Remote Memory-Deduplication Attacks

Martin Schwarzl
Graz University of Technology
martin.schwarzl@iaik.tugraz.at

Erik Kraft
Graz University of Technology
erik.kraft5@gmx.at

Moritz Lipp
Graz University of Technology
moritz.lipp@iaik.tugraz.at

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

***Abstract*—Memory utilization can be reduced by merging identical memory blocks into copy-on-write mappings. Previous work showed that this so-called *memory deduplication* can be exploited in local attacks to break ASLR, spy on other programs, and determine the presence of data, *i.e.*, website images. All these attacks exploit memory deduplication across security domains, which in turn was disabled. However, within a security domain or on an isolated system with no untrusted local access, memory deduplication is still not considered a security risk and was recently re-enabled on Windows by default.**

In this paper, we present the first fully remote memory-deduplication attacks. Unlike previous attacks, our attacks require no local code execution. Consequently, we can disclose memory contents from a remote server merely by sending and timing HTTP/1 and HTTP/2 network requests. We demonstrate our attacks on deduplication both on Windows and Linux and attack widely used server software such as Memcached and InnoDB. Our side channel leaks up to $34.41\,\mathrm{B/h}$ over the internet, making it faster than comparable remote memory-disclosure channels. We showcase our remote memory-deduplication attack in three case studies: First, we show that an attacker can disclose the presence of data in memory on a server running Memcached. We show that this information disclosure channel can also be used for fingerprinting and detect the correct libc version over the internet in $166.51\,\mathrm{s}$. Second, in combination with InnoDB, we present an information disclosure attack to leak MariaDB database records. Third, we demonstrate a fully remote KASLR break in less than 4 minutes allowing to derandomize the kernel image of a virtual machine over the Internet, *i.e.*, 14 network hops away. We conclude that memory deduplication must also be considered a security risk if only applied within a single security domain.

I. Introduction

Memory deduplication is a widely used technique to reduce memory utilization by detecting physical pages with the same content and merging them. Merged pages are marked as read-only and copy-on-write. If one of the merged pages is modified, a copy-on-write page fault is triggered, and the page is again copied to a new physical location. With the introduction of Windows 8.1, memory deduplication had become a default feature [58]. On Linux, kernel-same-page merging is used by kernel-virtual machines or if the `madvise` syscall is used with a flag indicating that the page is mergeable.

Previous work demonstrated memory-deduplication attacks performed by a local attacker in both local environments (*i.e.*, local native code execution) and the cloud (*i.e.*, local code execution in a virtual machine) [50], [4] exploiting page combining on Windows and kernel-same-page merging on Linux. Memory-deduplication attacks can detect co-location in the cloud [50], hide communication in virtualized environments [56], [57], fingerprint operating systems [41], fingerprint websites via JavaScript [16] and break ASLR on Linux as well as on Windows by exploiting pages with almost fixed content [4]. Bosman et al. [6] leveraged memory deduplication in combination with Rowhammer to escape from a browser sandbox. Razavi et al. [44] used memory deduplication to facilitate Rowhammer attacks on co-located virtual machines. Palfinger et al. [42] demonstrated that memory deduplication can also be exploited in file systems like ZFS. Lindemann et al. [27] demonstrated efficient fingerprinting via memory deduplication in co-located virtual machines. In concurrent work, Kim et al. [22] showed a KASLR break on virtual machines on VMWare ESXi. Following the recommendation of all these attack papers, memory deduplication was disabled on Linux and Windows by default.

More recently, vendors switched to more fine-grained security policies. Windows 10, for instance, again enables page combining by default but restricts it to only deduplicate within a security domain but not across security domains, stopping existing attacks. We also observe that the popular Ubuntu 20.04 Linux distribution enables kernel-same-page merging by default for KVM-based virtual machines. Memory-deduplication attacks with local code execution are considered out of scope in their threat model. Systems without local code execution (native or in a virtual machine) for the attacker can still be considered secure with these mitigation strategies. However, it remains unclear whether remote attacks *without local code execution* are possible.

Our work faces three challenges which have to be solved to perform remote memory-deduplication attacks:

- *C1: Remotely amplify latencies for non-repeatable events.* Remote timing attacks require high latencies in the side channel to deal with noisy networks. Page-fault-type interrupts cannot be arbitrarily repeated (e.g., for copy-on-write page faults, the page is copied and writeable after the page fault). Hence, existing amplification techniques are not directly applicable.
  All previous memory deduplication attacks focused on cross-domain deduplication. Deduplication within one domain is considered secure (Windows re-enabled it for that reason). Intra-domain deduplication is visible outside of the domain if the timing latency is exposed over a web server or public API to the attacker domain.
- *C2: Trigger and observe copy-on-write pagefaults in a victim domain that shares no memory with any attacker*

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February – 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23081
www.ndss-symposium.org

domain. All previous memory deduplication attacks require local code execution (in native or sandboxed code). Remote requests are usually not held in memory for a long time. To speed up, the access of frequent data, in-memory caching mechanisms like Memcached are used in websites.

- *C3: Find remote request paths that do not only keep attacker-controlled data in memory but also provide the attacker with control over alignment and in-memory representation.* To enable byte-by-byte leakage, a target is required that allows alignment changes as described by Bosman et al. [6].

In this paper, we solve the mentioned challenges and demonstrate the **first fully remote memory-deduplication attacks**, just using requests to an HTTP web server. Our attacks infer timing differences caused by copy-on-write page faults on the server from the latency of network requests and responses. We demonstrate attacks on default-configured and fully updated Windows (native) and Linux (virtual machines) installations using default-configured standard server software such as Memcached. We measure the capacity of our side channel in a remote covert channel scenario and achieve a transmission rate of $302.16\,\mathrm{B/h}$ in a local area network and $34.41\,\mathrm{B/h}$ over the internet, which is faster than comparable remote memory-disclosure channels (e.g., NetSpectre [48] achieved $7.5\,\mathrm{B/h}$ in a local area network).

We demonstrate three different remote memory-deduplication attacks, illustrating the potential of our technique. In the first attack, we disclose the presence of data on a remote server running Memcached. The information disclosure works by uploading data blobs into the key-value store, freeing the deduplicated item, getting the same item reassigned, and triggering a copy-on-write page fault by modifying the page's content. We also exploit this information disclosure channel for fingerprinting, *i.e.*, which shared libraries are used on the remote system. Our attack detects the correct libc version over the internet in $166.51\,\mathrm{s}$.

In the second attack, we present a fully remote KASLR break on a virtual machine running on a remote cloud machine. By targeting kernel pages that contain kernel addresses but have all remaining bytes of the page fixed, we can successfully derandomize the kernel offset of a Linux virtual machine. We show that we can not only mount this attack in a local area network setting using HTTP/1 but, moreover, leverage HTTP/2 to successfully break KASLR on a server that is 14 network hops away within 4 minutes. We emphasize that vendor responses to local KASLR breaks are often that KASLR is only meant as a mitigation for remote attacks.

In a third attack, we disclose database records byte-by-byte from a MariaDB database server with an InnoDB storage engine. Our attack works by crafting requests that create byte misalignments within target pages, allowing byte-wise content guessing. This attack is particularly dangerous as it leaks attacker-unknown memory contents from a remote server, similar as in powerful Spectre attacks [24], [48]. We can leak $1.5\,\mathrm{B/h}$ in a local area network.

We conclude that memory deduplication must also be considered a security flaw if only applied within a security domain and even if local attackers are excluded from the threat model. As our attacks are full remote attacks, we emphasize that the remote attack vector has to be mitigated as well. Consequently, we responsibly disclosed all of our attacks to the corresponding vendors and work with them on finding mitigations before the public release of this paper. We will open-source our tools on GitHub with the conclusion of the responsible disclosure [1].

Responsible Disclosure. We responsibly disclosed our findings to Microsoft, Red Hat, Canonical, and AWS, on February 8th, 2021. The issues are tracked under CVE-2021-3714.

Contributions. The main contributions of this work are:

1) We present the first fully remote memory-deduplication attacks and show that these must be considered a security flaw even if only applied within a security domain.
2) We show that we can remotely fingerprint shared libraries to infer the exact versions via Memcached in-memory databases.
3) We present a fully remote KASLR break on a Linux virtual machine running in the cloud within only 4 minutes.
4) We demonstrate a fully remote byte-by-byte memory disclosure attack on a MariaDB database server with an InnoDB storage engine, leaking $1.5\,\mathrm{B/h}$.

Outline. The remainder of the paper is organized as follows. In Section II, we provide the required background about memory deduplication and remote timing attacks. In Section III, we state a threat model and provide an attack overview. In Section IV, we present the attack primitives that we use for remote memory-deduplication attacks. In Section VI, we evaluate the performance of our remote memory-deduplication attacks in three case studies on Windows and Linux, targeting Memcached, MariaDB (with InnoDB), and the Linux kernel. In Section VII, we discuss the results and state-of-the-art mitigations for remote memory-deduplication attacks. We conclude in Section VIII.

II. Background

In this section, we provide background on memory deduplication, memory-deduplication attacks, and remote timing attacks, as well as Address Space Layout Randomization.

A. Memory Deduplication

Sharing memory is not only crucial for inter-process communication but also to reduce memory utilization and cache pressure. Modern operating systems use different techniques to use shared memory whenever possible. For instance, when creating a new process with `fork()`, the memory is marked as *copy-on-write*, meaning that it is first shared between parent and child process and only copied (*i.e.*, duplicated) when one of the processes attempts to write to it. This is implemented by marking the memory read-only and raising a page fault upon a write access. Another example is the loading of any type of file (including, e.g., a program or library binary files). The operating system keeps files in the page cache and maps them into all processes that request access.

[1] https://github.com/IAIK/Remote-Page-Deduplication-Attacks

Neither of these approaches leads to the deduplication of identical but dynamically generated memory pages. Hence, operating systems have introduced content-based memory deduplication, which regularly scans the entire physical memory for pages with identical content. All but one of the identical pages are released, while the remaining one is marked as copy-on-write. Content-based memory deduplication has traditionally been applied across all security domains on all major operating systems. On Windows, the mechanism is called page combining [58] and kernel same-page merging on Linux [3]. However, security research has revealed that this enables a range of attacks, as we discuss in the next sub-section.

B. Memory-Deduplication Attacks

In a memory-deduplication attack, the attacker first generates candidate pages for deduplication. If the attacker guesses the content of a page in memory fully correctly, it is deduplicated. Until the deduplication took place, the attacker repeatedly writes to the candidate pages (without changing the content). As soon as the deduplication took place, this triggers a copy-on-write page fault, increasing the access latency drastically. Hence, the access latency reveals whether a victim process had a page with the exact same content, *i.e.*, memory deduplication forms a content-probing oracle.

