David Bidner

Timing Attacks on Memory Deduplication

Bachelor's Thesis

Graz University of Technology

Institute for Applied Information Processing and Communications
Head: O.Univ.-Prof. Dipl.-Ing. Dr.techn. Reinhard Posch

Supervisor: Dipl.-Ing. Daniel Gruß

Graz, May 2015

This document was written with Kile, is set in Palatino, compiled with pdfLaTeX and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

In this thesis we demonstrate how dangerous attacks against Memory Deduplication are and how accurate information about the content of memory can be determined. We perform attacks against the Memory Deduplication technique introduced by the Linux Kernel Machine. Additional to already known attacks, we implemented a JavaScript version. This allows us, for example, to detect when a user opens a specific website. We compare the native and JavaScript implementation and show that users of virtual machines, smartphones and personal computers are exploitable via Memory Deduplication. Therefore, this feature has to be disabled.

Contents

List of Figures

1 Introduction

Virtual server hosts and smartphones have much in common, for example their growing memory demand. On some mobile devices basic system applications consume almost the whole RAM. On servers, virtual machines running the same operating system use unnecessary memory.

Page deduplication is a technique used to reduce the number of identical pages in memory. This method is especially effective when multiple virtual machines are running (cloud computing). The operating system is looking for these pages and merges them to a single shared copy-on-write page. The Linux kernel provides this feature via the so called Kernel Samepage Merging (KSM), which is not only used by the Linux Kernel Virtual Machine (KVM) but also on many Android phones. However, Page Deduplication introduces serious security problems and threatens user privacy. A single process without any permission can retrieve information about other processes through timing measurements.

In this thesis, we show how dangerous Memory Deduplication is. The work is based on attacks published by Xiao et al. [Xia+13], [Xia+12] and Suzaki et al. [Suz+11]. These previous attacks depend on a program running on the victim host. We ported these attacks to JavaScript.

Most public virtual server hosts are aware that Memory Deduplication should be disabled. We point this out again and expand the knowledge about the harm which could be taken. In private-, small business areas and on smartphones Page Deduplication is still commonly used.

2 General

2.1 Memory Deduplication

As described by Rachamalla et al. [RMK13], Memory Deduplication reduces the usage of physical memory by merging identical pages into one. Memory Deduplication is used in some hypervisors like the Kernel Virtual Machine (KVM), VMware ESXi and XEN. The implementations of Memory Deduplication differ in details, but in general the outcome is the same. The host operating system looks at every single physical page and compares them. When it finds the same one twice it merges it to a so called copy-on-write page.

A common way to detect identical pages is to keep hashes of each page. If the same hash value occurs twice, the system applies a bytewise check. The time until a full memory scan is done depends on the implementation, configuration and hardware. A full scan on KVM, with 4 gigabyte memory and default configuration, takes about three minutes. Besides the saving of virtual memory, Memory Deduplication changes the way of how a guest system writes into memory.

2.1.1 Side Effects of Memory Deduplication

In general write accesses to physical memory are applied directly. In case of a deduplicated page, the guest system tries to write to the memory in the same way. This is recognized by the host via a page fault, enforcing a copy of the page. Afterwards the write is performed to the copy. These steps increase the time needed for a write access. Hence, it is possible for an adversary to detect such pages with accurate time measuring methods.

2.2 Measuring Write Access Time

On x86, we can use the Time Stamp Counter (TSC) to measure time, as it counts the cycles since the last CPU reset. The assembly instruction `RDTSC` returns the TSC in `EDX:EAX`. With this instruction it is possible to measure the time an access to physical memory takes accurately. However, out-of-order execution can reorder instructions and thus `RDTSC` can be executed later than expected. Therefore, it is necessary to use a serializing instruction, like `CPUID`, to enforce the previous instruction to complete. Another way would be to use memory fences, achieving the same goal. The `RDTSC` instruction is implemented on all x86 CPUs since the first Pentium CPU [Into9].

JavaScript provides the function `performance.now()` [Nc15], which returns a floating-point number with up to nanosecond precision. As our experiments show, this function is accurate enough to detect copy-on-write behavior via JavaScript. In general we use `performance.now()` like `RDTSC` in native code. The only difference is the return value, a nanosecond time stamp instead of CPU cycles.

3 Attacking Memory Deduplication in Native Code

To attack Memory Deduplication, we need to fill physical pages with memory usable for our attacks. One way to do that is using `malloc` or `mmap`. We use `mmap` to produce page aligned memory mappings. One could also create an array with the `__attribute__(aligned())` option of `gcc`. However, this does not give us the same flexibility as we load images dynamically and do not want to recompile for every test. We need to access the created pages with write accesses. Measuring the access time makes it possible to find out, whether a page was in copy-on-write state and therefore already in memory.

