Practical Memory Deduplication Attacks in Sandboxed Javascript

Daniel Gruss(✉), David Bidner, and Stefan Mangard

Graz University of Technology, Graz, Austria
daniel.gruss@iaik.tugraz.at

Abstract. Page deduplication is a mechanism to reduce the memory footprint of a system. Identical physical pages are identified across borders of virtual machines and programs and merged by the operating system or the hypervisor. However, this enables side-channel information leakage through cache or memory access time. Therefore, it is considered harmful in public clouds today, but it is still considered safe to use in a private environment, i.e., private clouds, personal computers, and smartphones.

We present the first memory-disclosure attack in sandboxed Javascript which exploits page deduplication. Unlike previous attacks, our attack does not require the victim to execute an adversary's program, but simply to open a website which contains the adversary's Javascript code. We are not only able to determine which applications are running, but also specific user activities, for instance, whether the user has specific websites currently opened. The attack works on servers, personal computers and smartphones, and across the borders of virtual machines.

Keywords: Memory deduplication · Side-channel attack · Javascript-based attack · Website fingerprinting

1 Introduction

Software-based timing attacks are side-channel attacks which exploit differences in the execution time to derive secret values used during the computation. These timing differences arise from the attacked software itself, different memory types or optimizations implemented in modern computers. For instance, cache attacks exploit the timing difference between a cache access and a memory access caused by a cache miss. An attacker process can measure whether a victim process has evicted one of the attacker's cache lines [13] or whether a victim program has reloaded a cache line the attacker previously evicted from the cache [5,20].

A similar timing difference can be observed between a regular memory access and a pagefault. Upon a pagefault, the operating system loads the data to the given location in virtual memory and returns control to the process. Apart from the difference in the memory access time, pagefault handling is transparent to the user process. This timing difference can be exploited to build a covert channel [17].

G. Pernul et al. (Eds.): ESORICS 2015, Part I, LNCS 9326, pp. 108–122, 2015.
DOI: 10.1007/978-3-319-24174-6_6

Suzaki et al. [16] presented page-deduplication attacks, which enable an attacker on the same physical machine, to determine whether specific programs are running, even across the borders of virtual machines. This is possible, because identical physical pages are merged by the operating system or the hypervisor. After a page is merged write accesses to this page cause a pagefault which is then resolved by the operating system. The timing difference the pagefault causes can be observed by the attacker program. Thus, the attacker learns that somewhere on the same physical machine another instance of this page exists.

JavaScript-based timing attacks have first been described by Felten et al. [3]. They were able to identify recently visited websites if website elements are fetched from the local browser cache instead of the network. A similar attack has been presented by Bortz et al. [2]. More recently Stone [15] presented attacks which exploit timing differences caused by the modification of HTML5 elements. Using their approach, an attacker is able to determine whether specific websites have been visited, and even read pixels from other websites.

In this paper, we present the first page-deduplication attack mounted in sandboxed Javascript. This allows a remote attacker to collect private information, such as whether a program or website is currently opened by a user. In contrast to existing Javascript-based timing attacks, we do not exploit any weaknesses in Javascript or the browser, but timing differences caused by optimizations in the operating system or hypervisor.

Javascript is a scripting language implemented in modern browsers to create interactive elements on websites. It is strictly sandboxed, so it is not possible to access files or system services. The language has no representation of pointers or the virtual address space layout, and less accurate timing information than native code. Oren et al. [12] already demonstrated in a Javascript-based cache attack that timer accuracy is high enough to distinguish cache hits from cache misses. Our attack is possible with less accurate timers, on a microsecond or millisecond basis.

To demonstrate the power of our attack, we show that we can accurately determine whether the user has opened specific websites. Our attack can be applied in a generic way to any system which employs page deduplication, independently of the CPU architecture and in particular independently of the CPU cache structure. This is a significant share of modern personal computers and smartphones.

With our attack, an adversary is not only able to perform the attack remotely through a website, on an arbitrary number of victims, but an adversary is also able to attack a variety of different devices in the same way. Thus, page-deduplication attacks no longer target one specific system, but instead target large numbers of internet users simultaneously. For instance, a website can detect which other websites a user has opened and thereby add more valuable information to user profiles. Furthermore, the attack causes negligible CPU and memory utilization and is thus, hard to detect if placed in a large Javascript framework.

We show that page deduplication must be considered a security threat on any system and not only on public cloud servers. Therefore, we conclude that the only effective countermeasure is to disable page deduplication.

Outline. The remaining paper is organized as follows. In Sect. 2, we provide background information on shared memory and page deduplication, as well as existing attacks. We describe the implementation of our attack in Sect. 3. In Sect. 4.1, we present the performance of our attack in a private cloud and in Sect. 4.2, we present results of our attack on personal computers and smartphones. We discuss countermeasures against page-deduplication attacks in Sect. 5. Finally, we conclude in Sect. 6.

