Efficient Identification of Applications in Co-resident VMs via a Memory Side-Channel*

Jens Lindemann(✉) and Mathias Fischer

Department of Computer Science
University of Hamburg, Germany
{lindemann,mfischer}@informatik.uni-hamburg.de

Abstract. Memory deduplication opens a side-channel that enables attackers to detect if there is a second copy of a memory page on a host their Virtual Machine (VM) is running on, and thus to gain information about co-resident VMs. In former work, we presented a practical side-channel attack that can even detect which specific versions of applications are being executed in co-resident VMs. In this paper, we enhance this attack by testing for representative groups of pages for certain groups of application versions, so-called page signatures, instead of testing for a single application version only. As a result, our new attack is significantly more efficient. Our results indicate that the attack duration can be reduced from several hours to minutes at the cost of a small loss in precision only.

1 Introduction

Today, more and more services on the Internet make use of VMs rented from cloud providers. These individual VMs as well as the infrastructure of such providers are under constant attacks. To prepare the attacks, attackers actively scan their targets for vulnerabilities. For example, obtaining information on the versions of the applications running in VMs allows them to launch targeted attacks that exploit specific vulnerabilities of these applications.

Cloud providers do not allow vulnerability scanning to be performed on or from within their infrastructure [8]. Consequently, attackers that use easily detectable network scans are usually blacklisted or even banned permanently. However, attackers can still try to exploit the virtualised environments within clouds by using side-channels (e. g. [6, 7, 13]), which are far harder to detect.

In former work [9], we have presented a side-channel attack to detect the version of applications in co-resident virtual machines based on memory deduplication. For this attack, we generate signatures for each individual version, i. e. we look for memory pages that are unique to this specific version and that are not shared by any other version. We then test whether another copy of these pages is present on the system from within a VM. For that, we exploit a timing side-channel that exists because of the memory deduplication mechanism as part of many virtualization techniques. This attack works better the larger a signature is. We found that many signatures are very small (often only containing a single page) when trying to identify a specific application version.

* This paper is published in: Janczewski L., Kutyłowski M. (eds) ICT Systems Security and Privacy Protection. SEC 2018. IFIP Advances in Information and Communication Technology, vol 529, pp. 245259. The final publication is available at Springer via https://doi.org/10.1007/978-3-319-99828-2_18.

The main contribution of this paper is a modified side-channel attack along with an improved classifier that can detect application versions way faster than our original attack. We base our detection on groups of similar rather than on individual versions of applications to speed up the attack. We check for sets of pages, so-called page signatures, that are shared among a set of application versions. These page signatures are jointly loaded and subsequently overwritten in the attacker VM, while measuring the necessary time for the latter operation. In case the write takes longer than expected, duplicates of the pages got deduplicated to save memory. Writing to such a page causes it to be copied and duplicated first, thus increasing the latency when overwriting the page. However, several repetitions are required to be sure about a duplicate page in the victim VM. When testing for larger page signatures, i. e. several pages at the same time, fewer repetitions (and thus less time) are required to be sure that these pages also exist in another VM. Our evaluation indicates that using overlapping pages of groups of different application versions as signatures can significantly decrease the time for carrying out the attack compared to identifying the exact version (22 minutes instead of 17 hours). The loss of precision is minimal – attackers will still be able to narrow down the version sufficiently to perform a targeted attack on known vulnerabilities. In all but one case in our results, an attacker would be able to exactly identify the upstream version of an application, as the groups contain only different distribution patch levels of a version.

The rest of the paper is structured as follows: In Sect. 2 we present background information and related work. Sect. 3 describes how we detect groups of versions using our side-channel attack. Sect. 4 presents evaluation results on the efficiency and effectiveness of the attack as well as on the optimised classifier. Sect. 5 discusses countermeasures and Sect. 6 concludes the paper.

2 Background and Related Work

In this section we first explain the concept of memory deduplication and present our attacker model. Then, we briefly describe our approach for detecting individual application versions via a memory deduplication side-channel attack. Finally, we discuss related work.

2.1 Memory Deduplication

Memory deduplication is a technique that allows to save physical memory by removing redundant information. The memory of a computer is organised into memory pages, which are typically 4 096 bytes large. Memory deduplication looks for identical pages in memory. It then removes all but one copy of the page. All other occurrences are replaced by a reference to the remaining copy.

