically lower than those to deduplicated pages. However, when overwriting a single page, some of the slower measurements for non-deduplicated pages fall into the same range as some of the faster measurements for deduplicated pages. This implies that performing a single measurement only will not be sufficient to reliably distinguish the two cases and thus determine whether another copy of the page is present on the host. Figure 3c shows the write times with background load, generated by running six instances of memtester 4.3.0 on the host while performing the measurements. While write times to non-deduplicated pages are still lower on average than those to deduplicated pages, measurements for both are spread out over a significantly larger range of times and overlap more. This makes the two cases harder to tell apart.

Figure 3b shows a histogram of 1 000 write times each for 100 deduplicated or non-deduplicated pages without background load on the system. As with the single pages, overwriting 100 deduplicated pages takes longer than overwriting 100 non-deduplicated pages. However, the measurements do not overlap. This implies that we could reliably distinguish whether a 100-page signature is present on the host based on a single measurement only in our test setup. Figure 3d shows shows the write times for 100 pages with background load. As for single pages, these are spread out over a wider range of times. However, they do not overlap and could still be distinguished reliably based on a single measurement.

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 4a shows the number of matching pages between all versions for the *Apache-Debian-x86_64* dataset. Figure 5a 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 revisions corresponding to this upstream version are completely different and share only a single page with the non-backport versions. Further examples of similar binary versions are 2.2.22-6 to 2.2.22-13+deb7u2 and 2.4.10-1 to 2.4.10-9. While all versions in these clusters share more than 20 pages with each other, both clusters contain inner clusters of versions even more similar to each other. For example, while the versions in the former cluster all share at least 28 pages with each other, versions 2.2.22-7 to 2.2.22-10 share almost all their pages with each other. Figure 4b gives a detailed view of this cluster and shows how many of the pages in version 2.2.22-9 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 – contains a few clusters stretching across different upstream versions. While most of these span versions corresponding to directly adjacent upstream releases (e. g. 5.4p1 and 5.5p1), there are also clusters of slightly more distant versions. One notable example of this is the cluster comprising versions 7.2p2-6 to 7.2p2-8 as well as 7.4p1-1 to 7.4p1-5, which are more similar to each other in terms of memory pages than to the versions between. A detailed view of this cluster is shown in Figure 5b.

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.