2 Background

2.1 Shared Memory

Operating systems and hypervisors use shared memory to reduce physical memory utilization. Libraries which are used by several programs are loaded into physical memory only once, and are then shared among the processes using it. Thus, multiple programs access the same physical pages mapped within their own virtual address space.

The operating system makes use of shared memory in more cases. When forking a process, the memory is first shared between the parent process and the child process. As soon as one of the processes writes into the shared memory area, a copy-on-write page fault occurs and the operating system creates a copy of the according memory region. Note that write accesses into non-shared memory areas do not incur page faults and thus are significantly faster.

Shared memory is not only used when forking a process, but when starting instances of an already running program, or if a user program explicitly requests shared memory using system calls like `mmap` or `dlopen`. Mapping a file using one of these methods results in a memory region shared with all other proceses mapping the same file.

The form of shared memory we target in this paper is content-based page deduplication. The hypervisor or operating system scans the physical memory for pages with identical content. If identical pages are found, they are remapped to one of the pages, while the other pages are marked as free. Thus, memory is shared between completely unrelated and possibly sandboxed processes, and even between processes running in different virtual machines. If a process modifies its shared data, a copy-on-write page fault occurs and the hypervisor or operating system creates a copy of the memory region. Although searching for identical pages costs CPU time, page deduplication can increase the system performance, by reducing the number of block device accesses, as more data can be held in memory. Therefore, it is especially relevant in small systems like smartphones, besides the primary application in cloud systems.

2.2 Page-Deduplication Attacks

Page-deduplication attacks are a specific type of side-channel attacks, which exploit timing differences in write accesses on deduplicated pages. The first

attack on page deduplication was presented by Suzaki et al. [16]. They were able to determine whether specific applications are running in a co-located virtual machine in the cloud. Furthermore, they described the possibility of building covert communication channels between virtual machines by exploiting page deduplication.

In the basic attack scheme, an attacker is able to run a spy program on the victim's system. However, the spy program may be sandboxed or even run in a virtual machine. The spy program fills a page with data it suspects to find in the memory of the victim machine. The hypervisor or operating system constantly deduplicates identical physical pages. When the spy program tries to write to the page again, it can measure the elapsed time and infer whether a copy-on-write page fault occurs or not. Thus, the attacker can determine whether some other process on the same physical machine has an identical page in memory. Such attacks can be performed on both, binary code and static data as well as dynamically generated data.

Owens et al. [14] demonstrated that it is possible to efficiently fingerprint operating systems in co-located virtual machines by exploiting page deduplication. Since then, covert channels based on page deduplication [18,19] have been constructed and evaluated.

At the same time, researchers were able to build more efficient cache attacks if attacker and victim process share memory [5]. Page deduplication introduces a way to share memory with a victim process in a co-located virtual machine in the cloud. The possibility of performing a cache attack on a victim process across virtual machine borders has first been described by Yarom et al. [20]. Since then, several page-deduplication-based cache attacks have been demonstrated [8,9].

3 Description of Our Javascript-Based Attack

Our attack follows the same methodology as the page-deduplication attack presented by Suzaki et al. [16], which was implemented in native code. As our attack is implemented in Javascript, we face several new challenges, such as setting the content of a whole page in physical memory or detecting whether and when page deduplication has occurred.

As described in Sect. 2.2, the first step of a page-deduplication attack is to fill a page with data we expect to find on the system under attack. In native code, this done by filling a page-aligned region in an array with the according data. We found that Javascript engines in common browsers (Firefox and Chrome) perform a call to their own internal `malloc` implementation when creating a large array in Javascript. As a means of optimization, these `malloc` implementations align large memory allocations to page borders. Therefore, creating and filling a large array in Javascript works as in native code, in terms of our attack.

The second step is to wait until the operating system or hypervisor deduplicates our array. In our attacker model, the adversary performs the attack through a website on every visitor. Therefore, we cannot make assumptions about how long it takes until page deduplication has been performed. Instead, we repeatedly write the same value to the same position on the target page and measure

the time the write access took. We observed no influence of these repeated writes on whether the page is considered for deduplication. Thus, we can perform the deduplication check in a regular frequency.

The third step is the measurement of the write-access time, to infer whether a page has been deduplicated. This is done by measuring the time a write access on our own page takes. Based on the access time, we decide whether a copy-on-write page fault occurred. In native x86 code, we use the `rdtsc` assembly instruction for this purpose. In Javascript, we can use the function `performance.now()`. The accuracy of this function varies from sub-microsecond to millisecond range. If checking for deduplication of a single page in memory, our attack requires accurate microsecond measurements. However, usually more pages are attacked and thus less accurate timers are sufficient. For instance, when checking for deduplication of a 600 KB image, even an accurate millisecond-based timer can be used to implement our attack. Thus, `performance.now()` is sufficient to distinguish copy-on-write page faults from regular write accesses. Furthermore, `performance.now()` is available independently of the underlying hardware. Therefore, we can attack systems with a variety of different processors using the same Javascript code, such as personal computers or smartphones.