When a deduplicated page is to be modified, a new copy of the page has to be created first. This is referred to as “Copy-on-Write” (CoW). If a new copy was not created first, all instances of the page would be modified. Copying a page takes time, which results in writes to deduplicated pages taking longer

than writes to non-deduplicated pages. This timing difference can be measured to infer whether another copy of a page is present on a host.

The Linux kernel includes a memory deduplication mechanism called Kernel Samepage Merging (KSM) [1], which is used by the KVM hypervisor. KSM regularly scans the memory for pages that can be deduplicated. VMWare ESXi includes a similar memory deduplication mechanism [14]. While the Xen hypervisor also includes a memory deduplication mechanism [3], it relies on additional software to identify shareable pages, such as Difference Engine [5] or Satori [10].

2.2 Attacker Model

A host h runs a set of virtual machines M. All versions of an application are contained in the set A. We denote an individual version as $a_v \in A$. Our attacker is in control of a VM $m_a \in M$ and can only observe the network traffic of this specific VM. They cannot observe the traffic of another VM $m \in M \setminus m_a$ or the host h. In this paper, we assume that the attacker wants to know what version $a_v \in A$ of an application is being executed in another VM $m \in M \setminus m_a$ on h.

Alternatively, an attacker who does not have full control of m_a could also target applications that are running within their own VM m_a, but outside their scope of control, e. g. those executed by another user. If the deduplication mechanism is configured to also deduplicate pages of the host memory itself, an attacker could also use the side-channel attack to determine the version of an application running on the host (e. g. the hypervisor).

2.3 Memory-Deduplication-Based Detection of Applications

Memory deduplication opens up a timing side-channel that can reveal whether a page has been deduplicated or not. This can be used to detect applications [13] or data in general [2] in other VMs, but will not reveal in which particular VM a page is present. In former work [9], we presented an approach for detecting individual versions of applications in other VMs by means of a memory deduplication side-channel. In this subsection, we will briefly outline the attack procedure.

To detect the presence of a specific application version in another VM, one first needs signatures. These signatures should contain only pages that are unique to a version. We define $pages(a_v)$ to return all pages of an application version's load segments excluding duplicates within the binary and pages containing only zeroes or ones. These types of pages would be deduplicated even without the application being executed on the host. A signature for an individual version can then be generated:

$$sig(a_v) = pages(a_v) \setminus \bigcup_{a \in \{A \setminus a_v\}} pages(a) \tag{1}$$

To probe for the presence of an application version, an attacker first needs baseline measurements for both the deduplicated and the non-deduplicated case to compare their actual probing measurements to. The baseline measurements

are performed by taking a number of pages equal to the length of the signature and measuring the time it takes to overwrite these pages in one go. For the deduplicated case, two identical copies of the pages are written to memory and one is then overwritten after deduplication has occurred. Between writing and overwriting the pages, the attacker has to wait for a certain amount of time as the deduplication mechanism of the host only scans and deduplicates memory pages periodically. For the non-deduplicated case, pages containing random data are written to the memory and overwritten immediately. Based on the two baselines, a classification threshold can then be set. The most naive threshold would be the mean of the two baselines. An improved classifier is also introduced in Sect. 4.4.

For the actual probing, the attacker first writes the signature pages to their own VM's memory. Again, the attacker then needs to wait for some time to allow deduplication to take place. Afterwards, they overwrite all pages in the signature and measure the time this takes. This measurement is then compared to the classification threshold to determine whether the signature pages were deduplicated or not. If they are classified as having been deduplicated, the application version corresponding to the signature is likely being executed in another VM on the host. Otherwise, the application version is likely not being executed.

To increase the accuracy of the results, measurements can be repeated several times. In this case, the mean of the measurements should be used for any calculations or classifications. Especially for small signatures, repeating the measurements is necessary, which increases the time such an attack takes.

Note that if an attacker is interested in attacking a specific cloud service, they will first need to obtain a VM that is co-resident with a VM hosting it. Ristenpart et al. [12] found that this is feasible even in large commercial clouds.

Also, the attack will not reveal which specific VM a vulnerable application is running in. However, even without knowing the specific VM, it will help an attacker in narrowing down potential exploits to use for a targeted attack.

2.4 Related Work

Xiao et al. [15] demonstrate a covert channel using memory deduplication that allows two virtual machines on the same host to communicate with each other. They also show how the integrity of a guest operating system's kernel can be monitored from outside the virtual machine using memory deduplication.