The first memory-deduplication attack, demonstrated by Suzaki et al. [50], was used to detect applications running in other virtual machines. Owens et al. [41] also exploited memory deduplication to fingerprint the operating system version via unique pages per operating system in virtual machines. Gruss et al. [16] showed that memory-deduplication attacks are possible from JavaScript running on a website opened in a browser. Barresi et al. [4] demonstrated that it is possible to break address space layout randomization (ASLR) on both Windows and Linux using memory deduplication. Razavi et al. [44] exploited memory deduplication to perform Rowhammer attacks on applications in virtualized environments. Bosman et al. [6] used memory-deduplication attacks to create more sophisticated exploits and used the ASLR break via memory deduplication to create an end-to-end JavaScript exploit which leverages Rowhammer to achieve arbitrary memory read and write. Oliverio et al. [40] proposed a mitigation against active memory-deduplication attack called VUsion, which enforces same behavior when accessing shared and non-shared pages, a write-xor-fetch policy, and random memory allocation. Lindemann et al. [27] showed another fingerprinting attack to detect co-location in virtual machines.

Palfinger et al. [42] showed that memory deduplication can also be leveraged in file systems like ZFS to fingerprint the operating system in the cloud of co-located machines. In concurrent work, Kim et al. [22] demonstrated a KASLR break on VMWare ESXi.

C. Remote Timing Attacks

Timing attacks were heavily researched in the last two decades. Since network connections are getting more and more stable, at higher transmission rates, as well as lower and more consistent latencies, remote timing attacks have become increasingly interesting for attack research. Brumley and Boney et al. [7] demonstrated that it is possible to extract SSL private keys over a local area network. Acıiçmez et al. [1] attacked AES via a remote cache based attack. In 2009, Crosby et al. [10] showed the possibilities of remote timing attacks and how to reliably determine the number of requests required to distinguish certain timing differences over the network. There were several remote timing attacks on AES [59], [21], [2], [47] following Bernstein's idea of attacking AES [5]. Van Goethem et al. [51] exploited timing side channels in browsers. Irazoqui et al. [19] showed that it is possible to exploit cache timing differences in TLS in a local area network. Van Hoef et al. [53] leveraged TCP windows to observe the exact size of a cross-origin resource. Van Goethem et al. [52] showed that remote timing attacks can be performed over the Internet by exploiting concurrency in HTTP/2 and observing the order the packets return, which depends on the server-side timing, instead of the client-side timing. Kurt et al. [26] showed that Data Direct I/O can be used in combination with Remote Direct Memory Access to spy on keystrokes during SSH sessions.

More closely related to our work is Schwarz et al. [48], who showed that Spectre attacks are possible over the network if certain gadgets exist on the target system. Similar to the most powerful attacks we present, they can leak arbitrary data from an execution context. They achieve a leakage rate of up to $7.5\,\mathrm{B/h}$, which can be sufficient to leak a cryptographic key over the time frame of multiple hours.

D. Address Space Layout Randomization

To exploit memory corruption bugs, the knowledge of addresses of specific data is often required since address randomization is applied in both user space and kernel space. Over the past years, different side-channel attacks allowed to reduce the entropy of the randomization or to break it entirely. Hund et al. [18] measured the execution time of page-fault handling to observe which kernel addresses are mapped and thus cached in the TLB. Jang et al. [20] used hardware transactional memory to observe the same effect. Other software-based side channel attacks exploited predictors [13], [28], side channels introduced by mitigations against other attacks [9], the power consumption of the processor [29], and other microarchitectural properties [17], [25], [14], even from JavaScript [15], [8]. As a consequence of these local attacks on KASLR, operating system vendors but also parts of the academic community considered KASLR only as a defense against remote attackers. In remote attacks, KASLR indeed is still considered a valuable line of defense since the attacker cannot as easily probe the address space as with local attacks.

III. Threat Model & Attack Overview

In our threat model, the attacker has no ability to execute code on the target machine: not natively, not in a virtualized environment, and also not via JavaScript [6], [16] or another scripting language. However, the attacker can provide attacker-controlled content to the remote target, e.g., a network request the attacker sends to the host with content the attacker controls.

We assume that the victim keeps the attacker-controlled content in RAM. This occurs, for instance, if the attacker sends network requests that are cached in request pools, or binary large objects provided to a web application and later on stored in a database or cached in a buffer.

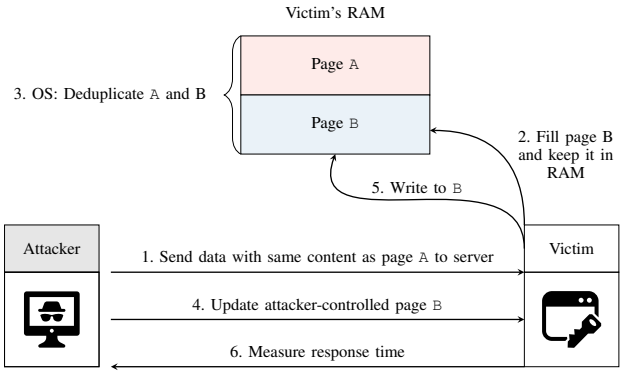


Fig. 1: Overview of a remote memory-deduplication attack.

We assume that memory deduplication techniques are active on the victim's machine. We emphasize that this is the case under default settings on current Ubuntu Linux installations (kernel-same-page merging for virtual machines) and on current Windows installations (page combining).

We make no assumptions about software bugs, for instance, memory safety violations in the applications we analyze.

Attack Overview. Six steps are required to perform a remote memory-deduplication attack as illustrated in Figure 1. First, the attacker sends a request to the victim with a page of data (page `B`) containing the same content as a page already present in memory (page `A`). Afterwards, the attacker waits for some time until the two pages are merged by the operating system and point to the same physical address. Next, the attacker updates the attacker-controlled data and triggers a page-fault on the victim application. Depending on the response time of the victim, the attacker observes whether the page was deduplicated or not.

Difference to already presented attacks. All of the previous presented attacks [50], [41], [16], [4], [6] require local code execution via a native binary or JavaScript and co-location to the victim's machine. Remote memory-deduplication attacks extend the scope by enabling attacks on remote web servers by exploiting an API that allows uploading of attacker-controlled data and place it into the main memory such that it might be deduplicated. Comparison of state-of-the-art memory deduplication attacks to our work is listed in Table I. While some of the techniques shown by previous work are similar, we solved those challenges for memory deduplication attacks in the context of a remote attacker. As evidenced by other fully remote attacks [48], [52], specific timing requirements and the applicability to many-hop internet connections, remain a challenge that is only solved for specific cases.

IV. Attack Primitives

In this section, we describe our basic attack primitives and define the requirements for a remote attacker to perform a fully remote memory-deduplication attack without execution of any attacker-controlled code on the victim system.

The main primitives for our attack are memory deduplication being enabled, a web service/API that lets a remote attacker read/modify data stored in RAM and an accurate remote timer that allows distinguishing the round-trip time of the network packets.

A. Memory Deduplication

Page combining. Page combining was introduced in Windows 8.1. On Windows, a special kernel thread scans over the whole memory to detect pages that have identical content [58]. This scan is triggered about every 15 minutes on Windows 10 [58]. If pages with identical content are found, the pages are combined to a single page to save memory. The page-table entries of the pages then point to one of the two pages, which is then shared across processes and marked as read-only and copy-on-write. When writing to this shared page, a copy-on-write fault occurs, and a new copy of the page is created for the writing process [58].

Page combining can easily be disabled via the Windows registry or using Powershell, e.g., using the `Disable-MMAgent` command. Page combining was temporarily disabled for security reasons after several memory-deduplication attacks were discovered [4], [6], [16]. However, page combining was re-enabled on Windows more recently and is active on desktop machines by default, as well as on server machines if the `full` terminal server role is enabled. In addition, a Windows 10 process has the possibility to disable page combining [37].

We observed this effect by checking all terminal server options in Windows 2016 (Version 1607, Build 14393.693) and Windows Server 2019 (Version 1809, Build 17763.737). We also empirically validated that for Windows 10 Professional 20H2 19042.746 and Windows 10 Home 19041.746 page combing was active by default. On Microsoft's Azure Cloud [36] it is also possible to acquire such Windows Server VMs with this configuration. We created a Windows 2019 Server (Version 1809, Build 17763.1697) and can also confirm that page combining is enabled after setting the full terminal server role. On Windows, it is also possible to force page combining using the `RtlAdjustPrivilege` and `NtSetSystemInformation` functions.

Linux Kernel Same-Page Merging Kernel-Same-Page Merging (KSM) is the counterpart of page combining on Linux [3], [45]. KSM is enabled and mainly used for Kernel Virtual Machine (KVM) virtualized machines, for instance, on Red Hat Linux [45]. On Ubuntu 20.04, we observed that when `qemu-system-common` with KVM support is installed on a host machine, `KSM_ENABLED` is set to `AUTO` in `/etc/default/qemu-kvm`, enabling KSM per default for non-virtualized instances. We also set up an Ubuntu 20.04 server image and observed the same behavior after installing QEMU. Like on Windows, a kernel thread scans over the memory and merges pages with identical content to a single page, which is then marked as copy-on-write [45].

On Linux, only pages are merged that are marked as mergeable, *i.e.*, using the `madvise` syscall and setting the `MADV_MERGEABLE` flag [3]. This is the default for pages of KVM virtual machines. The user can configure how many pages should be scanned per invocation (`pages_to_scan`). The default value on a Ubuntu 20.04 is 100 `pages_to_scan` in a time interval of 200 ms. Therefore, in the optimal case, up to 500 4 kB pages can be deduplicated per second. Figure 2

Attacks	Location	Environment	Local	Type	Attack Type	Performance
Suzaki et al. [50]	Co-located	Cross-VM (Cloud)	Yes	Native binary	Fingerprinting	-
Owens et al. [41]	Co-located	Cross-VM (Cloud)	Yes	Native binary	Fingerprinting	-
Gruss et al. [16]	Remote	Browser/Cross-VM (Cloud)	Yes	JavaScript	Fingerprinting	-
Barresi et al. [4]	Remote	Cross-VM (Cloud)	Yes	Native binary	ASLR break	8.7 days
Bosman et al. [6]	Remote	Browser (Same-machine)	Yes	JavaScript	Bytewise leakage, ASLR break, Rowhammer	2.75 h
Lindemann et al. [27]	Co-located	Cross-VM (Cloud)	Yes	Native binary	Fingerprinting	1.8 h
Kim et al. [22]	Co-located	Cross-VM (Cloud)	Yes	Native binary	KASLR break	12 min
Our work	Remote	**Internet/Local-NW**	**No**	**None**	Bytewise leakage, KASLR break, Fingerprinting	1.5 B/h (Local-NW) / 4 min / 166.51 s

Location: Attacker's location Local: local code execution Type: Type of local code execution Perf: Reported Attack Performance

TABLE I: Comparison of state-of-the-art memory deduplication attacks.

illustrates the required time for a single page being deduplicated, with a different number of pages_to_scan set, and the default value of 200 ms for sleep_millisec. We evaluate the deduplication time for a single page depending on the scanned pages on a remote server equipped with an Intel Xeon E3-1240 running Ubuntu 20.04. However, it is recommended to increase the number of `pages_to_scan` to increase the deduplication performance [49]. The tool KSMtuned sets the time interval per default to 10 ms and increases `pages_to_scan` to 1250 [45]. This would lead to a maximum 512 MB being deduplicated per second. We asked a **cloud provider**, which hosts multiple hundred thousand websites, for the KSM config used in **production**. The cloud provider uses a configuration of `sleep_millisecs=30,pages_to_scan=500`, leading to at maximum 65.84 MB (16500 pages) being deduplicated. The average time after a single page is deduplicated with that configuration is 34.57 s (n = 10, σ = 6.3%).