The only remaining question to perform our attack is how to know the data we want to fill the page with. Neither static code and data nor dynamically generated data is necessarily page-aligned. However, if the attacker knows the content of 8192 bytes contiguous in virtual memory, we can fill 4096 pages with data from these 8192 bytes, with every possible offset from the page alignment. Although this allows us to attack systems and programs with random offsets for the targeted data, we found that this is hardly necessary for most cases. For instance, we observed that images and CSS style sheets in websites are page-aligned in memory. This greatly facilitates our attack, as we can trivially extract the page content from a file and include it in our Javascript code.

The resulting attack applies to a wide range of scenarios, from mobile phone usage, over personal computers, to multi-tenant cloud systems. A user on a targeted system accesses a website, which contains the adversary's Javascript code. The Javascript code is then executed. After a few minutes, the Javascript code transmits the results back to the adversary. Our attack not only extracts sensitive information, like the browsing behavior of a user, but it is also extremely powerful due to its scalability. Once the Javascript code is deployed on a website, it automatically attacks anyone who accesses the website. We will demonstrate the attack in different scenarios in the following section.

4 Practical Attacks and Evaluation

In this section, we demonstrate our attack on a KVM-based private cloud server, on Windows 8 personal computers and finally on Android smartphones. In all scenarios, we use the same Javascript source code.

4.1 Cross-VM Attack on Private Clouds

Existing page-deduplication attacks have been demonstrated on public IaaS (Infrastructure-as-a-Service) cloud systems [16,18,19]. In this attack scenario, an adversary tries to be co-located on the same physical server with a targeted virtual machine. Once the adversary is co-located, the adversary extracts sensitive information from other virtual machines, e.g., whether vulnerable versions of specific server applications are running, or whether specific files are currently open.

Although public cloud providers reacted and now disable page deduplication in public IaaS clouds [7], we found that page deduplication is not yet considered a security problem on private cloud systems and servers. Popular Linux server distributions enable page deduplication, either by default, or automatically when reaching a certain memory usage level. For instance, we observed this behavior on Proxmox VE, Redhat Server and Ubuntu Server if configured as a KVM host.

Therefore, we demonstrate our attack in a private IaaS cloud. This is a realistic scenario, for instance in companies where users work on thin clients, connected to a virtual machine in the private IaaS cloud. In this scenario, a victim working in one virtual machine opens a website containing the malicious Javascript code, which is then continuously run in the background in a browser tab. Compared to existing attacks, our attack is possible even if the system does not allow users to start arbitrary programs, or if the user is well-educated to avoid executing programs from an untrusted origin. Furthermore, we want to emphasize that our attack is doubly sandboxed in this scenario, by running in the Javascript sandbox in the virtual machine separated from the targeted program in another virtual machine. That is, the adversary is able to extract sensitive information from the victim's virtual machine and other virtual machines on the same server.

The malicious Javascript code has to stay in memory until page deduplication has been performed. Depending on the system configuration, this can be between 30 sec and several hours. During our tests with 4 GB of physical memory and the default system configuration, we found that our memory is deduplicated after 3 min.

In order to evaluate the accuracy of our attack in Javascript code, we first perform the same attack in native x86 code. Figures 1 and 2 show write-access times on an array containing a 14 MB image file as measured by our native-code spy program within the same virtual machine, with low and high system load. This is equivalent to loading 3584 small images (2–4 KB) and measuring the deduplication of each of them. These write-access times quantify the accuracy of our page deduplication detection. When the image is loaded, we found no measurements to be lower than the expected copy-on-write access time. When having the image not loaded in the browser, we found less than 0.1 % of the measurements to be significantly above the expected regular write-access time. These 0.1 % can lead to false positive copy-on-write detection. However, as there is a timing difference of at least a factor of 10^3, we found an even smaller number of peaks to be above the lowest copy-on-write access times. We subsequently tested our attack using native code in the cross-VM setting and achieved the

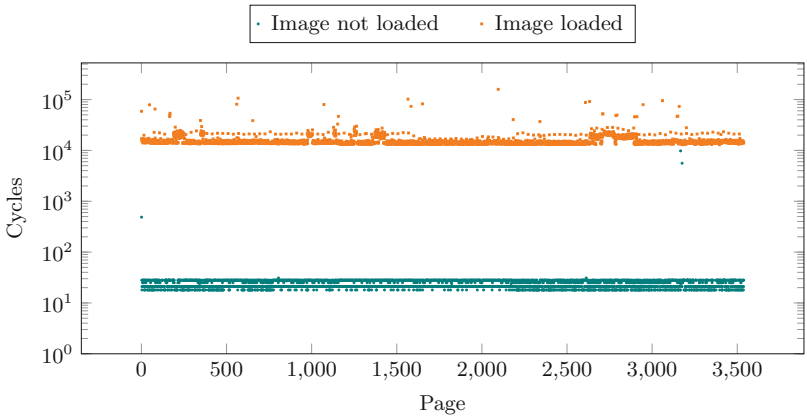

Fig. 1. Timings measured in native code on an otherwise idle Linux KVM virtual machine. The graph shows write-access times on an array containing an image file.