3.1 Detection of Page Deduplication

The simplest way to detect Page Deduplication is to produce the same page twice and write to one of them. If the write access time changes in a range of about ten thousand cycles the host has Page Deduplication active. In our detection step, we use zero pages and for comparison, pages filled with random numbers. Depending on the host system, the merging daemon needs up to twenty minutes to scan all pages in memory. When merging is done, we measure the write time to a byte for each physical page.

Figure 3.1 shows the difference between write accesses to zero pages to detect Page Deduplication. The red line indicates the higher access time caused by the copy-on-write page faults, the green line, about 10000 cycles lower, indicates the same write accesses but without any copy processes.

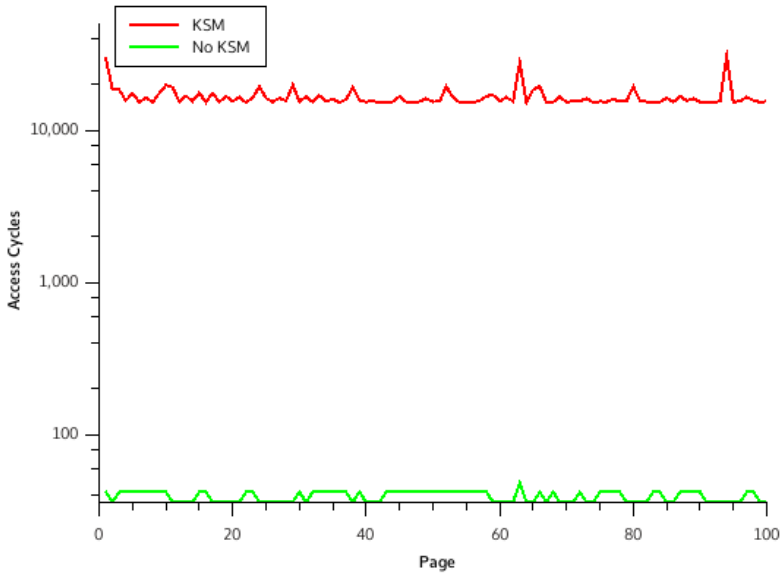

Figure 3.1: Access Cycles needed to write to zero pages, with and without Memory Deduplication active

3.2 Attacking KSM

To exploit the behavior of KSM, we `mmap` pages to check in our program. Possible attack scenarios are: find out what applications are running, what files are opened, whether a user logged in, even what sites are open in the browser.

We load parts of an application into memory, which are most likely to be static, in order to detect if it is currently running. These chunks of memory could be constants from an ELF binary or images from websites.

Figure 3.2 illustrates a binary detection, w3m in this example. In this figure, three states are shown. The green line indicates a closed binary, easily detected, as there are no peaks. After the binary is started on the system just a few memory pages get loaded, as shown by the red line. The black line shows what happens after the usage of the binary. All needed pages got loaded into memory.

Typical images for a website are used to detect whether a page is open in a browser or not. Figure 3.3 shows that `herold.at`, `amazon.de` and `willhaben.at` are opened in the victims browser as the access cycles to images placed on these websites are much higher than to the others. A possible attack would be to find out which sites are used by other workers in a company, as many use a shared environment with remote clients.

3.2.1 Operating System Noise

When a host has multiple virtual machines with a high CPU load, the time needed for a full scan increases. Additionally, there occur high peaks in access times. This leads to false positives when checking for a merged page. We introduced a threshold to prevent these kinds of false positives. If a write access takes much more cycles it is most likely because of noise and not due to copy-on-write behavior.

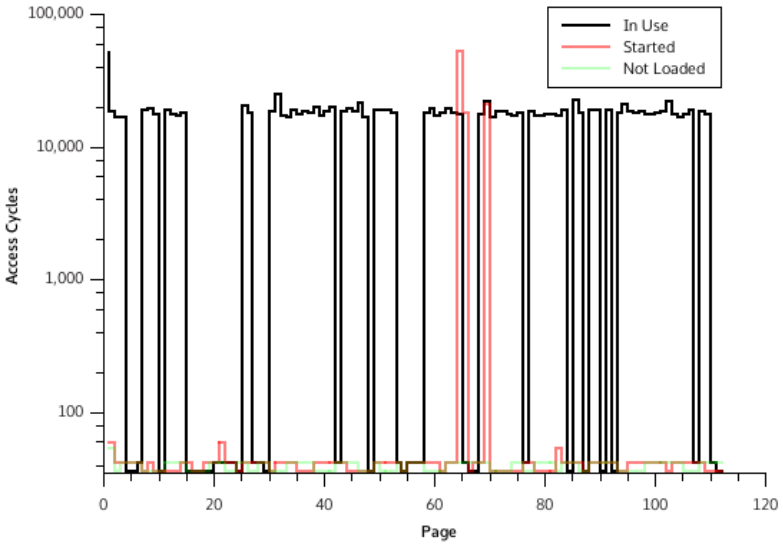

Figure 3.2: Comparison of write times to a binary in memory. We measured the time for three different states of the binary, as parts of it only get loaded after some usage.