B. Service/Web API.

We assume that the victim machine provides network-accessible services, e.g., a REST API, enabling users to store and modify data blobs. There are no restrictions in the way these data blobs are controlled, *i.e.*, the user could either upload and replace files or send strings to the server, as long as the memory location of the data blob does not change.

C. Remote Timer.

To get the best possible low-latency timing information, we use the hardware timestamps from the network interface card. We measure the timing difference between the last packet sent and the first response byte received from the server (tcp_flags=PUSH,ACK).

The victim side (which the attacker cannot control) runs under default configuration. However, on the attacker side (that is under full control of the attacker), we disable the following optimizations in the Linux network parsing `sudo ethtool -K enp3s0 tso off gso off gro off`. These options disable offloading of TCP packets to the network interface card. Offloading might influence the timestamps on the attacker (receiver) side. We observed for some network interface cards that due to receiver side packet coalescing, the TCP receive timestamp of the first received packet might be overwritten. To ensure that no coalescing happens, we developed a kernel module which disables packet coalescing on the receiver side, for network cards which have this problem.

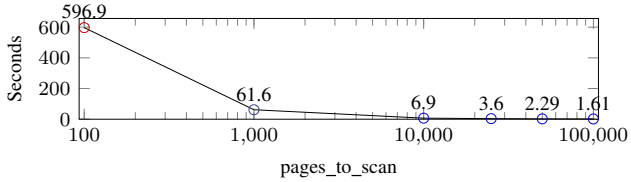


Fig. 2: The deduplication time of a single 4 kB-page strongly depends on the number of pages_to_scan (sleep_millisecs=200).

Network Timestamps. We found that one of the bottlenecks of remote attacks is the limited number of HTTP requests which can be sent using a simple HTTP requests library like `pyrequests`. Therefore, we use asynchronous IO mechanisms to increase the number of requests per second. Furthermore, we observed that Wireshark's TCP-field `tcp.time_delta` reflects the timing difference between copy-on-write pages and non copy-on-write pages best. This field calculates the timing difference between two captured packets. Compared to the network timestamp read from the NIC, we require only 20 requests instead of 40 to distinguish 16 overwritten copy-on-write pages over 14 hops in the internet to build a histogram.

D. Attack Setup.

For all our case studies, we use the following setup for our local and remote scenario.

Local Scenario. The local victim machine uses an i7-6700K processor with Ubuntu 20.04 (kernel 5.4.0) and runs QEMU 4.2.1 with KVM support enabled and virtualization extensions enabled. Co-located in the same local area network, we have our attacker machine, which also uses an i7-6700K processor and Ubuntu 20.04 (kernel 5.4.0). For the Linux setup, we host a virtual machine with KVM running Ubuntu Server 20.04 LTS (kernel 5.4.0-53-generic).

Remote Scenario. In addition, we used a remote Linux server by Equinix [12], running on an Intel Xeon E3-1240 CPU. We installed the same virtual machine on the Linux server. For our Linux machine, which was located in Amsterdam, we observed 14 network hops.

We created a virtual machine on Microsoft Azure of size Standard D4s v3 [36] and set up a Windows Server 2019 (Version 1809, Build 17763.1697) with page combining

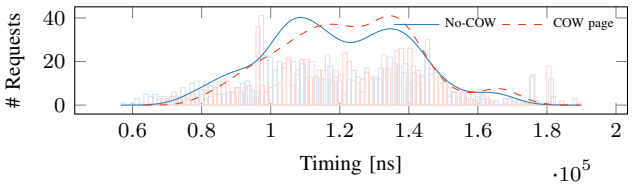


Fig. 3: Timing distribution of a single deduplicated page of a virtual machine in a local area network scenario on Linux KVM (n = 1000).

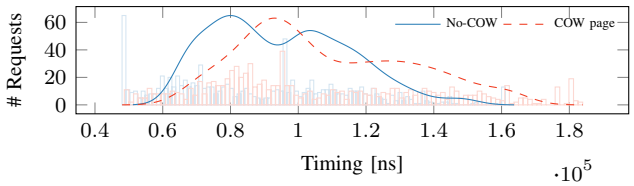


Fig. 4: Timing distribution of a single deduplicated page of a virtual machine in the internet(14 hops) (n = 1000).

enabled. We observed 28 hops, using the nmap traceroute command, between our network and the Windows 2019 server virtual machine, which was located in Amsterdam. We use the same attacker machine from our local setup to perform the internet attacks.

Settings. To estimate the highest possible capacity of our remote covert channel, we try to reduce the noise as far as possible, *i.e.*, by fixing the CPU frequency of the KVM virtual machine. To enable full scans on a moderate CPU utilization, we set the value of `/sys/kernel/mm/ksm/pages_to_scan` to 100 000. The `/sys/kernel/mm/ksm/sleep_milliseconds` remains at the default value of 200 ms. Furthermore, we set the CPU performance governor to performance using the `cpupower` tool to avoid noise from wake-up delays. We later on use the default configuration of Ubuntu, Windows, and the cloud provider to calculate the leakage rates for the cases studies.

Evaluation. For a single page, we measure a local timing difference directly in the virtual machine (KVM) and observe that the average local timing difference between a regular write memory access and a memory access causing a copy-on-write page fault is 7209.3 ns (n = 100, σ_{COW} = 26.23%, σ_{NOCOW} = 29%) using a local timer. We evaluate the timing difference in our local area network and on the internet using a simple HTTP server with a key-value store. Figure 3 illustrates the timing difference for a single page accessed with a copy-on-write page fault and a normal write access in a local area network. In the local area network, we observe a mean timing difference of 4353.91 ns (n = 1000). Figure 4 shows the timing difference for a single page accessed with a copy-on-write page fault and a normal write access from our Linux server on the internet (14 hops). While those two distributions overlap, they can be clearly distinguished in the mean respectively median values if enough samples are taken. In addition, the timing difference

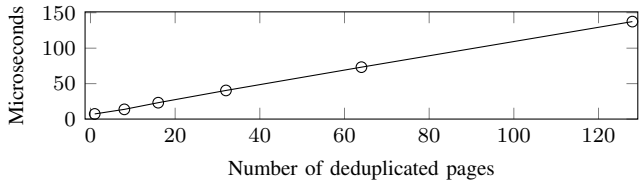


Fig. 5: Timing difference between amplified pages.

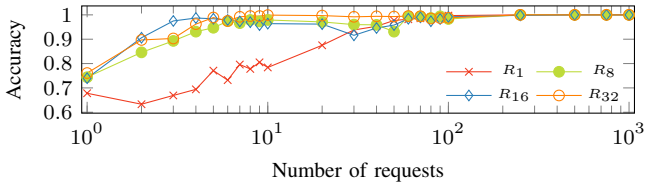


Fig. 6: Success rate of the classifier using the box test with a different number of deduplicated pages (R_x).

can be amplified by overwriting multiple copy-on-write pages in a single request.

Amplification. In the following paragraph we solve *C1: (Remotely amplify latencies for non-repeatable events.)*. A copy-on-write page fault can be amplified if multiple pages belonging to the same semantic entity (i.e., pages of an image file) get duplicated at the same time [16]. Therefore, we can amplify the timing difference between multiple deduplicated pages by sending a single request, writing to those which trigger the copy-on-write, and responding back. To evaluate the timing differences of multiple copy-on-write page faults, we evaluate a different set of pages, which triggers the page fault. We define a test set in our local KVM machine with a test set of 1, 8, 16, 32, 64, and 128 deduplicated pages and measure the average timing difference between triggering a copy-on-write page fault and a regular write access. We repeat the experiment 100 times and calculate the difference between the average times, which is plotted in Figure 5. We can see that there is a linear increase in terms of the timing difference with the increase of the number of deduplicated pages. For instance, with 8 pages, we get an average timing difference of 13 610.82 ns and with 16 pages, it is on average 22 946.14 ns.

Next, we evaluate the effect of amplification in our local area network setup with KVM. We use the term amplification factor to indicate the number of additional pages used to amplify the signal. We sample 1000 times and fit a CDF(cumulative distribution function) for each of the amplification factors $(1, 8, 16, 32)$ and randomly sample from the CDF. To discover the number of requests required to achieve an accuracy higher than 95% percent, we perform the box test by Crosby et al. [10]. Figure 6 illustrates the number of network requests required to achieve a certain success rate for a different number of pages deduplicated by the server. In this idealized setup, we observe that 10 requests with an amplification factor of 8 are enough to achieve a 95% confidence of distinguishing write accesses on a deduplicated page (incurring a page fault) and a non-deduplicated page in a local area network if amplification is used. The number of

requests required is in a similar range for the local area network observed by Van Goethem et al. [52].

C1

Remotely amplify latencies for non-repeatable events.

We showed the applicability of memory-deduplication attacks within the same security domain. We can amplify the timing differences for the copy-on-write page faults arbitrarily by leveraging the deduplication of multiple pages belonging to the same semantic entity. If the attacker is in control of overwriting the data, multiple copy-on-write page faults increase the latency.

R/W bit stays cleared. On Windows and Linux with page combining respectively kernel-same-page merging, we observe that when a page is deduplicated, and a write access occurs to one of the corresponding virtual pages, the remaining mappings of the same physical page remain marked as **copy-on-write**. We empirically validate this in an experiment, where we first map two pages A and B with identical content and wait for deduplication. We then write to page A and thus trigger a copy-on-write page fault. Subsequently, we analyze the R/W bit of the page-table entries for both pages and see that the R/W bit remained cleared for page B. This observation is especially useful when the attacker can align data, as was shown by Bosman et al. [6]. In Section VI-C, we exploit this behavior to amplify a single copy-on-write request via Memcached.

V. Remote Covert Channel

For our evaluation on both Windows and Linux, we first create a covert channel using our remote memory-deduplication channel. For this purpose, we implement a small HTTP/1.1 server in C++ to maximize the performance. We evaluate this attack on a local-area network with a hardware switch between the attacker and the victim.

Capacity. We build a covert channel to measure the performance of our remote memory-deduplication attack in a local area network scenario. The victim system for our transmission hosts a website that allows storing and updating files. The website keeps the files in in-memory storage, *i.e.*, in RAM.

The sender and receiver upload an identical large file to the website hosted on the victim system. Both use a $4\,\mathrm{kB}$ page in this large file to encode a '1'-bit. To transmit a '1'-bit, the sender puts the same page into RAM by updating the file via the website. The page is deduplicated with the page in the receiver's file. Conversely, to transmit a '0'-bit, the sender modifies the page in its file such that it is not deduplicated. The receiver sends a network request that either triggers a copy-on-write page fault or not. With measured round-trip time, the receiver distinguishes between a '1' and a '0'. The transmission can be parallelized in our setup by storing multiple bits at once and evaluating them in parallel.