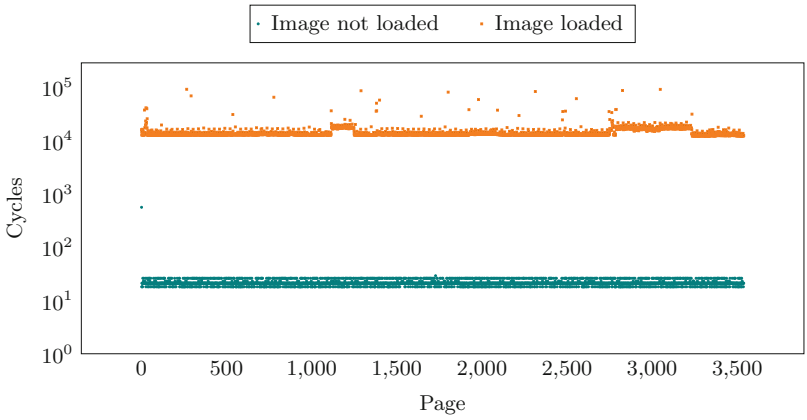

Fig. 2. Timings measured in native code on a Linux KVM virtual machine under high CPU load. The graph shows write-access times on an array containing an image file.

same accuracy. Therefore, we can accurately determine whether an image has been deduplicated and thus, has been loaded by a user.

Subsequently, we measured the performance of our Javascript-based attack. In Figs. 3 and 4, the write-access times on an array containing the same 14 MB image file are shown, but this time measured by our Javascript spy program. Even in with full system load and the browser under attack running in a different virtual machine, page deduplication was detected correctly in all of our measurements. However, in contrast to the native-code implementation of our spy program, we found up to 0.3 % of the pages to be falsely detected as deduplicated when having low system load and 1.1 % on average when having a high system load.

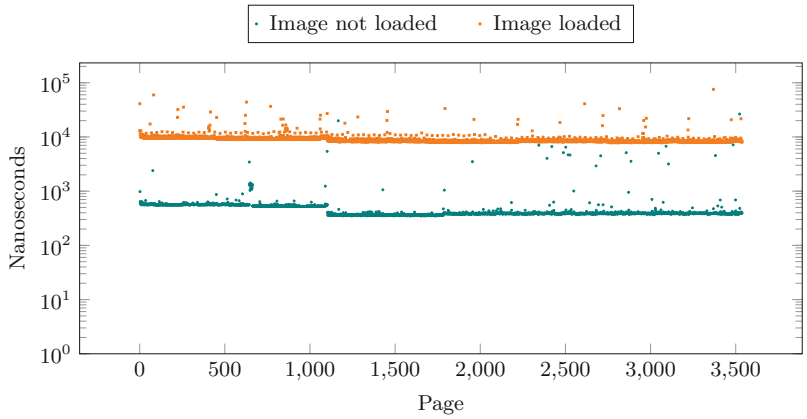

Fig. 3. Timings measured in Javascript on an otherwise idle Linux KVM virtual machine. The graph shows write-access times on an array containing an image file.

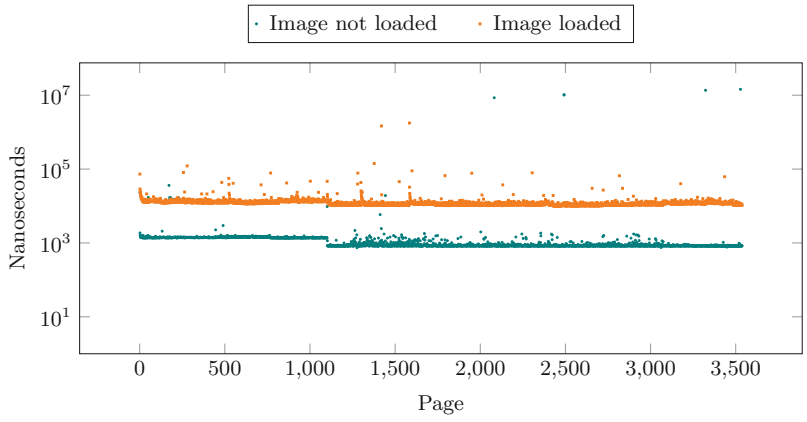

Fig. 4. Timings measured in Javascript on a Linux KVM virtual machine under high CPU load. The graph shows write-access times on an array containing an image file.

We performed this attack on recent versions of the most commonly used browsers, Chrome 40 and Firefox 31. As both browsers load the image file to a page-aligned location in memory, the attack works in exactly the same way and gives the same results for both browsers. Furthermore, we performed the same attack on a browser in a different virtual machine. Even in this setting, we did not find more false positives, and all deduplicated pages were detected successfully.