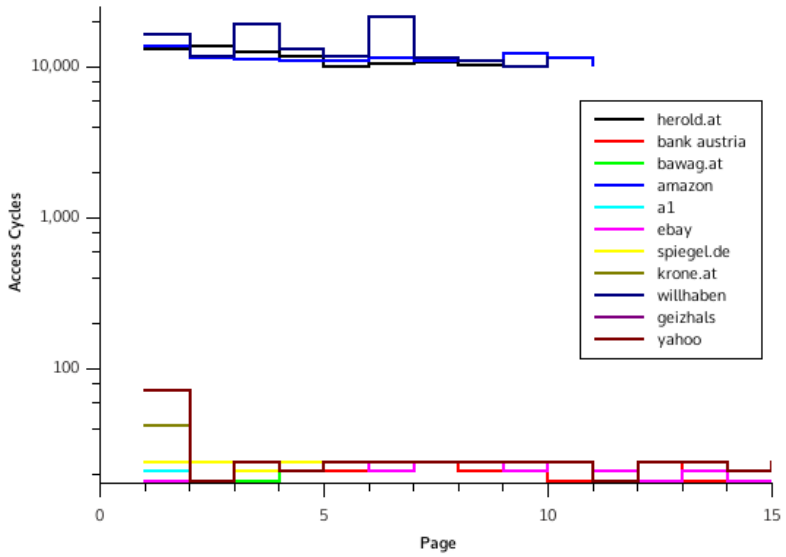

Figure 3.3: Detection of open websites. We measured access times to images found on some of the most popular websites in Austria [Int15]. In this test, the websites were opened with Chromium Version 41

Disk Cache Problems

When opening a file in GNU/Linux, the operating system loads the content into memory, to lower read times when using the file later on. In our case, we never want any other copies in memory. We trick the caching mechanism, to prevent this, by saving the pages we want to check as `ZIP` files and load the content at runtime into `mmaped` memory. Therefore we never get parts doubled in memory and thus, lower the number of false positives.

3.2.2 Accuracy of the Attack

In our experiments it was possible to determine copy-on-write page faults accurately. Even with high system load on the host, we achieved an accuracy of 99,4 percent inside one virtual machine. In our tests, we used a 14 megabyte image, i.e. the image spans over 3537 pages. We checked whether the image was loaded in the browser or not. We could not notice a variance in accuracy when increasing the system load on the host. Furthermore there were no differences when measuring accesses across virtual machines or inside the same one.

4 Attacking Memory Deduplication in JavaScript

Our experiments show that common browsers save images page aligned into memory. Additionally that it is the same for JavaScript arrays. The detection mechanism benefits from this knowledge, as we do not need to bruteforce the alignment. This behavior can easily be exploited remotely via hosting a malicious JavaScript on a website. With this attack, we show that Memory Deduplication on Android is even more dangerous than formerly known, as an attacker does not need to execute any application on the victim machine.

4.1 Page Border Detection

It is necessary to find out whether the `performance.now()` function is accurate enough to attack Memory Deduplication via JavaScript. We determined page borders by writing to each byte of zero memory pages. As shown in Figure 4.1, there are some peaks every 4096 bytes. The 4096 steps are marked by black lines in the background. This implies that JavaScript implementations store large chunks of memory page aligned.

Peaks occur due to pagefaults. Subsequent high measurements are due to the increased system load caused by the pagefault. Additionally, the time in the beginning is higher, because the Just-in-Time compiler of the web browser is applied later on.

The detection of page borders is not as precise as in native code and the number of false positives is higher. However, it is good enough to detect

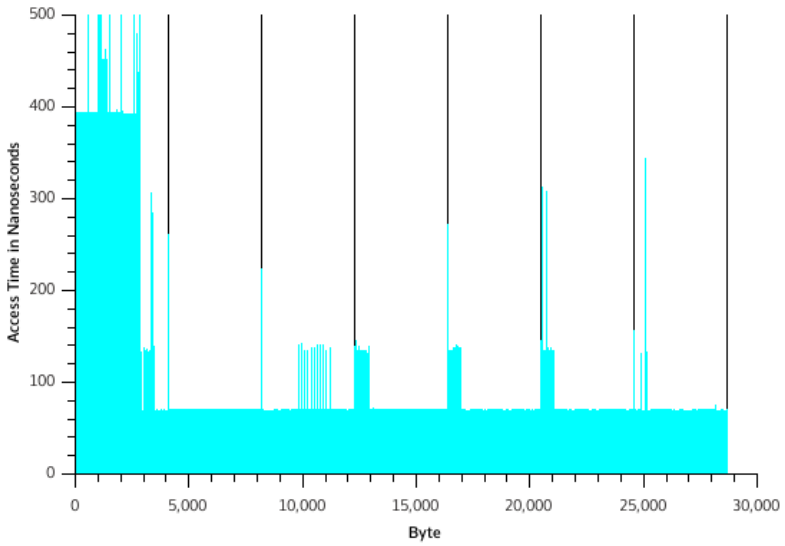

Figure 4.1: Detection of page borders in JavaScript. We measured access time to every byte of the mapped area which allowed us to determine exactly how large arrays get placed into memory in JavaScript

timing differences due to pagefaults especially when multiple tests are run.

4.2 Detection of Open Websites in Memory

Exploiting user behavior via JavaScript works generally as in native code. The only difference is that we do not need any zip preloader, as we hardcode the images into JavaScript arrays.

Figure 4.2 shows the difference in write-access times to image arrays. The high access times occur, because the `Ebay` and `Yahoo` homepages are already in memory. As reference, we used zero and random pages again.

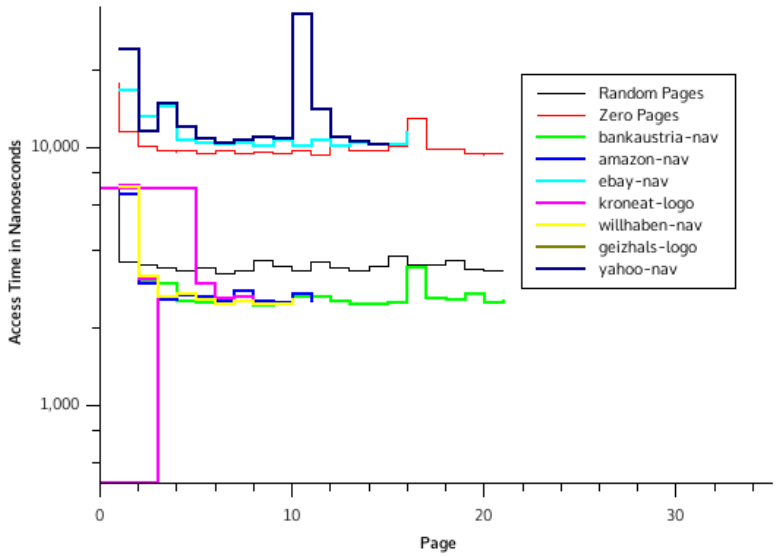

Figure 4.2: Detection of open websites in JavaScript. We measured access times to images found on some of the most popular websites in Austria [Int15]. In this test, the websites were opened with Chromium Version 41

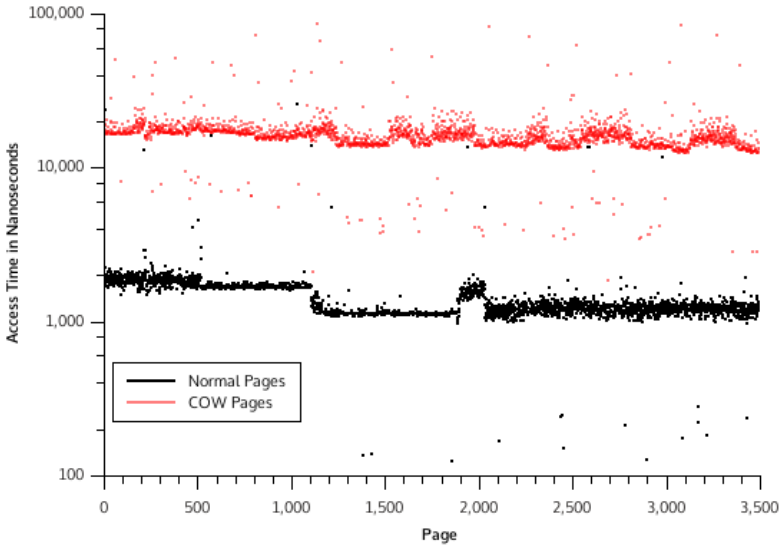

Figure 4.3: Difference of copy-on-write pages in JavaScript with high system load. We measured access times to 3537 pages to determine accuracy of our detecting mechanism.

4.2.1 Accuracy of the JavaScript Attack on a KVM Host

High system load slows down loading the JavaScript code. We achieved an accuracy of 97,68 percent when testing inside a virtual machine. The number of false positives is high for the first pages. It is possible to insert random pages at the beginning to compensate this. Our pages will then be checked by the optimized code, after these random pages. Figures 4.3 and 4.4 show the difference of timing caused by system load. Where black dots indicate a non open image, i.e. normal write accesses, and red dots show the higher time caused by pagefaults and the additional copy-on-write. Both figures show some false positives, as there are a few black dots in the upper area. This indicates that our attack does not depend on the system load.

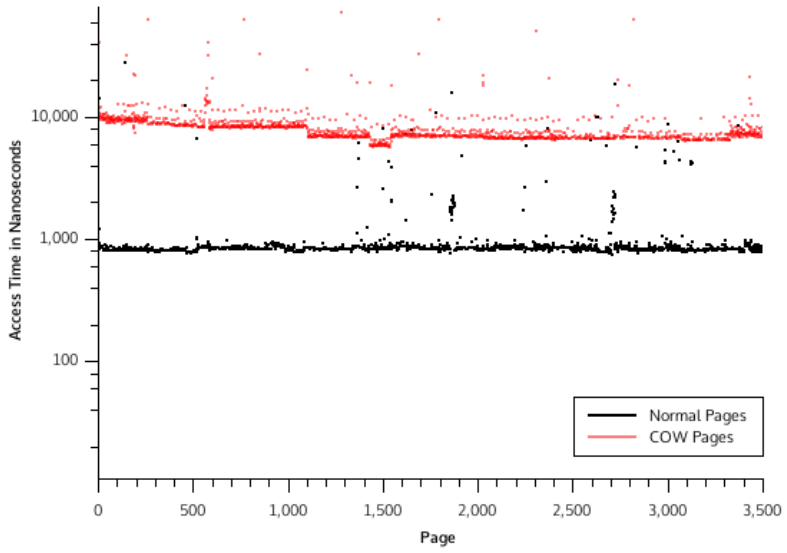

Figure 4.4: Difference of copy-on-write pages in JavaScript with low system load. We measured access times to 3537 pages to determine accuracy of our detecting mechanism.

5 Conclusion

In this thesis, we demonstrated that Page Deduplication can be detected easily and how much information leaks through this timing side channel. Operating systems keep large chunks of memory page aligned to lower access times, making the attack even easier. The number of false positives is very low. False negatives only occur when timing was too short for the merging process to complete a full scan.

Disabling `RDTSC` is not a good idea to fix this issue, as there are alternative methods to perform accurate time measurement. Additionally, it is possible to increase the timing differences up to milliseconds, by increasing the number of write accesses to memory.

Public hosting companies already disable memory deduplication per default. This increases the usage of memory, but also the security of the customers. The JavaScript implementation of the attack still threatens users of private clouds, deduplicating operating systems (Windows 8 [Cor15]) and smartphone users alike, because single websites can easily spy on them.

Memory is a major expense factor. However, companies need to balance pros and cons of money savings and exposing the users to danger of getting spied on. Based on our experiments, we recommend to disable Page Deduplication in any case!

Bibliography

[Cor15] Microsoft Corporation. *Reducing runtime memory in Windows 8*. Apr. 2015. URL: http://blogs.msdn.com/b/b8/archive/2011/10/07/reducing-runtime-memory-in-windows-8.aspx.

[Into9] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. 253669-033US. Dec. 2009.

[Int15] Alexa Internet. *Alexa - Top Sites in Austria*. Apr. 2015. URL: http://www.alexa.com/topsites/countries/AT.

[Nc15] Mozilla Developer Network et al. *Performance.now()*. Apr. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.

[RMK13] Shashank Rachamalla et al. "Share-o-meter: An empirical analysis of KSM based memory sharing in virtualized systems". In: *20th Annual International Conference on High Performance Computing, HiPC*. 2013.

[Suz+11] Kuniyasu Suzaki et al. "Memory deduplication as a threat to the guest OS". In: *Proceedings of the Fourth European Workshop on System Security, EUROSEC'11*. 2011.

[Xia+12] Jidong Xiao et al. "A covert channel construction in a virtualized environment". In: *the ACM Conference on Computer and Communications Security, CCS'12*. 2012.

[Xia+13] Jidong Xiao et al. "Security implications of memory deduplication in a virtualized environment". In: *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013.