Local Area Network. We transmit a random secret that is 8 bytes long, and repeat the experiment 100 times. On each repetition, we re-randomize a new $8\,\mathrm{B}$ secret. We observed that the Python capturing library has problems correctly parsing the packets when performing too many requests asynchronously on our webserver, we always leak 2 bytes (16 bit) in parallel for stable results. Between the send and receive process, a delay of $3\,\mathrm{s}$ was used to wait for deduplication.

In our Linux setup using amplification of 16, we achieve an overall performance of $302.16\,\mathrm{B/h}$ ($n = 100, \sigma = 5.81\%$), with an error rate of $0.6\,\%$.

Internet. We run the same experiment as for the local area network. On Linux, we used 20 requests per bit and used an amplification factor of 16 pages. On Windows, we used 20 requests per bit and an amplification factor of 32 pages.

On the Linux server, we achieve an overall performance of $34.41\,\mathrm{B/h}$ ($n = 100, \sigma = 5.87\%$) with an error rate of $0.83\,\%$. On the Windows server, we use constant triggering of memory deduplication and a delay of $50\,\mathrm{ms}$ and achieve an overall performance of $26.64\,\mathrm{B/h}$ ($n = 100, \sigma = 0.69\%$) with an error rate of $0.18\,\%$. We use this number to calculate the timing for the actual wait time on Windows of 15 minutes until the deduplication succeeded, which is $0.4\,\mathrm{B/h}$.

Using the same methodology as state-of-the-art work [4], [6], we simulate the covert's channel performance for the default configuration of 100 pages_to_scan on Linux. As it takes $596.9\,\mathrm{s}$ on the Equinix server to perform a full scan, the covert channel's performance shrinks down to $0.59\,\mathrm{B/h}$. For the provided numbers of the cloud provider, the covert channel would achieve $20.62\,\mathrm{B/h}$. These numbers are in a higher range as previous work, with the additional overhead of TCP, compared to the UDP sockets used in a similar attack scenario [48]. The other remote timing attacks did not provide concrete numbers on their covert channel [59], [21], [2], [47], [1], [52].

VI. Case Studies

In this section, we evaluate three case studies and demonstrate what types of attacks are possible with remote memory-deduplication. First, we demonstrate that we can exploit remote memory-deduplication in Memcached to fingerprint the system, including the operating system. We successfully detect the correct libc library over the internet in $166.51\,\mathrm{s}$. Second, we demonstrate a fully remote KASLR break by exploiting remote memory-deduplication within 4 minutes. Third and finally, we demonstrate how to leak database records byte-by-byte from InnoDB used in MySQL and MariaDB. In the following subsections we show how to solve C2, and C3.

A. Memcached

Memcached is a fast in-memory database offering a key-value store for applications [35]. The memory is managed using a slab allocator. A slab consists of a single or multiple memory pages, which are contiguous in physical memory. Memcached always allocates a $1\,\mathrm{MB}$ region and splits it into smaller chunks of equal size [35]. Chunks or objects with a similar object size get assigned to a certain slab class. For instance, if a slab class is $64\,\mathrm{B}$, the $1\,\mathrm{MB}$ page is split into $16\,384$ chunks. Newly inserted data is assigned to the smallest slab class that the data fits in [35]. This means a certain slab class contains objects of a certain size and assigns the objects to a chunk. A key-value pair is managed by the `item` structure, a linked list that contains the size of the key, the value of the

object, and some more metadata [35]. Each slab class has a free list, which is a linked list [35]. If an item gets freed, its former location is moved to the head of the free list.

Memory Management. The key-value pairs are stored contiguously in memory, which is ideal for triggering memory deduplication. We analyzed and profiled the source code of Memcached to check which functions are used and how the memory allocation works internally. In contrast to our expectations, Memcached does not perform an in-place replacement of the value to update. Even with the same key used in `memcached_set` and `memcached_replace` operations, a new location is assigned to the updated value. This new location is either an available free slab item from the head of the free list (do_slabs_alloc) or a new slab item.

After all input data from the new item is read, the old item is unlinked, and the new location linked for the item. The old location is freed and inserted to the head of the slab's free list (code path is from complete_nread → do_item_link). If a fixed memory size is reached, a least-recently-used (LRU) eviction policy is applied on a slab-base [11] This means that "old" items are replaced by more frequent items in a certain slab class.

1) Attack.: Our basic remote memory-deduplication attack on Memcached works as follows on Linux and Windows: First, the attacker places the targeted pages into the key-value store with a specific identifier. Then, the attacker waits some amount of time (delay) such that the pages are deduplicated. The deduplicated content can be for instance a static unique binary page of a specific version of the C standard library or other static binary pages in the system. After the delay, the attacker creates a new dummy item with the same key, which puts the deduplicated target page on the free list of Memcached. Then, the attacker updates the same item, which causes a copy-on-write page fault on the deduplicated page which is now overwritten.

Alignment. In general, it is not guaranteed that allocating memory with `malloc` internally uses `mmap` for a specific allocation size (this may depend on the libc variant, *i.e.*, glibc MMAP_THRESHOLD is 128 kB,system configuration, and operating system versions of the victim system). Thus, it is also not guaranteed that the allocated 1 MB region is aligned to any specific offset. Using `mmap` would ensure a page alignment, meaning the page offset would always be 0. However, in our experiments, we observed that malloc always used `mmap` internally for the 1 MB allocations on a default configured Ubuntu Linux installation.

It is also not guaranteed that the attacker inserts the first item in the slab class, which also causes an unknown alignment as also other chunks might be inserted on the 1 MB page. To overcome this limitation we propose a method to generate chunks of all different sizes possible for a slab class. We calculate all possible offsets the chunk could have on the page for a certain slab class. These possible offsets can be computed for each possible chunk per page i as `offset`:

$$(malloc\_offset + i \cdot chunk\_size + item\_header\_size + key\_size) \mod 4096$$

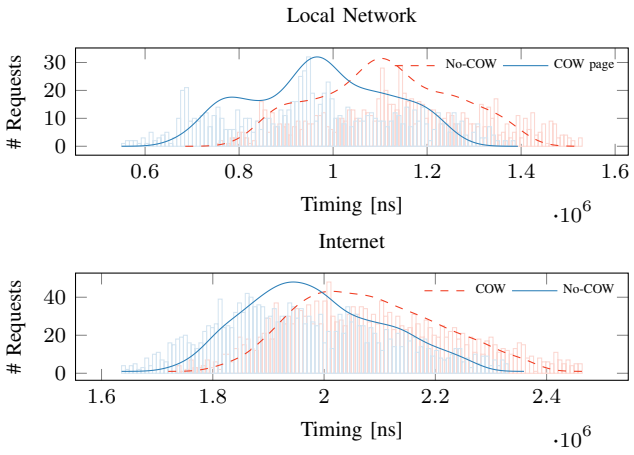


Fig. 7: Histogram of the network requests in a local area network and in the internet setup using 16 pages to amplify the results in Memcached.

where the `key_size` is attacker-controlled, the `chunk_size` depends on the slab class, `malloc_offset` = 16 and the size of the item header is defined as `item_header_size` = 56. Hence, to overcome the alignment issue, we use the same page with the different offsets to cover all possible alignments, which is guaranteed to include the correct alignment required for deduplication.

LRU. In a real-world application, Memcached can be expected to be heavily used by other users as well. However, we still need to keep the data inside the data store. We achieve this by frequently accessing the data using GET requests on the service to avoid being evicted by the LRU eviction strategy. Note that this does not trigger copy-on-write page faults as we only read the data but do not modify it. We discuss the eviction in more details in an attacker scenario in Appendix A.

Evaluation. We evaluate our attack on Memcached 1.6.8 and connect to the Memcached service using UNIX sockets. We evaluate this scenario using a PHP site (version 7.4.3), which is hosted on an Nginx server (version 1.18). Our evaluation uses the local area network setup, and we also run on a Linux server on the internet 14 hops away. Our victim server and attacker setup are the same as described in Section V.

We alternate between pages that do not trigger a copy-on-write page fault and pages that trigger a copy-on-write page fault due to deduplication. In addition, we alternate the order to avoid a potential bias, which could be introduced by a fixed request order. In total, we perform 1000 HTTP requests. Figure 7 shows the timing differences we observe in this setup. We can see that it is easy to distinguish between deduplicated pages and non-deduplicated pages.

Libc Fingerprinting. Operating system and library fingerprinting is a good starting point for penetration testing to determine potential vulnerabilities on the identified operating system or the running applications. Those results observed from Memcached can be used to perform fingerprinting of operating systems by looking at fixed memory pages as was proposed by Owens et al. [41]. We use the same setup as before and try to fingerprint the exact standard C lib (libc)

version. In our experiment, we probe 3 different versions of the libc. We perform 20 subrequests for each version we probe on Memcached. Our information disclosure attack detects the correct version in one sample within 44.28 seconds (n = 100,σ = 0.19%) and an accuracy of 90%, depending on which library is mapped on the victim. Memcached can be used as an additional possibility to force deduplication and evaluate the response time, which we show in Section VI-C. The timing differences for the correct library guesses in PHP via Memcached can be seen in Figure 14 (Appendix A).

Internet. We run the same experiment with the same setup (Nginx, PHP, Memcached) over the internet targeting the Equinix Linux VM `14` hops away. We detect the correct version in one sample within 166.51 seconds (n = 100,σ = 9.67%) and an accuracy of 90%. Using the default settings for KSM, the attack would take 3.36 h. With the settings provided by the cloud provider, the attack would take 0.22 h.

C2

Trigger and observe copy-on-write page faults in a victim domain that shares no memory with any attacker domain.

With our attack on PHP-Memcached hosted on an Nginx server, we demonstrated that it is possible to trigger copy-on-write page faults within the same security domain without relying on shared memory with the attacker domain. This can be used to perform operating system fingerprinting like was shown via Memcached over the internet.

B. Breaking KASLR Remotely

By randomizing the location of kernel code, data, and drivers at every boot, KASLR makes the exploitation of memory corruption bugs in the kernel much harder (Section II-D) as an adversary needs to guess the addresses for the attack correctly. In the past, different side-channel attacks allowed to reduce the entropy of the randomization or to break it entirely [18], [20], [13], [29], [28], [9], [8], [17], [25], [15], [14].

While Klein and Pinkas [23] used an information leak in IP headers to break KASLR, so far, no remote side-channel attack has been demonstrated against KASLR. In this section, we exploit memory deduplication to break KASLR of a virtual machine remotely. Concurrent work by Kim et al. [22] demonstrated a KASLR break on co-located machines on VMWare ESXi break via memory deduplication within 12 minutes.

We describe the necessary building blocks and threat model to mount the attack targeting one virtual machine over the network.

1) Attack Scenario & Attacker Model: We assume that the version of the operating system running on the victim machine is known to the attacker. That memory deduplication is active and enabled by the operating system (or hypervisor). This information can be obtained by an information leak or a fingerprinting attack, similar to the one described on Memcached in Section VI-A.

2) Attack & Building Blocks: Finding the content of memory pages that are identical to the ones used by the victim operating system forms the basis of our KASLR break. If the content of the attacker-controlled page is identical, the hypervisor deduplicates it. Thus, a subsequent write to the page yields a higher execution time forming the side-channel we exploit throughout this paper. While a page with the same content as a kernel page allows fingerprinting the operating system, data and pointers stored on the page either change during runtime or are randomized on every boot and are, thus, less predictable.