In order to demonstrate our attack on a real-world scenario, we determine the websites currently opened by a user. In this scenario, the adversary creates arrays containing image data of the websites to detect on the targeted machine. For demonstration purposes, we examined the 10 most-visited websites [1] and

chose an image or style sheet file from each website, to determine whether it is currently open in a web browser on the same machine. Furthermore, we generate several pages filled with zeros and several pages filled with random data, to measure reference timings for deduplicated and non-deduplicated pages. When the operating system or hypervisor has tried to deduplicate our pages, the zero-filled pages will have high write-access times, as they are deduplicated. The random-filled pages still have low write-access times, as each random-filled page is unique in the system and therefore not deduplicated. Some websites only contain very small or very few images. In these cases we combine several images to perform the attack more reliably. In all cases we had at least 24 KB of data to measure deduplication.

Figure 5 shows the write-access times to arrays containing image data from these websites, as well as the zero-filled pages and random-filled pages. We can clearly see which websites are currently opened in the browser, because of the higher write-access times, due to the copy-on-write page-fault handling. Based on such measurements, an adversary is able to spy on users' browsing behavior through malicious Javascript code, even across browsers and virtual machine borders.

4.2 Attack on Personal Computers and Smartphones

Our attack is even more precise if performed on a personal computer or smartphone, as the device under attack is only used by a single user at a time. There-

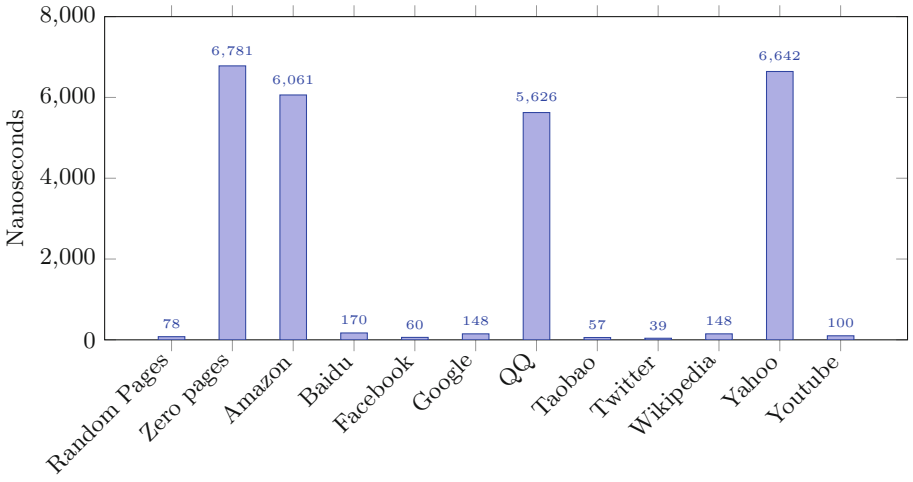

Fig. 5. Write-access times measured in Javascript inside a Linux KVM virtual machine, for images from frequented websites as well as random-filled an zero-filled pages. We measured high access times only for the currently opened websites: Amazon, QQ, Taobao, Wikipedia and Yahoo.

fore, we can create accurate profiles of single users. As in the cross-VM attack, the victim merely needs to access a website containing the malicious Javascript code.

This scenario is not only very simple and realistic, but moreover, it has a huge impact, as it can be applied to popular operating systems like Windows 8 on personal computers, or Android on smartphones. Windows 8 and 8.1 have a market share of around 15 % [11] on personal computers and have page deduplication enabled by default [10]. Android has a market share of 81.5 % [6] on smartphones, but it is device-specific whether page deduplication is enabled by default or not. However, Google recommends [4] that manufacturers enable page deduplication by default on memory-constrained devices, and many manufacturers follow this recommendation. Therefore, we assume that the number of smartphones having page deduplication enabled, and thus vulnerable to this attack, is significant.

In our attack, the malicious Javascript code runs continuously in the background in a browser tab. We found that on our Windows 8.1 test machine, page deduplication has been performed after 15 min on average. On our Android 4.4.4 test device, page deduplication has been performed after 45 min on average. As in the cloud scenario, we can then detect running applications and which specific version of an application is running, or even detect which specific websites are opened by a user. To evaluate the side channel, we again measure the deduplication detection rate for an image loaded in a browser. As we encountered problems with browsers on smartphones loading the 14 MB image file, we now use a 2 MB image file. This is equivalent to performing the same test with 512 small images (2–4 KB).

Figure 6 shows our Javascript-based measurements for the image file, using Firefox 36 on Windows. We can detect page deduplication almost as reliably as in the private-cloud scenario, with less than 2 % false positives. We are able to perform the attack without changes in Internet Explorer 11 and Firefox 36 on Windows, as both return micro- or nanosecond accurate timings via the `window.performance.now()` function. However, Chrome 41 on Windows only allows measuring time in milliseconds. Thus, we cannot measure the timing difference for each single page. Instead, we have to measure the time over a large number of pages at once. When measuring time over 150 write accesses at once, we are able to distinguish whether these 150 pages were deduplicated or not, with only millisecond timer accuracy.