Memory deduplication side-channels were first shown to exist by Suzaki et al. [13]. They also demonstrate that the side-channel can be used to infer if an application is running in another virtual machine. However, they simply use all pages of the binary for detection and do not analyse whether different versions of an application can be differentiated. Furthermore, they measure the write times for all pages separately and do not aggregate these.

Owens and Wang [11] present a memory-deduplication side-channel attack to detect the OS running on a co-resident virtual machine. Their approach to generating signatures is similar to the one we use to generate signatures for individual versions [9]: They compare the memory images of different operating systems and use unique pages as a signature. However, they survey only a small

number of OS versions, which are very different from each other. Furthermore, their approach works on the full memory of a running OS, thus requiring a high amount of manual work for each version to be included in a dataset.

Gruss et al. [4] demonstrate that it is possible to exploit memory deduplication side-channels from within a browser using JavaScript. Bosman et al. [2] apply this approach to read data from the memory of an end-user client computer running Windows 8.1 or 10, which use memory deduplication by default.

Cache-based side-channels can also be exploited to detect applications in co-resident virtual machines. Irazoqui et al. [7] demonstrate how the version of a cryptographic library running in another VM can be detected via a Flush-and-Reload attack. Their attack works by detecting whether a specific function of the target version is present in the CPU cache. Compared to our approach, this approach requires a lot of manual work to identify suitable target functions. Compared to memory deduplication side-channel attacks, such attacks have the advantage of being able to detect chunks of data that are smaller than a memory page. However, they require these to be present in the cache, i. e. the attacked virtual machine has to execute the target function.

Also, to detect a newly released version, signatures will first need to be generated. However, an attacker who has not yet included such a version in their dataset is unlikely to be aware of any vulnerabilities in such versions anyway.

In summary, memory deduplication side-channel attacks described in related work mostly aim to establishing a covert channel between colluding virtual machines or to reveal memory contents in another VM. Some techniques exist to detect applications, but the efficient detection of specific versions that is the topic of this paper has not been studied thoroughly so far.

3 Detecting Groups of Application Versions

In this section, we describe how to detect groups of versions in another VM via a memory deduplication side-channel. Groups are formed to obtain larger signatures, which will lead to more efficient detection. First, we will give an overview of the steps an attacker has to perform for the attack. Then, we describe how suitable groups can be identified and signatures generated for these.

3.1 Attack Procedure

The steps to probe for an application version matching a group signature are similar to that for probing signatures of indiviudal application versions (cf. Sect. 2.3).

The attacker must first establish baselines that correspond to the size of the signature for the deduplicated and the non-deduplicated case. Based on these, they can set a classification threshold. However, for the actual probing, a group signature (cf. Sect. 3.2) is now used instead of a signature for an individual version. For that, the attacker loads the signature into memory, waits for deduplication to occur, and then measures the time it takes to overwrite the signature. The result is classified according to the threshold to determine whether duplicate

pages of the signature are present on the host. These measurements should be repeated multiple times to increase the classification accuracy.

If the pages of the group signature have not been deduplicated, it is highly likely that there is no instance of an application version contained in the group running on the host. Alternatively, the deduplication mechanism might not have scanned the pages yet, e. g. because it is configured to only activate itself in case the physical memory is close to being full. If the pages of the signature are found to have been deduplicated, the attacker knows that at least one instance of an application version in the group is likely being executed in another VM. If some pages of the application have been swapped out, the signature may only be in memory partly, which will affect the overwrite time depending on the number of pages swapped. However, as long as only a small proportion of pages is swapped out (as should be the case for applications that are in active use), the application will still be detected as being present according to the threshold set earlier.

While the attack may not allow an attacker to get to know the specific version, the attack can narrow down the range of possible versions. In many cases, knowing that a version out of a group is present will be sufficient for an attacker, e. g. because a known security vulnerability affects all versions in the group. However, should an attacker not be satisfied with just knowing that one application version out of the group is present, they can still probe the exact version using signatures for the individual versions. The search space for this probing can be narrowed down to the group whose presence was detected. Thus, only signatures for application versions of the respective group need to be probed and not all different versions of an application.