However, on Linux, the text segment is mapped between the 1 GB region of `0xffff ffff 8000 0000` and `0xffff ffff c000 0000`. As the kernel is 2 MB aligned, there are only 512 possible offsets in this region where the kernel can be placed. If we find kernel pages that only contain kernel addresses and static values, *i.e.*, data that is not modified during runtime, we can generate 512 different versions of the page. Each version corresponds to one possible offset and contains the kernel addresses if the kernel would be mapped to said offset.

A page on the victim machine is now filled with a possible content candidate. The remote attacker uses the API provided by the victim machine to set the content of a page. Depending on the configuration of the hypervisor on the target machine, the adversary waits until pages should be deduplicated. Now the adversary writes to the same page using the API. The adversary measures the time it takes to write to the page, *i.e.*, the time it takes for the network request to be handled. If the content set by the adversary matches the targeted kernel page, the hypervisor has deduplicated the pages, and to handle the write. They have to be duplicated again. Thus, if the content matched, the adversary observes a higher timing. For all of the 512 different possibilities, the adversary performs these measurements, yielding a single candidate that corresponds to the currently used randomization offset. To deal with measurement noise, the adversary has to repeat these measurements.

In addition, it is possible to amplify the side-channel leakage. Instead of a single kernel page, multiple different kernel pages can be generated based on the assumed kernel offset and set at the same time. Thus, instead of a single deduplication, the adversary observes multiple ones within a single measurement.

Finding Suitable Kernel Pages. To send the content of possible kernel pages, the adversary first needs to scan possible page candidates. This can be done upfront in an offline phase and used for kernels of the same version, thus, one assumption is that the adversary knows the version used by the victim.

To find possible page candidates, we walk the page table levels of the Linux kernel and inspect the content of each mapped 4 kB page. We know in which region the text segment can be mapped and check each possible position of a pointer, *i.e.*, each 64 bit, if it lies in this region. If so, we dump the contents of the page as well as all the offsets representing a pointer within the page. We also extend this approach to kernel pages belonging to kernel modules, as they are also randomized in a certain memory region and could be used to break the randomization of the modules. On a machine running

Linux 5.4.92, we find 15 737 pages where 4070 contain values matching pointers within these memory regions.

In a second step, we filter the dumped pages for possible candidates that we can use for the attack. We try to find corresponding symbol names to the detected addresses by matching them to `/proc/kallsyms`, yielding 15 pages that only contain resolvable kernel text addresses. 3973 pages contained module addresses, 39 resolvable and unresolvable addresses, and 43 no symbols at all. These pages now need to be checked if their content is static and, thus, does not change over time. This can be achieved by dumping the content periodically and checking it for modifications. Further, we want the pages to not contain any data initialized during boot time and, thus, we need to check if the content of those pages changes (excluding the kernel addresses) while rebooting the system multiple times. In order to rule out hardware-specific data, this should be done on different physical machines.

3) Remote Attack: For our remote KASLR break, we implemented the victim server in two ways. First, as a RESTful API listening for HTTP/1 requests implemented in C++ using the pistache framework [43]. For simplicity reasons, the API allows the adversary to set and modify the content of pages directly. However, as we have shown in Section VI-A, the data could be stored in an in-memory database as well. Second, we elevate the service for HTTP/2 to support multiplexing, allowing us to mount timeless timing attacks described by Van Goethem et al. [52]. In both scenarios, an Nginx [39] web server running on the target machine forwards the request to the victim service. The attacker sends the crafted pages for the offset to test to the victim using the API. After 2 s, *i.e.*, the time the page would be deduplicated on our system with a high chance, the attacker sends a network request modifying and, thus, causing the probable duplication, and measures its response time.

HTTP/1. In the first scenario, we use HTTP/1 to communicate with the network service and measure the response time of the network requests. Figure 9 illustrates the distribution of response times for the correct offset and an incorrect offset. Figure 8 shows the mean response time of a network request for a specific offset in a remote-attack scenario. After sending 100 requests, we can clearly see the increased response time for the currently used randomized kernel offset.

In the local setting, we were able to recover the correct randomization offset with a success rate of 100 % and an average runtime of 21.3 s (n = 100). In the remote setting, we were able to recover the correct randomization offset with a success rate of 73 % and an average runtime of 5 min 57.9 s (n = 100). With the default configuration of the cloud provider (cf. Section IV-A) this would yield an average simulated runtime of 34 min 28.69 s. With the default Linux settings, it would take 9 hours and 2 minutes.

HTTP/2 Multiplexing. To improve on the measurement noise introduced by the connection between the victim and the adversary and the necessity of accurate time stamps, we utilize Timeless Timing attacks [52] to overcome this issue. HTTP/2 allows to pack multiple requests within a single packet and, thus, the requests reach the server at the same time. However, the response of the request that reaches the sender faster has likely been processed quicker.

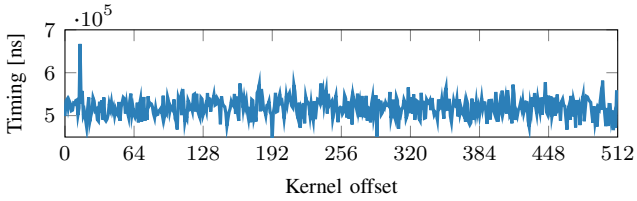


Fig. 8: Execution time to a page containing the content adjusted to one of the possible kernel offsets. The high peak at offset 14 yielded the same content as the kernel page of the VM and, thus, has been deduplicated by the hypervisor.

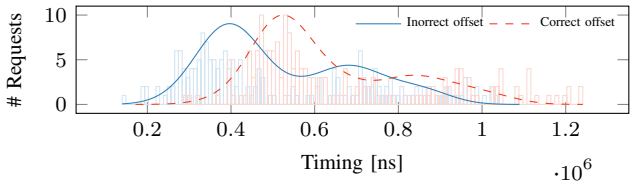


Fig. 9: Histogram of the measured access times for an incorrect and the correct offset for the KASLR break. A correct guess can be clearly distinguished from an incorrect guess.

In contrast to the sequential HTTP/1 attack, we pick pairs of kernel offsets that we send to the server using multiplexed HTTP/2 requests. For every attempt, we send each pair to the server and record for which request we receive the response first. Note that we do not need to rely on measured access times but just on the response order of the requests. For pairs of both incorrect kernel offsets, we should observe a uniform distribution between the offsets. However, if one of the offsets is the correct one, we should observe an unequal distribution. To optimize the approach, we reduce the number of candidates and filter out pairs with a uniform distribution early. With each filter step, we re-combine the candidates to new pairs.

To amplify the signal, we crafted 7 kernel pages for each possible kernel offset. In the local-network setting, we achieved a success rate of 88.89 % with an average runtime of 1 minute and 38 seconds (n = 100). The raw timing differences observed in the HTTP/1 setting enable a faster attack than HTTP/2. We were able to successfully find the correct offset in the remote setting with a success rate of 92 % with an average runtime of 3 minutes and 15 seconds (n = 100). With a prolonged waiting time using the default configurations of the cloud provider of how many pages are scanned by the operating system per minute, this would yield an average simulated attack time of 18 minutes and 25 seconds. With the default Linux settings, it would take 4 hours and 48 minutes.

C. InnoDB Record Data Leakage

InnoDB is a storage engine used by default in the database management systems MySQL and MariaDB. The storage engine efficiently buffers record data and index caches in the memory and is used instead of using the operating system's page cache directly. InnoDB has the advantage of providing faster access to frequently used data.

Database systems use indices to allow quick access to records, *i.e.*, normally, an index is placed automatically on columns marked as primary key. InnoDB implements indices using a B+ tree, which allows fast record lookups. The nodes of the tree are represented by index pages, which are the basic storage unit of InnoDB and have a size of 16 kB by default. The leaf index pages contain the actual user data. The non-leaf ones link to other leaf or non-leaf pages. Index pages on the same tree level are linked together to allow scanning operations. User records in an index page are logically linked in ascending order by their key but may be placed anywhere in the page's physical memory.

High-Level Overview of the Attack. To achieve byte-by-byte leakage, the attacker needs to control the content and size of data that is stored before the target data to bytewise shift the target data onto the attacker-controlled page. Bosman et al. [6] showed that byte-by-byte leakage is possible.

InnoDB performs a data reorganization of data if an insert or update query fails as an optimization. This optimization enables byte-by-byte leakage if the attacker controls most of the InnoDB record. Using this primitive to perform memory massaging, an attacker can shift the secret.

Assumptions. We assume that Memcached can be used in addition as a leakage primitive to leak the secret data co-located to the attacker-controlled data bytewise. As we will later analyze, the Linux page cache caches Note that this can be any primitive used for triggering deduplication and copy-on-write page faults, *i.e.*, nginx, as was shown by Bosman et al. [6]. We assume a database application with a user table which is defined in Figure 15 and that the InnoDB index page has a certain layout, which is explained in more detail in Section VI-C3. We assume that the attacker can perform multiple tries in parallel until such a layout is given. If the layout is given, the attacker can verify whether the requirements are fulfilled.

Attack steps. Figure 10 illustrates the five steps of the InnoDB reorganization attack. In the first round, the attacker triggers the reorganization and shifts the first byte of the secret value (**"SECRET"**) onto the controlled 4 kB-page. Next, the attacker stores multiple guesses into Memcached. The attacker waits until the deduplication happened. After the deduplication happened, the attacker updates the Memcached guess pages and measures the round-trip time of the network packets. The right guess should lead to a significantly higher timing than the other guesses with enough samples taken. Afterwards, the attacker repeats the procedure to shift the second byte into the attacker-controlled InnoDB page, updates the guesses in Memcached, including the first recovered byte, and leaks the second byte. This procedure can be repeated up to a certain leakage size. The limits are discussed in Appendix A.

Why an Additional Leakage Primitive is Required. InnoDB tries to circumvent the page cache of the Linux kernel by using the `O_DIRECT` flag in `mmap` [38]. However, the data is still in the page cache and gets deduplicated. The page-cached data cannot be overwritten directly via InnoDB. Therefore, we cannot use a second InnoDB record to trigger a copy-on-write page fault since the data would also get deduplicated. We found no convenient and reliable way to get external blobs consistently in the memory and replace them to trigger copy-on-write page faults in InnoDB. For external blobs, we have a similar race as in Memcached, since updates are not performed in-place. Instead, resource releasing is performed in a similar way compared to Memcached. Consequently, for our attack, we use a memory-resident second channel (Memcached) to trigger the copy-on-write page fault. However, this could, in general, be any web application/resource providing such a leakage primitive that is running on the same machine.

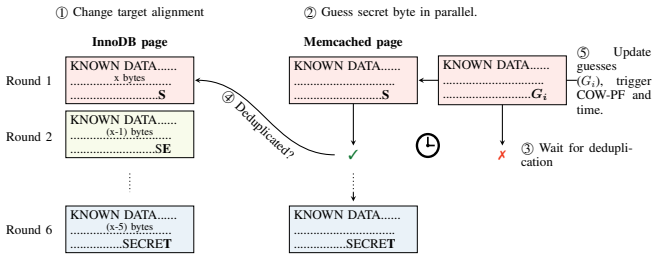


Fig. 10: High-level idea of the InnoDB Reorganization attack.

1) Determining InnoDB Attack Requirements.: We analyze the memory ordering of InnoDB by performing different SQL statements:

Insert. Upon inserting a new record, InnoDB first tries to place the record in its corresponding index page. If no such page exists, a new one is created.

In case of an existing index page with sufficient consecutive free space, e.g., unused space at the end of the page or a gap from previous deletes, the record is placed on this index page. Should this not be possible, e.g., the page is full, or the free space is too fragmented, either the current index page is defragmented (*reorganized*), a new page is allocated, or a page *split* is performed. Inserting into a new index page is only possible if it does not break the existing relations in the index tree. Otherwise, a page split has to be performed. As the space of previously deleted records is reused, the physical order of records in an index page does not always reflect their logical order, e.g., a record with key 5 might be inserted in memory before a record with key 2.

Delete. When a record is deleted, it is added to the index pages free record list. Should the free space resulting from deletions reach a certain merge threshold, InnoDB tries to perform a `merge` operation to save space. A merge operation is possible if the utilization of the next or previous linked index page is low enough to combine it with the current page [30].

Update. Update queries in InnoDB update a record in-place, as long as the updated record fits in the same size as the old one (`new_record_size` ≤ `old_record_size`) [31]. Otherwise, the update operation is realized as a delete with a subsequent insert operation, inserting the updated record.

Reorganization. An insert or update query can fail even if enough space is available on the index page because the free space is fragmented. In such a case, InnoDB performs an optimization called `reorganization` [32]. During reorganization, the page is rebuilt by clearing its contents and inserting existing records in their logical order. Afterwards, the pending insert or update operation is completed using the freed space at the end of the page.

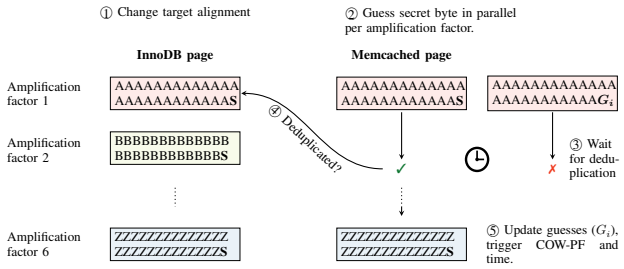


Fig. 11: Leakage of a secret byte (**S**) from an InnoDB record using Memcached with amplification.

2) Reorganization Attack.: As shown by Bosman et al. [6], an attacker can use memory-deduplication attacks to leak data byte-by-byte if the attacker can change the memory layout on a byte granularity. We demonstrate that this approach can also be applied in a fully remote attack scenario.

We leak data byte-by-byte by exploiting the `reorganization` of database records in InnoDB index pages. The reorganization is triggered if data is updated or inserted, and reorganization keeps the data on the same index page. Figure 15 in Appendix A shows the user table and its fields in the database, including an `id`, `username`, `password`, and an `image` field. We assume that the attacker can register an arbitrary number of users and modify their content.

Alignment Changing. To leak attacker-unknown record data, we need a large record r_{AT} to `shift` bytes from a target record r_T into an attacker-controlled region. To trigger the `reorganization`, we require an additional record r_{AX} in the user table. The reorganization orders the records in RAM in their logical order. With targeted size modifications of the attacker-controlled records r_{AT} and r_{AX}, we can trigger the reorganization and bytewise shift record data from r_T into an attacker-controlled $4\,\mathrm{kB}$ region. To leak the targeted byte, we use Memcached used in a simple HTTP web server as a second channel, same as in Section VI-A.

Amplification. To use amplification, we fill multiple pages on both Memcached and InnoBD with different fill bytes but with a constant leaked offset, as shown in Figure 11. As already mentioned, the copy-on-write bit stays set for the correct guess in Memcached. Therefore we can amplify by updating both Memcached and InnoDB fill bytes and waiting again for the deduplication. This procedure can be repeated up to a certain amplification factor. To trigger copy-on-write pagefaults, we send an HTTP request to the server, which overwrites the content of the Memcached pages. To reset to an index page layout, which allows leaking a different byte offset, it is required to trigger another reorganization by modifying the sizes of the records r_{AX} and r_{AT}. All requirements are explained in full detail in Section VI-C3.

3) In Detail Analysis of Attack Requirements.: To perform the reorganization attack, the attacker and victim have to be placed on the **same** InnoDB index page. While this is a question on the workload of the system, we assume that an attacker can perform sufficient repetitions to generate a layout leading to data leakage.

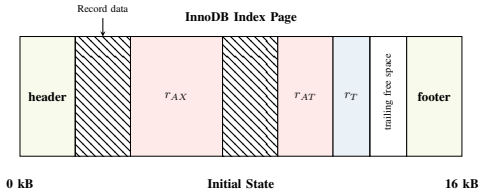
A `reorganization` can be caused by updating the size of a record so that it does initially not fit in any available consecutive free space on the page but does fit after defragmentation. By choosing record sizes in the right way, it can be guaranteed that such reorganizations are always possible.

Initial Page Layout. Figure 12a describes the initial page layout required by InnoDB to leak record data. We exploit InnoDB's reorganization feature as a primitive for the attacker to align the secret on a byte granularity. At the beginning of an index page, there are a couple of headers and system records, summing up to $120\,\mathrm{B}$ [33]. Then the data of the user records follows. At the end of the page, there is a so-called page directory and further meta-data. The user records are also preceded by a dynamic-sized header, which depends on the table layout and contains information necessary for using and organizing the records. Figure 12a shows the assumed initial physical and logical layout for our InnoDB attack. The hatched areas represent unknown records. r_{AT} and r_{AX} are attacker-controlled records and r_T is the target record to leak.

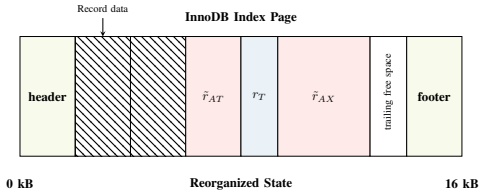
Analysis of Required Sizes for Exploiting Reorganization. During the rebuilding of index pages, records are inserted consecutively in memory by their logical order, except for the record that triggered the reorganization, which is inserted last, regardless of its key. In total, it is possible to insert $16\,252\,\mathrm{B}$ (`max_free_space`) of record data into an index page. The layout requires that the record r_{AT} is logically located before r_T. r_{AX} is required to be physically before the two records, somewhere in between the two hatched regions in Figure 12a. The record r_{AT} is used to change and control the target record's alignment r_T. The attacker wants to make r_{AT} as large as possible to change the target record's r_T alignment. In the default setting of InnoDB with a default index page size of $16\,\mathrm{kB}$, the maximum size for a record is $8125\,\mathrm{B}$ [33]. Thus, to leak as much data as possible, we choose r_{AT} to be 8125. The validation of all requirements and the potential leakage rate is described in Appendix A. Next, we discuss the attack steps in more detail.

Preparing the Alignment and Triggering the Reorganization. To trigger a reorganization, the attacker increases the size of record r_{AX} using an update query. The reorganization only happens if the updated size still fits into the total free size of the index page. The new reorganization moves r_{AX} to the `trailing free space` within the index page, which is large enough to contain at least $r_{AX} + 1$.

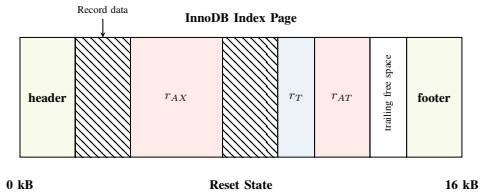
If the attacker wants to shift a byte of the target record by δ bytes such that the byte moves closer to r_{AT}, the attacker can update the size of r_{AT} and decrease it by δ and increase the size of r_{AX} by δ. The reorganization takes out the record r_{AX} and moves it to the newly created free location. It causes the record $\tilde{r}_{AX}$ to be moved after $\tilde{r}_{AT}$ and r_T. r_{AX} can only be modified up to the maximum record size. While the header of the user record has a dynamic size, we assume that the record does not change during the attack. If there is an additional record after r_T, the header stays the same. If there is no record after r_T, the record header points to the Supremum [34], which is at the beginning of the page. With each alignment change, the next offset field in the records header needs to be incremented by the byte offset to leak.



(a) InnoDB page layout, which is susceptible to a reorganization attack. A simple model of an index page consists of a fixed-size header, user records, the unused space at the page end, and a footer. In this scenario the attacker controls the records r_{AX} and r_{AT}. r_{AT} is used to control the alignment of target record to leak r_T. r_{AX} is used to trigger the reorganization within an InnoDB index page.



(b) If the reorganization was triggered $\tilde{r}_{AX}$ is moved to the beginning of the trailing free space.



(c) Reset and reorganized InnoDB record.

Fig. 12: InnoDB reorganization steps.

Leaking the Secret Byte. Since we cannot leak the records in MariaDB alone on Linux, a second way to trigger the memory deduplication is required. Furthermore, we apply amplification for our bytewise leakage, as discussed in Section V. Figure 11 illustrates the amplified version of the InnoDB record attack. We use different pages with the same content but only a different last byte in the page, which is our probe byte in Memcached (in Figure 11 the secret byte is x). In MariaDB, we update our record r_{AT} with the content of the first page (`AAA....S`). Afterwards, we wait for a certain delay until the memory deduplication is triggered. If the secret byte is correct, the page gets deduplicated. After the delay, we modify r_{AT} again. The page in Memcached still has the R/W bit cleared. If all amplification pages are deduplicated, we use the web application to write on each of our amplification pages and measure the response time.

Reset. After the reorganization, the alignment is changed, and we get a record layout, as illustrated in Figure 12b. Unfortunately, we cannot modify the base alignment since we do not know the size of the other records on the index page. However, we can either try to repeat the attack until we start at the beginning of a $4\,\mathrm{kB}$ page or leak the base alignment. To reset the state back to the initial one, we again exploit reorganization, changing to the previous sizes. The requirement to trigger another reorganization via r_{AT} is that the trailing free space is smaller than the reset size of r_{AT}. The reorganization causes that the updated record r_{AT} is now moved after r_T, leading to the memory layout illustrated in Figure 12c. However, this is no problem since we can force another reorganization, bringing back our reorganized state, as illustrated in Figure 12b. After each alignment change, we switch between the reset and reorganized state and never return to the initial state.

Evaluation. We implement and evaluate our attack on MariaDB version 10.5.8, using UNIX sockets and a simple HTTP server to connect to it and to Memcached 1.6.8. The database is setup as shown in Figure 12a. We choose a random secret of $4\,\mathrm{B}$ and repeat our data leakage experiment 20 times. We apply the amplification technique shown to leak a single byte via 8 pages. To be on the safe side, we send 40 requests for all 256 possibilities. Afterwards, we probe all 256 possibilities for the secret byte at once via Memcached. We look at the timing difference between the means of the received distribution of writing to copy-on-write and non-copy-on-write pages. Our attack automatically detects if a byte was accidentally classified as copy-on-write in case we do not get clear results for the following byte to leak. In this case, we can backtrack to the last byte that was correctly guessed. Therefore, our approach is self-correcting in case we accidentally received a wrong byte, and, thus, our approach is nearly complete error-free, despite the last byte where an error might occur.

On Linux, we observed that the time to wait for the deduplication on InnoDB is, in many cases, more than twice as big as in the previous cases. To be on the safe side, we increased the wait time to 4 seconds. As the amplification needs to be triggered sequentially, this leads to a wait time of 32 seconds per guess round. This longer delay is required since the target page is constantly changed, and KSM does not immediately deduplicate pages which are often modified [45]. The runtime of the attack to leak four random bytes is on average 5644.20 seconds ($n = 100, \sigma = 0.54\%$). Thus, the attack leaks on average a single byte in 39.07 minutes or about $1.5\,\mathrm{B/h}$ from a virtual machine running on a remote server in the local area network. We simulate the attack's performanc using the default configuration to $0.018\,\mathrm{B/h}$. With the provided configuration of the cloud provider, we got a simulated time of $0.07\,\mathrm{B/h}$. Note that the large bottleneck of this attack is the amplification technique *i.e.*, for one iteration $32\,\mathrm{s}$ have to be waited.

Limitations. For the initial setup, cf. Figure 12a, the uncontrolled record data before r_{AT} can be modified in-place as long as the overall size is not changed. Every in-place update of other records does not influence the attack. However, a memory split, merge, or reorganization would interfere with the attack and potentially destroy the needed layout.

C3

Find remote request paths that do not only keep attacker-controlled data in memory but also provide the attacker with control over alignment and in-memory representation.

We demonstrated a scenario for InnoDB used in MariaDB and MySQL, which allows changing the alignment of database records remotely. By changing the sizes of two attacker-controlled records, an attacker can load bytewise parts of victim's data to an attacker-controlled 4 kB page. Amplification can be achieved by leveraging the fact that deduplication can be triggered multiple times by modifying the attacker-controlled record and adding certain amplification pages to Memcached (like shown in Figure 11.)

VII. Mitigations and Further Attack Targets.

A. Mitigations

Our attack showed that memory deduplication is still a threat and even exploitable over the network. Even isolation into security domains like performed on Windows is not enough to mitigate information disclosure via memory deduplication.

Deactivation. While the simplest solution would be to altogether disable memory deduplication on Windows and Linux (Ubuntu), it is probably the most costly in terms of performance overhead. Especially on Windows server, where multiple users would use the same application, this could lead to immense memory overhead. Windows allows disabling of memory deduplication per process [37].

Only Deduplicate Zero Pages. Another mitigation by Bosman et al. [6] would be only to deduplicate zero pages. According to their evaluation, between 84% and 94% of the deduplication in Microsoft Edge are only zero pages [6]. However, the covert channel is still possible with this solution since we can still trigger copy-on-write page faults on deduplicated zero pages.

TPS. VMWare TPS [54] uses additional salts to enable memory deduplication. The salt value and the content of page have to be identical to be shared. If VMs want to deduplicate shared content, the salted value is unknown to an attacker. While this approach protects against cross-VM attacks, TPS does not protect against remote memory-deduplication attacks in the same domain.

CovertInspector. Wang et al. [55] demonstrated an approach to detect memory-deduplication attacks by modifying KVM by 300 lines of code. Their approach has a particular focus on intercepting the `rdtsc` instruction triggered by the VM and also the number of pagefaults. Remote timers are not considered by CovertInspector.

VUsion. VUsion [40] mitigates all kinds of memory-deduplication attacks by applying a share-XOR-fetch policy and fake merging. All pages that are considered for deduplication behave the same in terms of access times and copy-on-write pagefaults. Fake merging guarantees that every access on a page, both shared or non-shared, behaves the same in terms of access time. This mechanism prevents attacks on the detection of pages being actually deduplicated [40]. While fake merging would mitigate all of our attacks based on the copy-on-write page fault, it is not implemented nor intended to be merged in the Linux kernel.

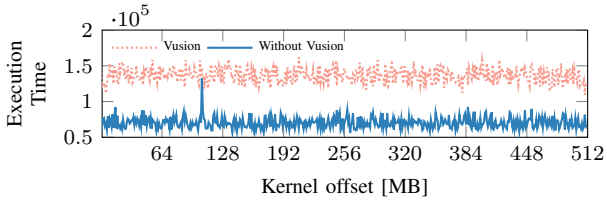


Fig. 13: Mean response time for all possible kernel offsets. While an adversary can easily observe the correct offset 106 on an unprotected system (blue), the VUsion-protected system (red) prevents the leakage.

We experimentally verified the effectiveness of VUsion against our remote memory-deduplication attacks in a local area network setting. Figure 13 illustrates the KASLR break (cf. Section VI-B) on a protected (red) and an unprotected Linux kernel 4.10 (blue) running Ubuntu 17.04 LTS. We measured the response time for every possible offset 100 times and reported the mean value. One can clearly see that our attack successfully recovers the correct offset 106 while the attack against the VUsion-protected kernel only observes higher timings.

Network-layer Countermeasures. On the network layer, we can mitigate remote memory-deduplication attacks via network packet inspection tools and DDoS monitoring. Another possibility to mitigate remote memory-deduplication attacks is by adding additional noise to the network, *i.e.*, by performing load balancing or adding discrete time delays. This would require more samples for the attacker, and at a certain point, it could make the attack infeasible.

B. Alternative Attack Targets

We want to emphasize that fixing Memcached does not mitigate the problem of remote memory-deduplication attacks as our techniques are generic and can be applied to other applications as well. In addition to Memcached and InnoDB, we analyzed further applications which could be susceptible to remote memory-deduplication attacks. Many web applications offer the possibility to use Memcached, such as PHPBB, WordPress, Moodle, and PrestaShop. Moodle allows image caching, which might be already used to perform the fingerprinting attack. We analyzed the in-memory DB Redis and found that 4 kB pages can be also placed into the memory. There is again meta-data stored about the stored item, and it is again a question of the correct alignment for the attacker to perform remote memory-deduplication attacks. If the attacker's guess about the alignment is correct, copy-on-write pagefaults can be triggered in a similar manner to Memcached by freeing an item and again inserting a new one with the equal size. This leads to an overwrite of the deduplicated memory. Furthermore, we analyzed the other popular alternative for in-memory databases SQLite. However, we found that we could not fully place a single 4 kB page into memory. We also checked Aerospike

and observed that memory is in DRAM as key-value pair and that the `aerospike_key_put` function directly replaces the content and could be used to trigger copy-on-write pagefaults. As already shown by Bosman et al. [6] also request pools like used in nginx are susceptible to memory deduplication attacks.

VIII. Conclusion

In this work, we presented how memory deduplication can be exploited from a remote perspective. This attack does neither require local code execution nor JavaScript execution in the browser, as demonstrated in previous work. With targeted web requests, we can observe timing differences between duplicated pages over the network. We first evaluated the speed of our remote covert channel based on an HTTP web server achieving a performance of up to $302.16\,\mathrm{B/h}$ in a LAN setting and $34.41\,\mathrm{B/h}$ over the internet. Further, we fingerprinted libraries used on the system by exploiting the Memcached database. It is possible to fingerprint libraries within $166.51\,\mathrm{s}$ over the internet. Within only 4 minutes, we successfully broke KASLR from a virtual machine running on a server 14 network hops away. Even though there are potential mitigations against memory deduplication within the same security domain, they are not applied in Linux systems. Finally, we leaked the database records' content from InnoDB with $1.5\,\mathrm{B/h}$.

Acknowledgments

We would like to thank our anonymous reviewers for valueable feedback and comments on the paper. Furthermore, we want to thank Tom Van Goethem for feedback on the draft and support on the HTTP/2 experiments. We want to thank Equinix Metal for providing us bare metal servers. This work was supported by generous funding and gifts from the EU project SOPHIA, Red Hat and AWS. Any opinions or recommendationsw expressed in this work are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Acıiçmez, Onur and Schindler, Werner and Koc, Cetin K. Cache based remote timing attack on the aes. In *CT-RSA*, 2006.

[2] Hassan Aly and Mohammed ElGayyar. Attacking aes using bernstein's attack on modern processors. In *International Conference on Cryptology in Africa*, 2013.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009.

[4] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. CAIN: silently breaking ASLR in the cloud. In *WOOT*, 2015.

[5] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[6] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*, 2016.

[7] David Brumley and Dan Boneh. Remote Timing Attacks Are Practical. In *USENIX Security*, 2003.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[9] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[10] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):17, 2009.

[11] Skyscanner Engineering. Journey to the centre of memcached, 2020. URL: https://medium.com/@SkyscannerEng/journey-to-the-centre-of-memcached-b239076e678a.

[12] Equinix, 2021. URL: https://www.equinix.com.

[13] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[14] Enes Göktaş, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative Probing: Hacking Blind in the Spectre Era. In *CCS*, 2020.

[15] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*, 2017.

[16] Daniel Gruss, David Bidner, and Stefan Mangard. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS*, 2015.

[17] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[18] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

[19] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In *AsiaCCS*, 2015.

[20] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[21] Darshana Jayasinghe, Jayani Fernando, Ranil Herath, and Roshan Ragel. Remote cache timing attack on advanced encryption standard and countermeasures. In *ICIAFs*, 2010.

[22] Taehun Kim, Taehyun Kim, and Youngjoo Shin. Breaking kaslr using memory deduplication in virtualized environments. *Electronics*, 2021. URL: https://www.mdpi.com/2079-9292/10/17/2174.

[23] Amit Klein and Benny Pinkas. From IP ID to device ID and KASLR bypass. In *USENIX Security*, 2019.

[24] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[25] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the Isolated Kernel Address Space Using Tagged TLBs. In *EuroS&P*, 2020.

[26] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *S&P*, May 2020.

[27] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. 2018.

[28] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*, 2020.

[29] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *S&P*, 2021.

[30] Percona Marco Tusa. Innodb page merging and page splitting, 2017. URL: https://www.percona.com/blog/2017/04/10/innodb-page-merging-and-page-splitting/.

[31] MariaDB, 2020. URL: https://github.com/MariaDB/server/blob/09a1f0075a8d5752dd7b2940a20d86a040af1741/storage/innobase/row/row0upd.cc.

[32] MariaDB, 2020. URL: https://github.com/MariaDB/server/blob/09a1f0075a8d5752dd7b2940a20d86a040af1741/storage/innobase/btr/btr0cur.cc.

[33] MariaDB, 2020. URL: https://github.com/MariaDB/server/blob/09a1f0075a8d5752dd7b2940a20d86a040af1741/storage/innobase/include/page0page.ic.

[34] MariaDB, 2020. URL: https://dev.mysql.com/doc/internals/en/innodb-infimum-and-supremum-records.html.

[35] Memcached, 2020. URL: https://memcached.org/.

[36] Microsoft. Azure serverless computing, 2019. URL: https://azure.microsoft.com/en-us/overview/serverless-computing/.

[37] Microsoft, 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_side_channel_isolation_policy.

[38] MySQL, 2017. URL: https://blog.toadworld.com/2017/10/19/data-flushing-mechanisms-in-innodb.

[39] nginx, 2021. URL: https://www.nginx.com/.

[40] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Secure Page Fusion with VUsion. In *SOSP*, 2017.

[41] Rodney Owens and Weichao Wang. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *International Performance Computing and Communications Conference*, 2011.

[42] Gerald Palfinger, Bernd Prünster, and Dominik Ziegler. Prying cow: Inferring secrets across virtual machine boundaries. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, 2019.

[43] Pistache, 2020. URL: http://pistache.io/.

[44] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium*, 2016.

[45] Red Hat. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*, 2017.

[46] Redis, 2013. URL: https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached.

[47] Vishal Saraswat, Daniel Feldman, Denis Foo Kune, and Satyajit Das. Remote cache-timing attacks against aes. In *Workshop on Cryptography and Security in Computing Systems*, 2014.

[48] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[49] SUSE, 2021. URL: https://documentation.suse.com/sles/12-SP4/pdf/article-vt-best-practices_color_en.pdf.

[50] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In *EuroSys*, 2011.

[51] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *CCS*, 2015.

[52] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security Symposium*, 2020.

[53] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. In *Black Hat US Briefings, Location: Las Vegas, USA*, 2016.

[54] Vmware, 2021. URL: https://kb.vmware.com/s/article/2097593.

[55] S. Wang, Weizhong Qiang, H. Jin, and Jinfeng Yuan. Covertinspector: Identification of shared memory covert timing channel in multi-tenanted cloud. *International Journal of Parallel Programming*, 2015.

[56] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. A covert channel construction in a virtualized environment. In *CCS*, 2012.

[57] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *International Conference on Dependable Systems and Networks (DSN)*, 2013.

[58] Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals Part 1*. Microsoft Press, 7 edition, 2017.

[59] Xin-jie Zhao, Tao Wang, and Yuanyuan Zheng. Cache Timing Attacks on Camellia Block Cipher. *Cryptology ePrint Archive, Report 2009/354*, 2009.

APPENDIX

TIMING DIFFERENCE OF LIBRARY FINGERPRINTING IN PHP

The copy-on-write page faults can be observed in PHP when triggering the deduplication via Memcached Figure 14.

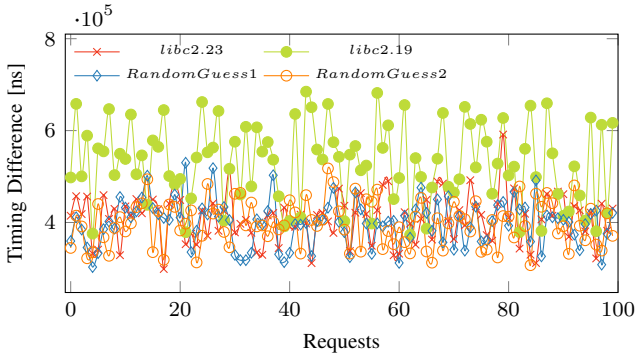


Fig. 14: Timing difference for mapped libc version (2.19) vs. other guesses.

VALIDATION OF REQUIREMENTS FOR REORGANIZATION.

For mounting a successful oracle attack against an InnoDB record, it has to be guaranteed that a reorganization can be triggered reliably. Reorganizing is needed to switch between the different states introduced in Section VI-C3.

The condition for the first reorganize from the initial state (Figure 12a) to the reorganized state (Figure 12b) is already guaranteed by the calculation of the initial size of $|r_{AX}|$ in Section VI-C3.

For the switch from the reorganized state in Figure 12b to the reset state in Figure 12c it must be guaranteed that the restoring of $|r_{AT}|$ always triggers reorganization. Therefore the following inequality must hold:

$$|r_{AT}| > max\_free\_space - |\tilde{r}_{AT}| - |r_T| - |r_{AX}| - footer\_sz$$

We can claim that $|\tilde{r}_{AT}| + |r_T| \geq 4096$ must hold as otherwise the attacker does not even control one full page which is needed for the deduplication side channel. Using this and neglecting the `footer_sz` we get:

$$\begin{aligned} |r_{AT}| = 8125 &> max\_free\_space - (|\tilde{r}_{AT}| + |r_T|) - |r_{AX}| \\ &> 16252 - 4096 - 4064 = 8092 \end{aligned}$$

For the last state switch from the reset state to a new reorganized state there are two possibilities: $|r_{AX}|$ is either increased by δ as long as the resulting size is smaller than the maximum record size or it is set to the maximum record size. In both cases a reorganize should be triggered. Therefore for case 1 the following inequality must hold:

$$\begin{aligned} |r_{AX}| + \delta &> max\_free\_space - (|r_{AT}| - \delta) - |r_T| - |r_{AX}| \\ &\quad - footer\_sz \\ 2 * |r_{AX}| &> max\_free\_space - |r_{AT}| \\ 2 * 4064 = 8128 &> 16252 - 8125 = 8127 \end{aligned}$$

For case two we again use that $|\tilde{r}_{AT}| + |r_T| \geq 4096$ must hold:

$$\begin{aligned} |r_{AX,max}| &> max\_free\_space - |\tilde{r}_{AT}| - |r_T| - |r_{AX}| - footer\_sz \\ |r_{AX,max}| &> max\_free\_space - (|\tilde{r}_{AT}| + |r_T|) - |r_{AX}| \\ 8125 &> 16252 - 4096 - 4064 = 8092 \end{aligned}$$

Required Sizes of Records and Potential Leakage Rate. The record r_{AX} is required to trigger the reorganization of records. Therefore, it initially has to be large enough so that we can trigger a reorganization. We determine the worst case size of the left free space for records within an index page as follows after the first reorganization:

$|r_{AX}| + 1 >$ trailing free space, which is always the case if
$|r_{AX}| + 1 >$ `max_free_space` $- |r_{AX}| - |r_{AT}| - |r_T| (-$`footer_sz`$)$.
`footer_sz`, $|r_T|$ can be neglected in worst case inspection

$$|r_{AX}| > \frac{\texttt{max\_free\_space} - |r_{AT}| - 1}{2} = 4064\,\text{B}$$

therefore:

$$\texttt{left\_free\_space} = 16\,252\,\text{B} - 8125\,\text{B} - 4064\,\text{B} = 4063\,\text{B}.$$

Next we want to determine the boundaries for the shift into our attacker-controlled $4\,\text{kB}$-page and the requirements. We define the maximum alignment change $max\_alignment\_change$ as $r_{AT} - r_{AT_{header}}$. To leak data from r_T, one page of our attacker-controlled $\tilde{r}_{AT}$ record needs to be page aligned. As we chose the size of r_{AT} to be $8125\,\text{B}$, we do not fully control 2 pages. We use a certain part of r_{AT} to bring the last $4096\,\text{B}$ into a page alignment. A certain page misalignment is even required to enable a successful attack, since with a very low misalignment (e.g., 42), we cannot control a full $4\,\text{kB}$ page. For instance with a misalignment of 42 bytes we only control 4071 $(8125 - 4096 + 42)$ bytes of the page to leak the record data (leak page). Therefore, the misalignment needs to be large enough to control a full leak page. Furthermore, the misalignment is unknown and we need to leak the misalignment of the page. This can be done via a remote memory-deduplication attack by trying different offsets until the correct one is found. The misalignment is the start of the record r_{AT} (cf. Figure 12a). We determine the minimal necessary misalignment $\mathit{offset}_{r_{AT}}$ after the initial reorganization:

$$\begin{aligned} \text{offset}_{r_{AT}} mod 4096 + |r_{AT}| - 1 &\geq 4096 * 2 \\ \text{offset}_{r_{AT}} mod 4096 &\geq 4096 * 2 + 1 - |r_{AT}| \\ \text{offset}_{r_{AT}} &\geq 68. \end{aligned}$$

Hence, we need a misalignment of r_{AT} of at least $68\,\text{B}$. Furthermore, in case the record header moves to the leak page, we would have another unknown value to leak. Therefore, the misalignment needs to be smaller or equal to $4096 - |r_{AT_{header}}|$. Each record is preceded by a record header ($r_{AT_{header}}$), which is maximum 27 bytes in our scenario. We have the probability of $0.66\,\%$ that the $r_{AT_{header}}$ moves to the leak page and $1.66\,\%$ that the misalignment is less than 68. We derive the maximum leakage rate for a InnoDB index page as:

$$\begin{aligned} max\_leakage\_possible = \\ \min(|r_{AT}| - 1 - |r_{AT_{header}}| - \\ \text{offset}_{r_T} mod 4096 + |r_{AT}| - 2 \cdot 4096 + 1), |r_T|, 4096) \end{aligned}$$

Hence, we can leak up to a full size of $r_T \leq 4096$ if we leak the misalignment and the requirements for $\text{offset}_{r_{AT}}$ hold. This limits the leakage potential of the InnoDB attack.

MariaDB User Table.

User table used in attack on InnoDB used in MariaDB Figure 15.

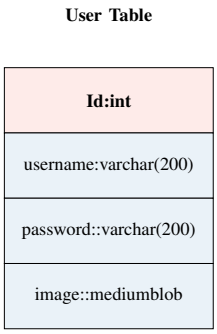


Fig. 15: MariaDB user table, which is susceptible to a remote memory-deduplication attack.

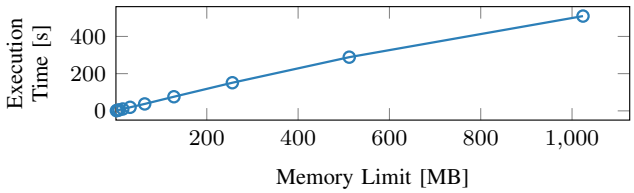


Fig. 16: Average execution time in seconds ($n = 10$) until a newly added entry is evicted from memcached depending on its given memory limit.

Memcached Eviction.

The attacks on Memcached rely on the assumption that an attacker can reliably trigger copy-on-write pagefaults by updating the same item. However, one problem that can occur in the Memcached attack is that another user gets the free deduplicated page assigned instead of the attacker. Therefore, it is a race between the attacker and potential other users to get the page and then trigger the copy-on-write page fault on the deduplicated page. Another issue is whether the pages stays cached in Memcached for a longer period until the deduplication by the operating system happens, *i.e.*, 15 minutes on Windows. The eviction totally depends on the size of the Memcached instance itself.

In this experiment, we validate how long an entry is cached in an Memcached instance with different memory limits. First, we launch a new memcached instance and load as many entries into the instance until the memory limit is exhausted and the instance has to evict existing entries.

Then, we add a new entry and probe how many seconds this entry remains cached while we simultaneously apply a realistic workload on the instance. We utilize memtier_benchmark [46], spawning 4 threads that simultaneously write and read entries from the memcached instance using a gaussian access pattern with an average of 671 121 ops/sec and an average bandwith of $268.26\,\text{MB/s}$. Figure 16 illustrates after how many seconds on average the added entry is evicted from the memcached instance ($n = 10$). Figure 16 illustrates the eviction time for different Memcached node sizes. As can be seen, the larger the node is, the longer it takes to evict a certain item.