When targeting website usage as a real-world scenario, the adversary creates arrays containing image data of the websites to detect on the targeted machine. As in the private-cloud scenario, we examined the 10 most-visited websites [1]. Figure 7 shows the write-access times to arrays containing image data from these websites, as well as the zero-filled pages and random-filled pages. Again, we clearly see which websites are opened, based on the higher write-access time.

When attacking Android smartphones, we found that although it takes up to one hour until deduplication is performed, the accuracy is not much worse than in the other scenarios we tested. We measured up to 0.8 % of false positives when having the image file not loaded in a browser and up to 0.5 % false negatives when having the image file loaded in a browser. This is slightly less accurate than

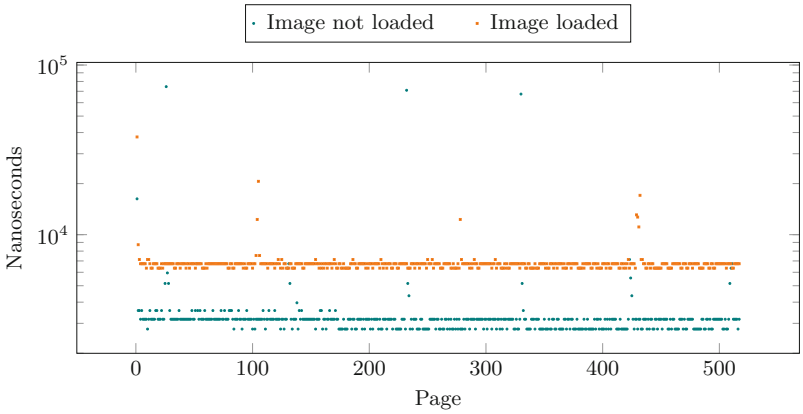

Fig. 6. Timings measured in Javascript on Windows 8.1. The graph shows write-access times on an array containing an image file.

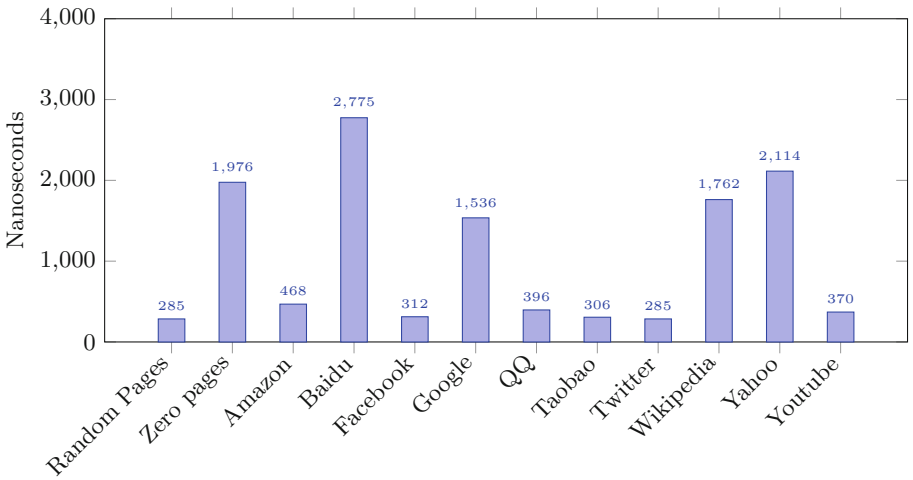

Fig. 7. Write-access times measured in Javascript on Windows 8.1, for images from frequented websites as well as random an zero-filled pages. We measured high access times only for the currently opened websites: Baidu, Google, Wikipedia and Yahoo.

in the other scenarios. Figure 8 shows the timing difference with and without the image loaded by a browser. Again, we examined the 10 most-visited websites [1]. Figure 9 shows the write-access times to arrays containing image data from these websites, as well as the zero-filled pages and random-filled pages. As in all other scenarios, we also see on Android which websites are opened, based on the higher write-access time.

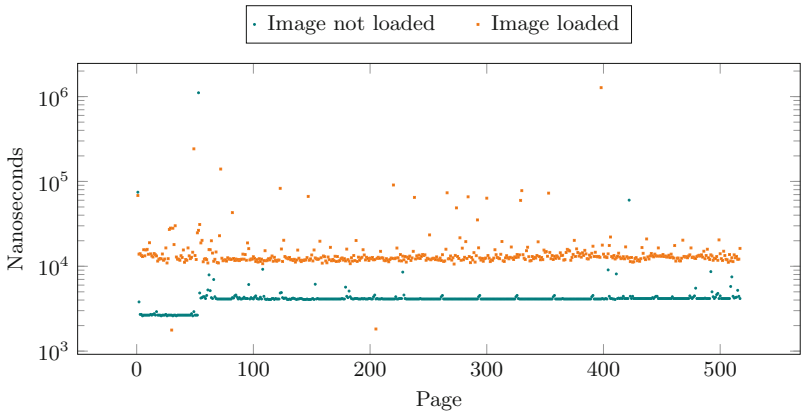

Fig. 8. Timings measured in Javascript on Android 4.4.4. The graph shows write-access times on an array containing an image file.