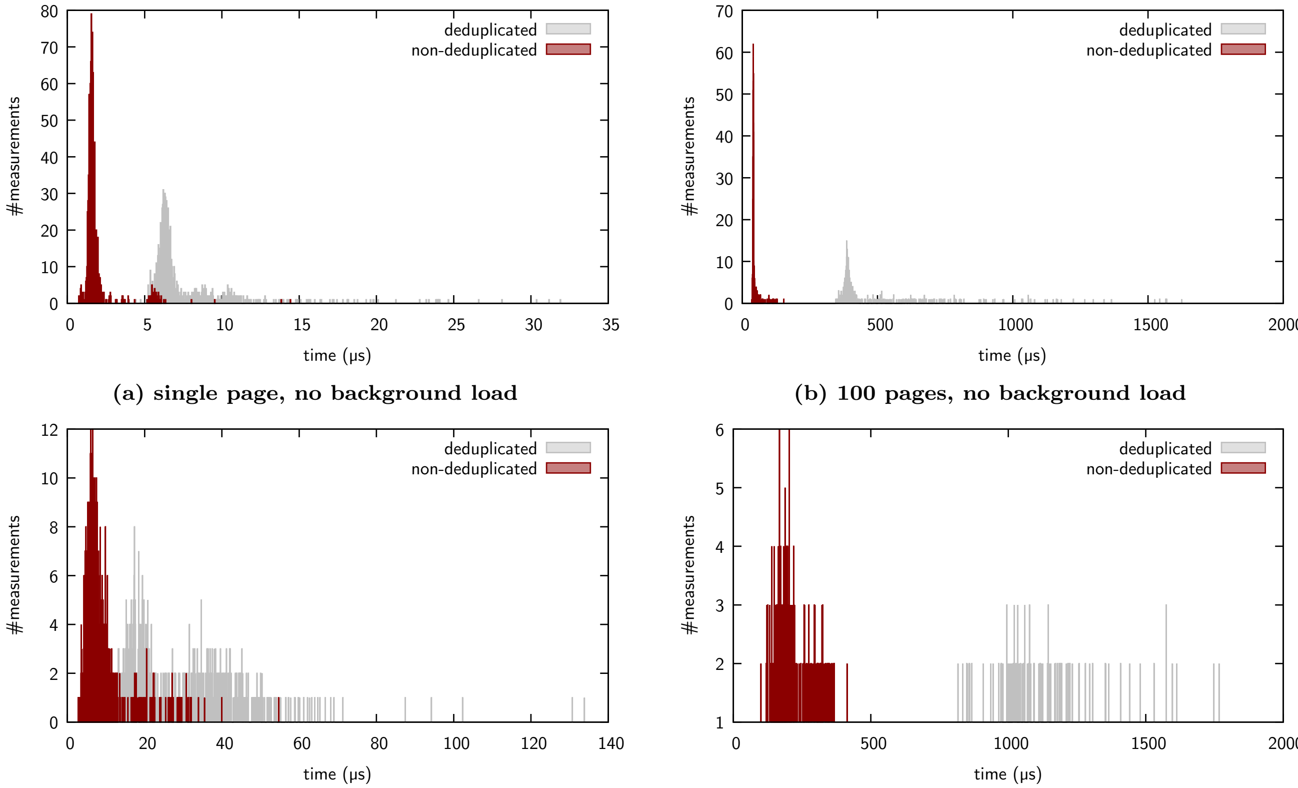
Figure 6a shows how many memory pages can be used in a signature for a binary of the *Apache-Debian-x86_64* dataset. The values are calculated on the assumption that signatures shall be used to identify not only the upstream version, but also the exact Debian patch level of the binary. The figure also 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. Figure 6b shows the results for the sshd-Debian-x86_64 dataset.

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.

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.

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



(a) single page, no background load

(b) 100 pages, no background load

(c) single page, with background load

(d) 100 pages, with background load

Figure 3: Histogram of write times (in microseconds) to deduplicated and non-deduplicated pages

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.

Figure 7 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, the Fedora releases are relatively similar to each other. We found release 7.9p1-1 for Fedora 29 to be more similar to 7.9p1-1 for Fedora 30 than to the 7.9p1-2 releases for both Fedora 29 and 30. The Debian and Ubuntu releases share five to seven pages with each other. All other cross-distribution pairs of releases exhibit no similarities.

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.

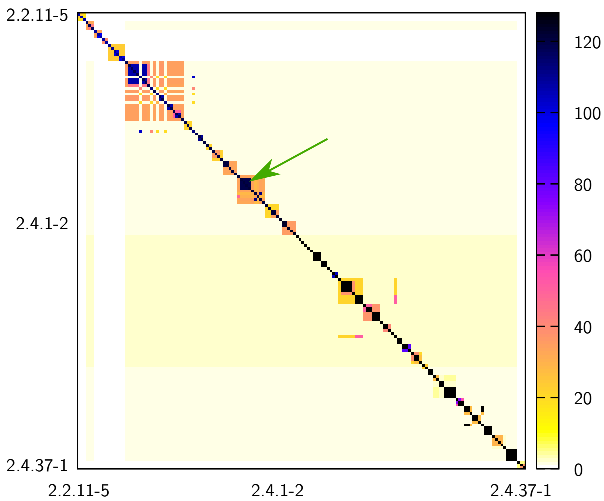
The results of the experiment are shown in Figure 8. We divide all pairs of versions into two categories: high-sharing pairs ($\geq 5\%$ of pages shareable) and low-sharing pairs ($< 5\%$ of pages shareable). The results indicate that reducing the page size increases the percentage of shareable pages for pairs of versions that were already similar at standard page size. However, sharing opportunities remain almost unchanged for lower page sizes.