3.2 Group Identification and Signature Generation

Our attack requires defining suitable groups of similar application versions. By forming groups, we can achieve a larger signature: When creating signatures for individual versions, we need to discard all pages also contained in other versions. However, when we create group signatures, we can combine versions that share pages and build a signature from these shared pages, while only pages contained in another version outside the group need to be discarded. This allows to detect the group more easily, as more pages can be utilized in the side-channel attack.

To generate the signature for a group G of application versions, we start with all pages contained in the first version of the group. As in signatures for individual versions, we remove any internal duplicate pages as well as pages containing only zero or one bits from the signature. As these steps can be performed on an application binary, irrespective of group membership, we define $pages(a)$ to return all pages of a binary excluding the internal duplicates and pages containing only zeroes or ones. For group signatures, we additionally remove all pages that are not present in *all* other versions in the group. Finally, any pages that are also contained in *any* version outside the group are removed.

Algorithm 1 shows how we identify groups. After initialising the data structures, the algorithm will add the first application version in the dataset to a new group (lines 4-5). Then, the algorithm begins to add additional versions to the

group in the order of their similarity to the first version (lines 9-18). For each potential combination of versions in the group, the algorithm determines the signature size. When the next candidate for addition to the group shares fewer pages with the first version in the group than are in the best signature found so far, the signature cannot be improved by adding further versions. The optimal combination of versions is then added to the group configuration and the algorithm proceeds to form a new group from the remaining unassigned versions.

Checking all potential combinations of versions can take very long for large datasets. Thus, it may be necessary to restrict which versions are considered as candidates for inclusion in a group, e. g. by imposing a lower limit on the similarity between a candidate and the first version in the group or by limiting the distance between versions in a group based on version numbers.

Algorithm 1 Group identification algorithm

Require: A ▷ set of all versions
1: $U := A$ ▷ Initialise set of unassigned versions
2: $\mathcal{C} := \emptyset$ ▷ Initialise group configuration (set of sets of versions)
3: **while** $U \neq \emptyset$ **do**
4: $a_g = a_v \in U : min(v)$
5: $G := \{a_g\}$
6: $G_{opt} := G$
7: $s_{opt} = \bigcap_{a \in G} pages(a) \setminus \bigcup_{b \in (A \setminus G)} pages(b)$
8: $V := U$ ▷ set of versions not considered for this group yet
9: **while** V $\neq \emptyset$ **do**
10: $a_c = a \in V : max\,|pages(a_g) \cap pages(a)|$ ▷ choose most similar version
11: $V := V \setminus \{a_c\}$ ▷ Remove from list of versions not considered for group yet
12: $G := G \cup \{a_c\}$
13: $s_{new} = \bigcap_{a \in (G \cup \{a_c\})} pages(a) \setminus \bigcup_{b \in (A \setminus (G \cup \{a_c\}))} pages(b)$
14: **if** $|s_{new}| \geq |s_{opt}|$ **then** ▷ If we have a new best signature size
15: $G_{opt} := G$ ▷ save new optimal group
16: $s_{opt} = s_{new}$ ▷ save new optimal signature
17: **end if**
18: **end while**
19: $\mathcal{C} := \mathcal{C} \cup \{G_{opt}\}$ ▷ Add optimal group to group configuration
20: $U := U \setminus G_{opt}$ ▷ remove group members from unassigned versions
21: **end while**

4 Evaluation

In the following, we will first describe the tools and datasets that we used to evaluate our attack. Then, we analyse the properties of the groups created by our algorithm and compare the signature sizes for groups to those for individual versions. We also evaluate an optimised classifier, which further increases the effectiveness of our attack. Finally, we discuss the complexity of the attack.

4.1 Signature Generation and Measurements

To generate group signatures and facilitate our experiments, we extended the software from our previous paper [9]. The code of our software is open-source and is available at https://github.com/jl3/memdedup-app-detection. We have extended our main analysis tool with the functionality to identify groups of versions of an application. It works according to the algorithm described in Sect. 3.2. Depending on the dataset checking all possible group configurations can take very long. Thus, users can configure two thresholds to decide which versions are considered as group candidates: (1) the minimum number of pages shared with the first version in the group and (2) the maximum distance – based on the canonical order of version numbers – between the first and last version in a group. The tool will also generate signatures for the identified versions.

The shell scripts for extracting binaries from distribution packages as well as our tools for performing timing measurements are also available online. These do not require any modifications for use with group signatures and thus remain unchanged compared to the individual version of the signature approach.

4.2 Datasets

For our evaluation, we use the same three datasets that were used to evaluate signatures for individual versions [9]:

- **Apache-Debian-x86_64**: contains all 131 versions of Apache released by Debian for the x86_64 platform, ranging from 2.2.11-5 to 2.4.25-3
- **sshd-Debian-x86_64**: contains all 185 versions of sshd released by Debian for the x86_64 platform, ranging from 4.2p1-7 to 7.5p1-5
- **sshd-multidist**: contains 11 releases of sshd 7.5p1 from the Arch Linux, Debian, Fedora, OpenMandriva and Ubuntu distributions. For Debian and Fedora, 3 and 5 releases are included, respectively.

4.3 Size of Group Signatures

Our results indicate that suitable groups can be found in both the Apache-Debian-x86_64 and the sshd-Debian-x86_64 dataset. Fig. 1 shows the signature size for the individual application versions in the Apache-Debian-x86_64 dataset. Fig. 2 shows the signature size for the groups that were identified in this dataset using the algorithm described in Sect. 3.2. All versions whose signature consisted of only few pages are now part of groups, resulting in much larger signatures.

Table 1 shows a comparison of the signature sizes for generating signatures for each individual version and group signatures. For all datasets, the signature size increases considerably when forming groups for similar versions. Bigger signatures result in better accuracy and efficiency of the attack. In all three datasets, small signatures were eliminated completely, with the smallest signatures consisting of 56, 37 and 174 pages. When generating signatures for each individual version, 32 to 47 % of signatures consisted of 5 or fewer pages.

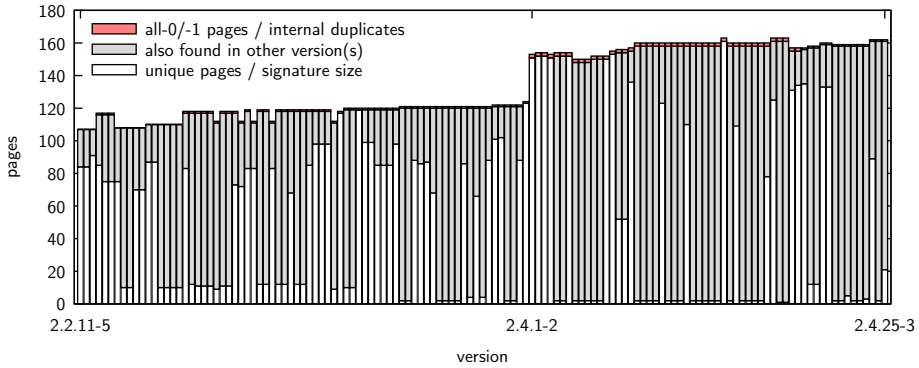

Fig. 1. Size of signatures for individual versions of the Apache-Debian-x86_64 dataset

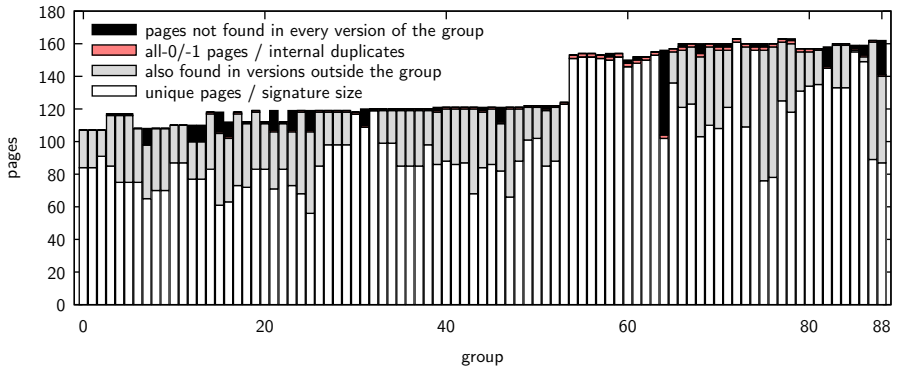

Fig. 2. Size of group signatures for the Apache-Debian-x86_64 dataset

Dataset	Number of signatures	Signature size				
		Minimum	Average	Median	Maximum	% ≤ 5
Apache individual	131	1	51.24	12	161	32.82
Apache groups	89	56	102.35	89	161	0
sshd indiviual	185	1	51.97	6	200	46.49
sshd groups	117	37	120.77	112	200	0
sshd-multidist idv.	11	2	105.82	174	202	45.45
sshd-multidist grp.	7	174	190.86	194	202	0

Table 1. Number and size of signatures for individual versions and groups of the Apache-Debian-x86_64, sshd-Debian-x86_64 and sshd-crossdist datasets.

Table 2 shows how many versions are contained in the group signatures formed by our algorithm and how closely related these are for the Debian datasets. For the Apache dataset, the average group size is 1.47 and the largest group contains six versions. Numbers for the sshd dataset are similar: The average group contains 1.58 versions, while the largest group contains five versions. Where groups contained more than one version, these were all different Debian patch levels corresponding to the same upstream versions released by the software's developers. No group contained versions belonging to different upstream releases of the software. The only exception was found in the sshd dataset, where one group contained three versions of two neighbouring upstream releases. Based on the canonical ordering of version numbers, the average distance between two versions in a group is less than one for both datasets. The maximum average distance of 19 was observed in a group containing two backport releases of Apache. For both datasets, $> 95\%$ of groups contained only adjacent versions.

This implies that forming groups in the dataset still allows the attacker to identify versions with a relatively high precision, while having to invest less effort. While the specific distribution patch level of a software can in many cases not be detected precisely using group signatures, the upstream release can be detected accurately in almost all cases. Where this is not the case, there will be a very low number of adjacent upstream releases matching the signature.

Even when an attacker needs to know the exact version, group signatures can be useful. An attacker can narrow down the candidate versions by probing for group signatures and specifically probe for the individual version afterwards.

Dataset	group size		groups with different	Avg. distance		skipped versions		
	Avg.	Max.	upstream versions	Avg.	Max.	Avg.	Max.	% = 0
Apache groups	1.47	6	0	0.63	19	0.30	18	95.5
sshd groups	1.58	5	1	0.5	3.17	0.08	3	95.7

Table 2. Distance between versions in signatures for the Apache-Debian-x86_64 and sshd-Debian-x86_64 datasets

4.4 Optimised Classification

The detection accuracies we used previously [9] are based on a rather naive classifier and are to be interpreted as the lower bound of what is achievable. For the naive classification, we use the mean of the two baselines for the deduplicated and non-deduplicated case as our classification threshold.

By selecting a better classification threshold, accuracy can be further increased compared to using the naive threshold. To find a suitable threshold, the ROC curve for a given number of measurements and signature size is first calculated. Then, an optimal point can be chosen from the curve. For our evaluation, we use the point where $\frac{TPR+FPR}{2}$ is minimal.

We apply this technique to the measurements taken for our previous paper [9]. Fig. 3 shows the ROC curves for (individual or group) signatures containing three

pages for different numbers of measurements. Curves for other configurations look similar in nature. For all configurations, a certain amount of false positives has to be accepted (i. e. signatures that were matched despite the corresponding application not being executed) to achieve a satisfactory true positive rate. As expected and in line with our observations, the area under curve increases with both the number of measurements and the number of signature pages.

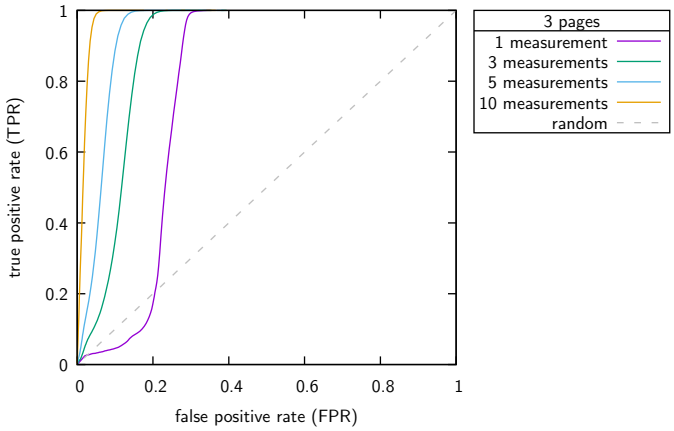

Fig. 3. ROC curves for three-page-signatures for different numbers of measurements

For small signatures, determining the classification threshold experimentally can increase the accuracy of the side-channel attack significantly for a given number of measurements. Fig. 4 shows a comparison of the accuracy between the naive and the optimised classifier for 1, 3 and 20 measurements for different signature sizes. Alternatively, the attack can be sped up by reducing the number of measurements, while still attaining the same level of accuracy. For instance, probing a signature of one page with an accuracy of $\geq 95\%$ requires the timing measurement to be repeated at least 93 times. When using an experimentally determined classification threshold, this can be reduced to 72 measurements, thereby decreasing the time taken to execute the attack by about 3.8 hours. For larger signatures, the absolute speedups are smaller: Probing a signature of 10 pages with an accuracy of $\geq 95\%$ takes two instead of four measurements. This means that the time needed to probe the signature can be halved from 43.7 minutes to 21.8 minutes. An accuracy of $\geq 99.5\%$ can now be achieved with four instead of nine measurements, i. e. the time for probing a signature decreases from 98.3 minutes to 43.7 minutes.

4.5 Attack Complexity

The use of group signatures will lead to larger signatures. Larger signatures can improve the efficiency of our side-channel attack, i. e. the number of measure-

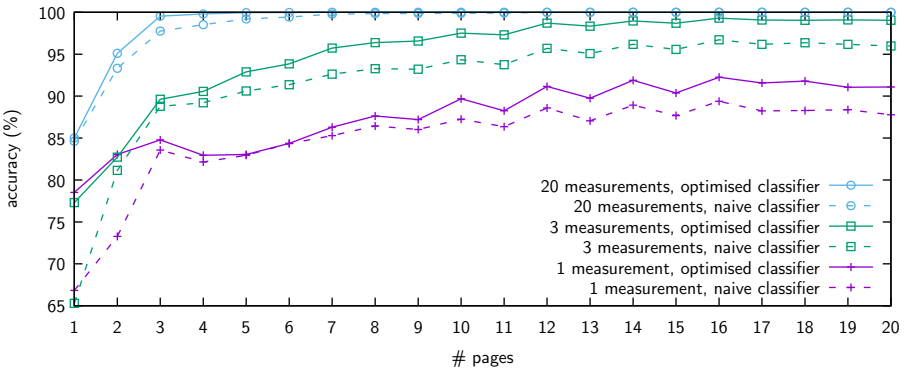

Fig. 4. Accuracy for naive vs. optimised classification for different signature sizes and number of measurements

ments can be decreased while maintaining the same accuracy. Alternatively, for a specific number of measurements we can be more certain whether an application is present. On the other hand, it will cause a slight decrease in precision, i. e. in some cases, we will be unable to tell which exact distribution patch level of a software is being executed.

More concretely, let us assume a desired confidence in the classification results of 95 %. Furthermore, let us assume one measurement requires 655.36 seconds, which is the maximum time that it takes KSM to scan the full host memory on Fedora/RHEL systems using the ksmtuned default configuration. For a signature of one page, this requires the timing measurement to be repeated at least 72 times even using the optimised classifier, i. e. the attack would take 13.1 hours. Probing signatures of two pages requires 20 measurements, which take just 3.6 hours. For signatures of three pages, at least seven measurements are required, i. e. the attack would take about 1.3 hours. Generating signatures for individual version leads to many small signatures for both, the Apache-Debian-x86_64 and the sshd-Debian-x86_64 datasets: For Apache-Debian-x86_64, there are two signatures of one page and 37 signatures of two pages For the sshd-Debian-x86_64 dataset, there are 11 signatures of one page and 46 signatures of two pages.

On the other hand, the smallest group signatures for these two datasets contain 56 and 37 pages, respectively. For these signature sizes, two measurements are sufficient to achieve 95 % accuracy. This means that the attack will take under 22 minutes to probe the group signatures for our datasets.

The larger signature size also allows to increase the accuracy of the results, while still achieving satisfactory runtimes. For signature sizes of 20 pages or more, just five measurements (resulting in a runtime of about 55 minutes) are sufficient to achieve an accuracy of $\geq 99.5\%$. For signatures consisting of only one or two pages, as often seen in signatures for individual versions, achieving such an accuracy would be prohibitively expensive.

5 Countermeasures

The countermeasures that can be employed to defend against the side-channel attack presented in this paper are similar to those that can be used against an attack using signatures for individual versions [9].

The operator of the host can disable memory deduplication altogether. This would eliminate the side-channel completely, but would also mean that any memory savings provided by deduplication are lost. Alternatively, deduplication could only be deactivated for pages containing executable code. While this only eliminates part of the memory savings possible by deduplication, it is hard to implement, requiring changes to the hypervisor and likely the guest OS.

Users can defend against the side-channel attack without cooperation of the host's operator by encrypting their VM's memory. This makes it impossible for the hypervisor to deduplicate the pages, unless another machine was using the same encryption key. An alternative defence mechanism is to move the memory contents relative to page boundaries, e. g. by employing sub-page ASLR.

Also, binaries could be modified so that they do not equal those used in other VMs. This can be achieved by compiling the applications using some uncommon compile flags or by inserting NOP opcodes into the binary.

Instead of preventing the detection of an application version, attackers could also be deceived by loading signature pages of versions not actually being executed into memory. Finally, one could also aim at detecting that a side-channel attack is taking place.

6 Conclusion

We have introduced a side-channel attack to detect groups of versions in co-resident VMs when memory deduplication is active on the host. While this attack does not allow to determine the exact version of an application as in our former work [9], it leads to significantly larger signatures compared to attacks aiming at detecting the exact version. As a result, an attack targeting groups of versions can be executed much faster than when probing for individual versions. Also, we have introduced an improved classifier, which helps to further speed up the attack by up to 50% depending on the size of the used group signatures.

Our results indicate that small signatures can be eliminated by targeting groups of versions (given a sufficiently large binary). For our evaluation datasets, all created group signatures contain at least 37 pages, while $> 30\,\%$ of signatures for individual versions contain less than five pages and many of them contain only a single page. Combined with our improved classifier, this means that we can reduce the number of measurements for an attack from 93 to 2 if a confidence in the results of $95\,\%$ is desired. This means that the attack takes 22 minutes instead of 17 hours to complete. We found the loss of precision to be minimal: While the exact distribution patch level of a binary cannot be detected in many cases, almost all groups contain only different distribution patch levels of the same upstream version of an application. The only exception that we found consists of releases from two neighbouring upstream releases.

A provider can remove the side-channel by turning off memory deduplication altogether or at least for pages containing executable code. Users can defend themselves by encrypting their VM's memory or using modified binaries.

In future work we will extend our approach to other operating systems and will evaluate it on a larger set of applications. Moreover, we will look into improved countermeasures that still allow memory deduplication but that will restrict the information gained by the side-channel.

References

1. Arcangeli, A., Eidus, I., Wright, C.: Increasing memory density by using KSM. In: Linux Symposium. pp. 19–28 (2009)
2. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup est machina: Memory deduplication as an advanced exploitation vector. In: IEEE Symposium on Security and Privacy. pp. 987–1004 (2016)
3. Fraser, K., H, S., Neugebauer, R., Pratt, I., Warfield, A., Williamson, M.: Safe hardware access with the xen virtual machine monitor. In: OASIS (2004)
4. Gruss, D., Bidner, D., Mangard, S.: Practical memory deduplication attacks in sandboxed javascript. In: ESORICS, Part I. pp. 108–122 (2015)
5. Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A.: Difference engine: harnessing memory redundancy in virtual machines. Commun. ACM **53**(10), 85–93 (2010)
6. Harnik, D., Pinkas, B., Shulman-Peleg, A.: Side channels in cloud services: Deduplication in cloud storage. IEEE Security & Privacy **8**(6), 40–47 (2010)
7. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Know thy neighbor: Crypto library detection in cloud. PoPETs **2015**(1), 25–40 (2015)
8. Lindemann, J.: Towards abuse detection and prevention in IaaS cloud computing. In: ARES. pp. 211–217 (2015)
9. Lindemann, J., Fischer, M.: A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In: ACM SAC (2018)
10. Milos, G., Murray, D.G., Hand, S., Fetterman, M.A.: Satori: Enlightened page sharing. In: USENIX Annual Technical Conference (2009)
11. Owens, R., Wang, W.: Non-interactive OS fingerprinting through memory deduplication technique in virtual machines. In: IEEE International Performance Computing and Communications Conference (IPCCC). pp. 1–8 (2011)
12. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: ACM CCS. pp. 199–212 (2009)
13. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory deduplication as a threat to the guest OS. In: European Workshop on System Security (EUROSEC) (2011)
14. Waldspurger, C.A.: Memory resource management in VMware ESX server. In: Symposium on Operating System Design and Implementation (OSDI) (2002)
15. Xiao, J., Xu, Z., Huang, H., Wang, H.: Security implications of memory deduplication in a virtualized environment. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 1–12 (2013)