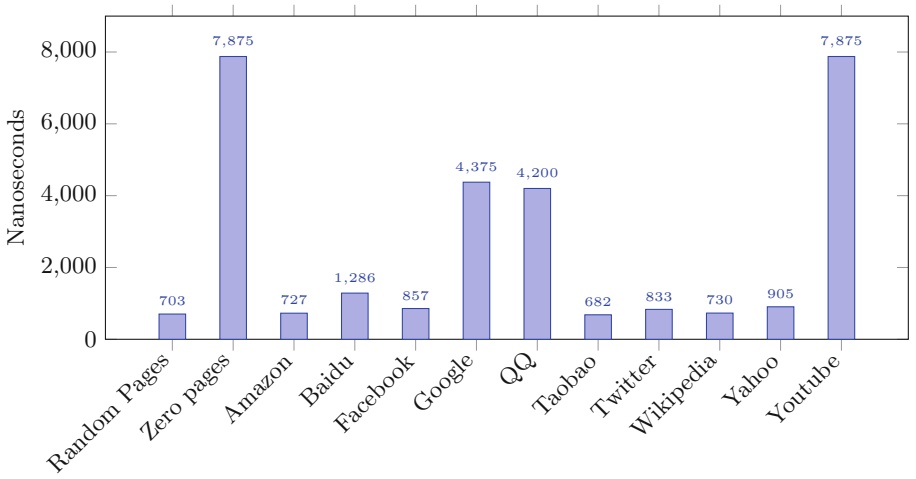

Fig. 9. Write-access times measured in Javascript on Android 4.4.4, for images from frequented websites as well as random an zero-filled pages. We measured high access times only for the currently opened websites: Google, QQ and Youtube.

5 Countermeasures

Our attack shows that even in sandboxed Javascript code, an adversary is able to extract significant sensitive information from real-world applications if the underlying system employs page deduplication. Our specific attack can be prevented on application level, i.e., in the browser executing the adversary's code, or in the applications under attack. However, countermeasures on this level incur

limitation of functionality. Disabling page deduplication is the only generic effective countermeasure against page-deduplication attacks.

It is possible prevent or at least weaken our specific attack in Javascript runtime environments by changing the way data is stored in memory, reducing the accuracy of timers, or disabling Javascript execution for untrusted code completely.

Our attack benefits from the fact that we are able to allocate page-sized physically contiguous memory areas. Thus, we are able to define the value of each byte on a physical page. Javascript engines could prevent this by adding small offsets to array indices, so that a few bytes per page cannot be controlled by the attacker. Consequently, the attacker-controlled memory will not be deduplicated. This would cause a small performance impact while impeding page-deduplication attacks in Javascript.

Another optimization we exploit is page alignment of large data, like images, as performed by modern web browsers. However, adding a random offset to the page alignment would not prevent our attack. The adversary can create 4096 copies of a targeted page, and thereby perform the same attack with only a small overhead. Furthermore, such a countermeasure would require manual modification of existing software, and would incur a performance penalty at the same time.

Oren et al. [12] suggested reducing the accuracy of Javascript timers as a countermeasure against Javascript-based cache attacks. However, a reduced timer accuracy would not prevent our attack. It is easily possible to measure the timing over a large number of pages and thereby invoke several copy-on-write page faults, resulting in timing differences in a millisecond range, which can be detected even with coarse-grained timers.

Our attack could also be prevented by disabling the execution of untrusted Javascript, i.e., disables Javascript on websites completely. However, this imposes a significant drawback on functionality of modern browsers and websites. In any case, the attack is still possible if implemented in a browser plugin or smartphone application, where Javascript-level countermeasures do not apply.

However, we think that any form of content-based page deduplication implies a security problem. As writable pages can be generated in any script language, sandboxed or not, and furthermore, we only require coarse-grained timer accuracy, we consider it insecure to perform page deduplication on writable pages. Considering only read-only pages has already been suggested by Suzaki et al. [16] as a countermeasure. Apparently, this countermeasure has not been implemented on the systems we attacked. We assume that one of the reasons is that the hypervisor or operating system is not able to distinguish between read-only pages and writable pages within virtual machines, one of the core applications of page deduplication.

However, not considering writeable pages would prevent page-deduplication attacks in Javascript or other script languages which do not support read-only data. Still, even in case that only read-only pages are merged, an attack could still be possible through browser plugins or smartphone applications on code and static data of targeted binaries, as they are able to load read-only pages or even

execute native code. Thus, disabling page deduplication completely is the only way to effectively prevent page-deduplication attacks as presented in this paper.

6 Conclusion

In this paper, we presented the first page-deduplication attack in sandboxed Javascript. In particular, the attack can be launched from any website. We show how the attack can be used to determine whether specific images or websites are currently opened by a user. We demonstrated the attack on private clouds, personal computers and smartphones. In all scenarios, it is even possible to mount the attack across the borders of virtual machines. Thus, we conclude that page deduplication must always be considered vulnerable to attacks as presented in this paper. Systems which have page deduplication enabled cannot be considered secure anymore.