4.7 Attack Complexity

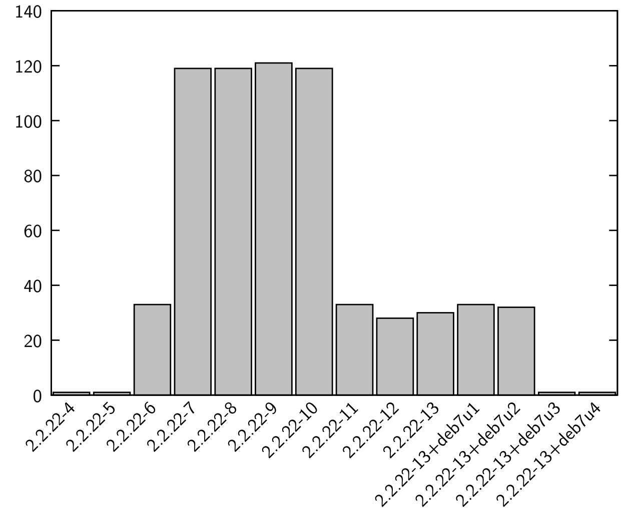
We will now analyse how long it takes to perform our attack. The duration for a successful run of our side-channel attack depends on the configuration of the deduplication mechanism and on the desired accuracy of the results.

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.

In the following, we want to establish which accuracy can be achieved depending on the signature size and the number of measurements performed (i. e. the time needed for the attack). As in Sect. 4.3, our test setup includes two VMs m_a and m_v. We first created a training dataset that



(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 2.2.22-9 (marked by the arrow in (a)) that can also be found in neighbouring versions.

Figure 4: Cross-version similarities – Apache-Debian-x86_64 dataset

was used to determine the classification threshold. As in a real attack (cf. step 1 in the attack procedure in Sect. 3), we loaded n pages into the memory of m_a. As the concrete content of the pages is irrelevant for this experiment, pages were generated randomly. Then, we loaded the same n pages into memory again as well as n pages filled with different data. After waiting for the deduplication to occur, we measure the time it takes to overwrite each set of pages. This process is performed 1 000 times, so that we have 1 000 training measurements for both the deduplicated and the non-deduplicated case. Note that this number of training measurements is not unrealistic in an actual attack, as measurements can be taken in parallel if different sets of data are used. The approach for creating our test dataset is identical except that the pages that are to be deduplicated are loaded into the memory of m_v (and later into m_a's memory only once).

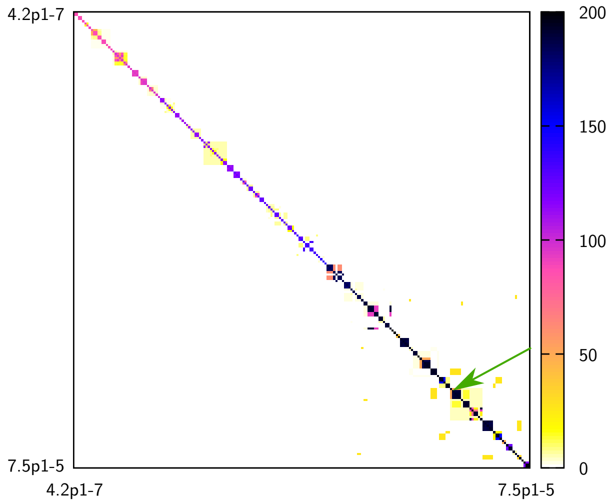
We perform our experiments in two scenarios: In the first scenario, only m_a and m_v are active on the host system. The VMs run only the OS and our analysis tools. The host was only running the base system and the two VMs. No further VMs were active. This ensures that there is relatively little load on the memory of the host that is not attributed to the measurements themselves.

In the second scenario, we simulate background memory activity on the host. As in the first scenario, the attack (m_a) and victim (m_v) VMs were active and no further VMs were active. However, in addition to the base system and the two VMs, the host was concurrently executing six instances of memtester 4.3.0. Each instance was configured to use 1 GiB of memory and run in an infinite loop. This ensures that there were constantly read and write accesses being made to the physical memory of the host.

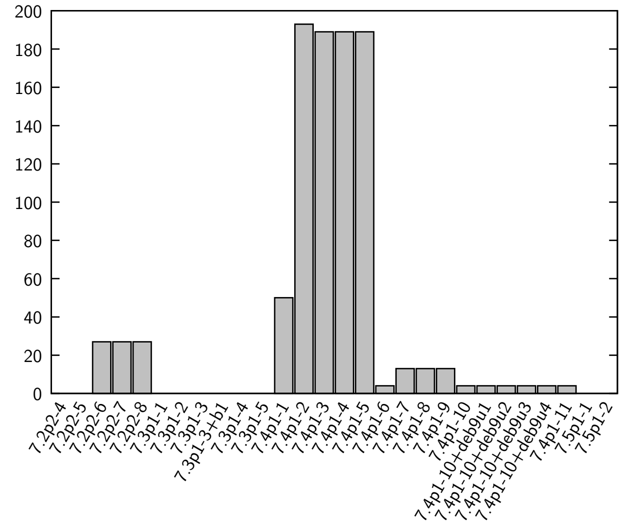
To probe a signature, multiple measurements should be performed to increase the accuracy of the results. The time this takes for one signature depends on the desired accuracy of the results. Figure 9a shows the impact of the number of measurements performed and the size of the signature on the accuracy of our version detection mechanism when there is no background load on the system. We calculated the mean of the training 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 this classification rule is relatively simple, 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 and can lead 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.

Without background load on the host, relatively few measurements are required to achieve a high accuracy even for small signatures: For signatures of a single page, six measurements were needed for an accuracy of $\geq$ 99.9%. For larger signatures, fewer measurements were needed to achieve a similar level of accuracy, e. g. three measurements for sig-



(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 7.4p1-2 (marked by the arrow in (a)) that can also be found in neighbouring versions

Figure 5: Cross-version similarities – sshd-Debian-x86_64 dataset

natures of two pages and two measurements for signatures of five pages or more.

Under load, the number of measurements required to achieve a certain level of accuracy increases, as shown in Figure 9b. In our experiments, nine measurements were required to achieve an accuracy of $\geq 99.9\%$ for single-page signatures. For two-page signatures, we were able to achieve this level of accuracy with six measurements, while three measurements were sufficient for signatures of five pages.

For example, if six measurements are desired, it takes about 66 minutes 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 eight days if six 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 66 minutes, 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.

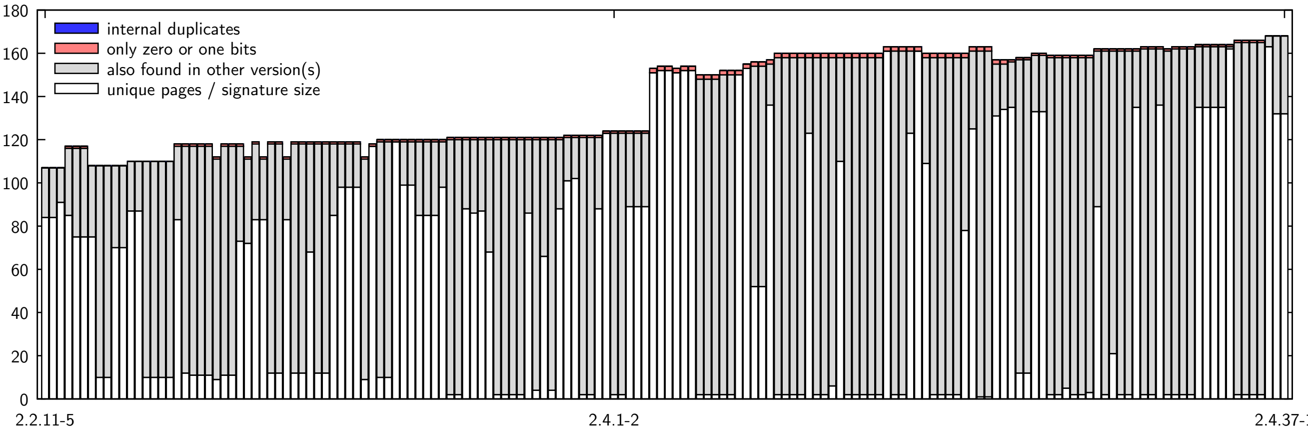
To further reduce the number of measurements required, similar versions can be grouped [15]. This results in larger signatures and eliminates all small signatures for our datasets. While this means that an attacker can no longer identify the exact version and distribution patch level of an application, we found that for our datasets, almost all groups contain only different distribution patch level releases belonging to the same upstream version released by the original developers of the software. The only exception from this was a group in the sshd-Debian-x86_64 dataset, which contained three versions from two neighbouring upstream versions. Thus, the versions in a group are likely to contain similar security vulnerabilities, which means that for most attackers, the version can still be identified precisely enough if groups are formed.

4.8 Limitations

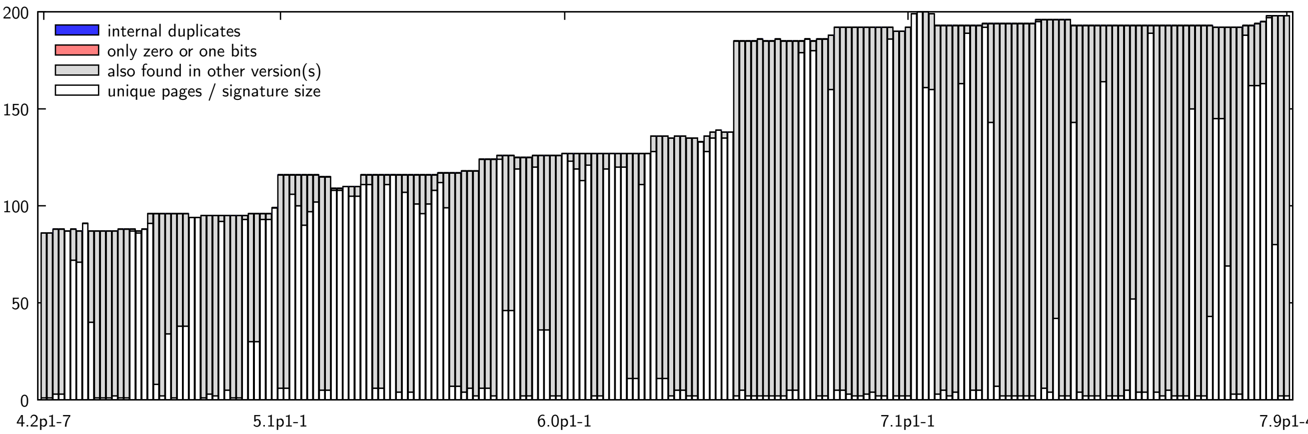
In the following, we discuss some limitations to the presented side-channel attack.

Attacker does not know IP of co-resident VMs.

The attack presented in this paper allows an attacker to find out whether a specific version of an application is running in another VM that is co-resident on the host. However, it does not allow an attacker to find out in which specific VM the application is being executed. It also does not provide any information on how to contact the VM that runs the identified application, which is necessary to exploit a potential security vulnerability. If an attacker is interested in attacking a specific online service, they may try to obtain a VM that is co-resident with a VM hosting it. Varadarajan et al. [26] have shown that this is realistic in public cloud environments. Depending on a cloud service provider's infrastructure, IP addresses may also be correlated with the placement and type of VMs [23], which would allow an attacker to increase its chances of obtaining a co-resident VM by choosing the deployment zone and instance type accordingly. This may also help an attacker that does not have a specific target in mind in narrowing down potential IP addresses of vulnerable co-resident VMs. For that, the attacker can randomly spawn attack VMs to find vulnerable VMs.



(a) Apache-Debian-x86_64 dataset



(b) sshd-Debian-x86_64 dataset

Figure 6: Number of unique pages that can be used as a signature (white) and reasons for the unsuitability of other pages

Memory deduplication must be active.

Our attack assumes that deduplication is activated on the host. Nowadays, many of the larger public cloud service providers such as Google [10] have turned off memory deduplication in fear of side-channel attacks. However, the technique can offer large memory savings [4, 27], which makes it attractive to server operators. This is especially true in environments where users of VMs are believed to be at least somewhat trustworthy, e. g. in private clouds.

Application versions might be indistinguishable.

Another assumption is that application versions are sufficiently different from each other. For the datasets we surveyed, this is the case. When generating signatures for individual application versions in our dataset, all signatures contain at least one page. This implies that all versions can be differentiated. Theoretically, however, it is possible for the signature generation to fail. This could be caused by two identical binaries in two different versions of a package, e. g. if only a default configuration file was changed between the package versions. It can also be caused by a version containing only pages that are also present in multiple different other versions, e. g. $v_1 = \{a, b, c\}, v_2 = \{a, d, e\}, v_3 = \{b, c, f\}$ will lead to the signature generation for version v_1 failing. Such situations can be resolved by creating group signatures [15] for the affected versions.

Our experiments were conducted on Linux only.

Experiments were only conducted for the Linux OS, which is dominant in cloud environments. However, we believe that our results are applicable to other operating systems as well. Most Unix-based OSs use the ELF file format for their executables and will employ a very similar loading mechanism to Linux. Adapting the attack to another OS requires taking into account how executables are structured and loaded on that system. For example, Windows uses the Portable Executable (PE) format [17] for its executables. For PE files, sections are loaded instead of segments. Sections are described in a similar header table as in ELF files, which can be used as a base for analysing a PE executable.

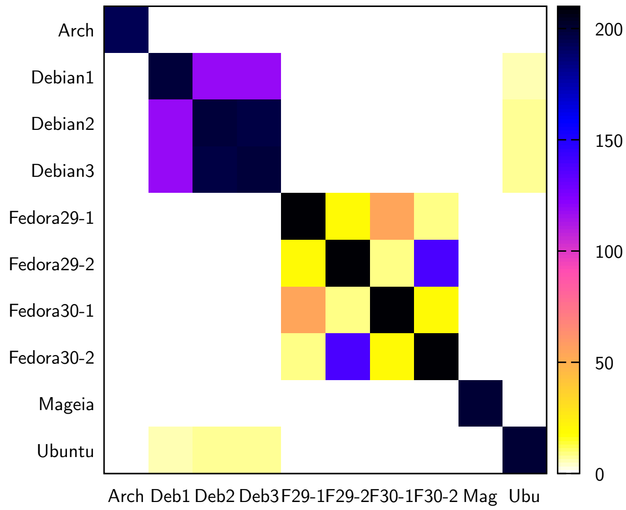


Figure 7: Cross-distribution similarities. Number of matching pages for all pairs of versions in the sshd-crossdist dataset. The colour at the intersection of the line of version v_r with the column of v_c indicates how many pages of v_r can also be found in v_c.

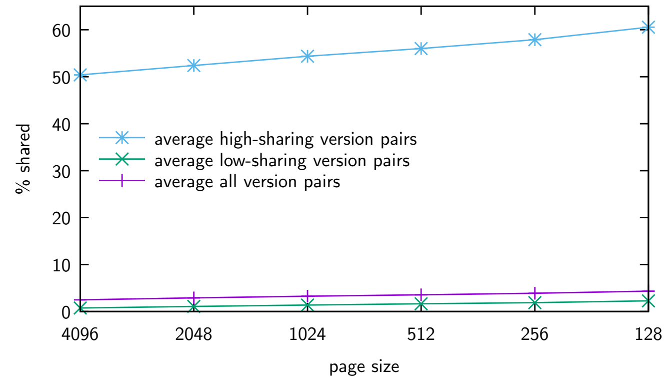


Figure 8: Influence of changing the page size on the proportion of shared pages. High-sharing version pairs are pairs of versions with $\geq 5\%$ shared pages at standard page size.

5. COUNTERMEASURES

In this section we discuss potential countermeasures against the presented attack. On the one hand, some of these aim at removing the side-channel altogether, but will also remove the memory savings offered by deduplication. On the other hand, some countermeasures aim at reducing the effectiveness of attacks without fully eliminating the side-channel.

Deactivating Memory Deduplication.

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 guest OS would be necessary to make them aware whether a page actually belongs to a binary.

Slow down writes to non-deduplicated pages.

Another approach that the operator of the host can take would be to slow down writes to non-deduplicated memory pages. If write operations are slowed down to the level of deduplicated pages, the side-channel is eliminated. However, this requires significant modifications to be made to the host operating system as write operations to non-deduplicated pages will normally not pass through the deduplication mechanism. This should not affect the performance of read-heavy workload, but it is unclear how large the adverse effect on performance for more write-heavy workloads would be.

Obfuscate Memory.

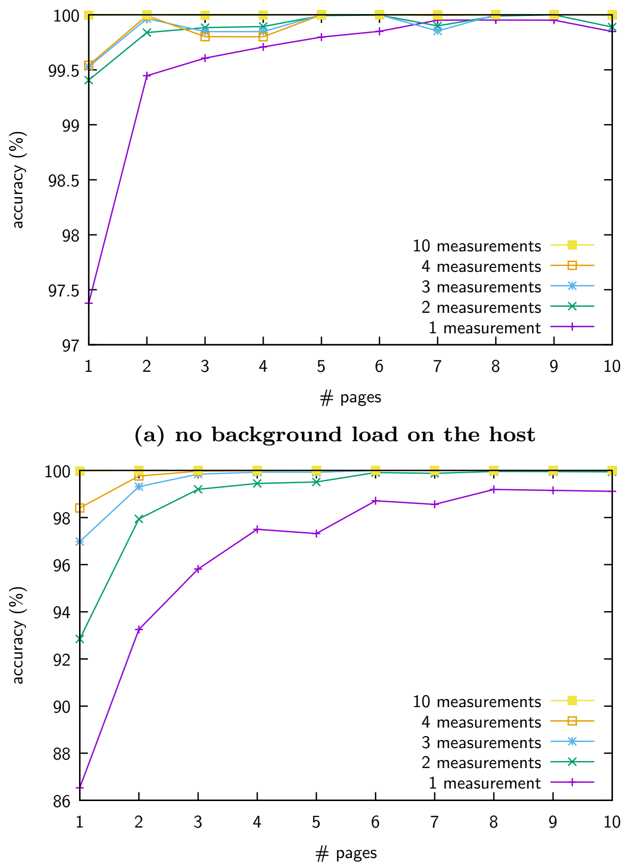
If a user who merely rents a VM on a host whose configuration they cannot control wants to prevent memory deduplication side-channel attacks on their VM, a possible solution would be to obfuscate the VM's memory. This could be achieved by deploying an Address Space Layout Randomization technique that – unlike the standard Linux implementation – does not only shuffle pages in memory, but randomises the memory contents on the sub-page level. This would ensure that all bits of the start address of a segment of an ELF segment are random. Therefore, the alignment of the segment's contents to page boundaries would be randomised, resulting in 4 096 possible alignments. As two pages will only be deduplicated if they match entirely, a different alignment prevents deduplication. An attacker could thus not simply use signatures as described in this paper. The attack would need to take all possible alignments of an application's pages into account, i. e. attackers would need to probe 4 096 times as many signatures. While these signatures can still be checked in parallel, this requires a lot more memory. If not enough memory is available, some signatures will have to be checked sequentially, increasing the time for the attack.

Modify Binaries.

Another solution would be to slightly modify all executed binaries. This can be achieved without recompiling by inserting randomly-placed NOP opcodes into an application's binary. Alternatively, it should also be sufficient to compile the programs manually with some less commonly used compile options considering that the binaries released by different distributions are based on the same upstream version are highly different in their memory pages.

Encryption.

The user of a VM can also encrypt its memory. However, all of these techniques will make it very hard or impossible for the hypervisor to deduplicate memory pages, thus preventing memory savings.



(a) no background load on the host

(b) with background load on the host

Figure 9: Influence of signature size and number of measurements on accuracy of the version detection

Decoy Signatures.

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 a specific sensitive application from memory deduplication attacks. It is not a replacement for regularly updating the system, though: In case of a lack of updates of both the application and the decoy signatures, an attacker would still be able to establish an upper bound on the application version.

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 targeting specific 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 five pages or more in another VM or on the host with a reasonable amount of three measurements with an accuracy of $\geq 99.9\%$ even if there is significant load on the memory of the host. However, an actual attack takes time and for three measurements it will take about 32 minutes.

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.

7. REFERENCES

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

[2] M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In *EUROCRYPT*, pages 296–312, 2013.

[3] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy*, pages 987–1004, 2016.

[4] C. Chang, J. Wu, and P. Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2011*, pages 244–249, 2011.

[5] K. Fraser, S. H, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the xen virtual machine monitor. In *Operating System and Architectural Support for the On-demand IT Infrastructure (OASIS)*, 2004.

[6] D. Gruss, D. Bidner, and S. Mangard. Practical memory deduplication attacks in sandboxed javascript. In *ESORICS, Part I*, pages 108–122, 2015.

[7] B. Gulmezoglu, T. Eisenbarth, and B. Sunar. Cache-based application detection in the cloud using machine learning. In *ACM Asia Conference on Computer and Communications Security, AsiaCCS 2017*, pages 288–300, 2017.

[8] 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. *Commun. ACM*, 53(10):85–93, 2010.

[9] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[10] A. Honig and N. Porter. 7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext. https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext?m=1, January 2017. Accessed: November 28, 2018.

[11] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Know thy neighbor: Crypto library detection in cloud. *PoPETs*, 2015(1):25–40, 2015.

[12] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *The Israeli Experimental Systems Conference (SYSTOR)*, page 7, 2009.

[13] J. Lindemann. Towards abuse detection and prevention in IaaS cloud computing. In *International Conference on Availability, Reliability and Security (ARES)*, pages 211–217, 2015.

[14] J. Lindemann and M. Fischer. A Memory-Deduplication Side-Channel Attack to Detect Applications in Co-Resident Virtual Machines. In *ACM Symposium on Applied Computing (SAC) 2018*, pages 183–192, 2018.

[15] J. Lindemann and M. Fischer. Efficient identification of applications in co-resident vms via a memory side-channel. In *ICT Systems Security and Privacy Protection - IFIP TC 11 International Conference (SEC) 2018*, pages 245–259, 2018.

[16] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Trans. Storage (TOS)*, 7(4):14:1–14:20, 2012.

[17] Microsoft Windows Dev Center. PE Format. https://docs.microsoft.com/en-gb/windows/desktop/Debug/pe-format, November 2018. Accessed: November 30, 2018.

[18] G. Milos, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *USENIX Annual Technical Conference*, 2009.

[19] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. R. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, 2011.

[20] R. Owens and W. Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *IEEE International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2011.

[21] P. Puzio, R. Molva, M. Önen, and S. Loureiro. ClouDedup: Secure deduplication with encrypted data for cloud storage. In *Cloud Computing Technology and Science (CloudCom)*, pages 363–370, 2013.

[22] Red Hat. Kernel Same-Page Merging (KSM). In: Red Hat Enterprise Linux 7 Virtualization and Optimization Guide.

[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *ACM Conference on Computer and Communications Security (CCS)*, pages 199–212, 2009.

[24] K. Suzaki, K. Iijima, T. Yagi, and C. Artho. Memory deduplication as a threat to the guest OS. In *European Workshop on System Security (EUROSEC)*, 2011.

[25] Tool Interface Standards (TIS) Committee. Executable and Linking Format (ELF) Specification Version 1.2. 1995.

[26] V. Varadarajan, Y. Zhang, T. Ristenpart, and M. M. Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security, 2015*, pages 913–928, 2015.

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

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