The fact that page-deduplication attacks can be launched through websites marks a paradigm shift, from a targeted attack on a specific system towards large-scale practical attacks launched on a huge number of devices simultaneously. Therefore, we strongly recommend to disable page deduplication.

Acknowledgments.

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644052 (HECTOR).

Furthermore, this work has been supported by the Austrian Research Promotion Agency (FFG) and the Styrian Business Promotion Agency (SFG) under grant number 836628 (SeCoS).

References

1. Alexa Internet Inc: The top 500 sites on the web, March 2015. http://www.alexa.com/topsites
2. Bortz, A., Boneh, D.: Exposing private information by timing web applications. In: Williamson, C.L., Zurko, M.E., Patel-Schneider, P.F., Shenoy, P.J. (eds.) Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8–12, 2007. pp. 621–628. ACM (2007). http://doi.acm.org/10.1145/1242572.1242656
3. Felten, E.W., Schneider, M.A.: Timing attacks on web privacy. In: Gritzalis, D., Jajodia, S., Samarati, P. (eds.) CCS 2000, Proceedings of the 7th ACM Conference on Computer and Communications Security, Athens, Greece, November 1–4, 2000, pp. 25–32. ACM (2000). http://doi.acm.org/10.1145/352600.352606
4. Google Inc.: Android 4.4 platform optimizations. https://source.android.com/devices/tech/low-ram.html (Feb 2015)
5. Gullasch, D., Bangerter, E., Krenn, S.: Cache Games - Bringing Access-Based Cache Attacks on AES to Practice. In: IEEE Symposium on Security and Privacy - S&P, pp. 490–505. IEEE Computer Society (2011). http://dx.doi.org/10.1109/SP.2011.22

6. International Data Corporation: Android and iOS Squeeze the Competition, February 2015. http://www.idc.com/getdoc.jsp?containerId=prUS25450615
7. Irazoqui, G., Eisenbarth, T., Sunar, B.: Jackpot - Stealing Information From Large Caches via Huge Pages. IACR Cryptology, p. 970, ePrint Archive 2014 (2014). http://eprint.iacr.org/2014/970
8. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Fine grain Cross-VM Attacks on Xen and VMware are possible! IACR Cryptology, p. 248, ePrint Archive 2014 (2014). http://eprint.iacr.org/2014/248
9. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! a fast, cross-VM attack on AES. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 299–319. Springer, Heidelberg (2014)
10. Karagounis, B., Sinofsky, S.: Reducing runtime memory in Windows 8, October 2011. http://blogs.msdn.com/b/b8/archive/2011/10/07/reducing-runtime-memory-in-windows-8.aspx
11. Net Applications.com: Desktop Operating System Market Share, February 2015. http://www.netmarketshare.com/operating-system-market-share.aspx
12. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The Spy in the Sandbox - Practical Cache Attacks in Javascript. ArXiv e-prints, February 2015
13. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
14. Owens, R., Wang, W.: Non-Interactive OS Fingerprinting Through Memory De-Duplication Technique in Virtual Machines. In: International Performance Computing and Communications Conference - IPCCC, pp. 1–8. IEEE (2011). http://dx.doi.org/10.1109/PCCC.2011.6108094
15. Stone, P.: Pixel Perfect Timing Attacks with HTML5. Technical report, Context Information Security, June 2013. http://www.contextis.com/files/Browser_Timing_Attacks.pdf
16. Suzaki, K., Iijima, K., Yagi, T., Artho, C.: Memory Deduplication as a Threat to the Guest OS. In: European Workshop on System Security - EUROSEC, pp. 1–6. ACM (2011). http://doi.acm.org/10.1145/1972551.1972552
17. Warner, A., Li, Q., Keefe, T.F., Pal, S.: The impact of multilevel security on database buffer management. In: Martella, G., Kurth, H., Montolivo, E., Bertino, Elisa (eds.) ESORICS 1996. LNCS, vol. 1146. Springer, Heidelberg (1996). http://dx.doi.org/10.1007/978-3-319-11379-1_15
18. Xiao, J., Xu, Z., Huang, H., Wang, H.: A covert channel construction in a virtualized environment. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) the ACM Conference on Computer and Communications Security, CCS 2012, Raleigh, NC, USA, October 16–18, 2012, pp. 1040–1042. ACM (2012). http://doi.acm.org/10.1145/2382196.2382318
19. Xiao, J., Xu, Z., Huang, H., Wang, H.: Security implications of memory deduplication in a virtualized environment. In: 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24–27, 2013, pp. 1–12. IEEE (2013). http://doi.ieeecomputersociety.org/10.1109/DSN.2013.6575349
20. Yarom, Y., Falkner, K.: FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium, pp. 719–732. USENIX Association (2014). https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom