Page Cache Attacks

Daniel Gruss
Graz University of Technology

Erik Kraft
Graz University of Technology

Trishita Tiwari
Boston University

Michael Schwarz
Graz University of Technology

Ari Trachtenberg
Boston University

Jason Hennessey
NetApp

Alex Ionescu
CrowdStrike

Anders Fogh
Intel Corporation

ABSTRACT

We present a new side-channel attack that targets one of the most fundamental software caches in modern computer systems: the operating system page cache. The page cache is a pure software cache that contains all disk-backed pages, including program binaries, shared libraries, and other files. On Windows, dynamic pages are also part of this cache and can be attacked as well, e.g., data, heap, and stacks. Our side channel permits unprivileged monitoring of accesses to these pages of other processes, with a spatial resolution of 4 kB and a temporal resolution of 2 µs on Linux (≤ 6.7 measurements per second), and 466 ns on Windows 10 (≤ 223 measurements per second). We systematically analyze the side channel by demonstrating different hardware-agnostic local attacks, including a sandbox-bypassing high-speed covert channel, an ASLR break on Windows 10, and various information leakages that can be used for targeted extortion, spam campaigns, and more directly for UI redressing attacks. We also show that, as with hardware cache attacks, we can attack the generation of temporary passwords on vulnerable cryptographic implementations. Our hardware-agnostic attacks can be mitigated with our proposed security patches, but the basic side channel remains exploitable via timing measurements. We demonstrate this with a remote covert channel exfiltrating information from a colluding process through innocuous server requests.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

Software-based Attacks; Cache Attacks; Operating Systems

ACM Reference Format:
Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. 2019. Page Cache Attacks. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3319535.3339809

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3339809

1 INTRODUCTION

Modern processors are highly optimized for performance and efficiency. *Caching* takes advantage of temporal and spatial locality to minimize slower memory or disk accesses. Caches typically fetch or prefetch code and data into fast buffers closer to the processor.

While side channels have been known and utilized primarily in military contexts for decades [47, 91], the idea of hardware-cache side-channel attacks has gained more attention over the last twenty years [3, 46, 61]. Notably, Osvik et al. [58] showed that an attacker can observe the cache state at the granularity of a cache set using Prime+Probe, and later Yarom et al. [90] showed this with cache line granularity using Flush+Reload.

The pages targeted by Flush+Reload attacks reside in the so-called page cache, a purely software cache that is implemented in all major contemporary operating systems and contains virtually all pages in active use. The operating system uses the page cache to store frequently used pages in memory, obviating otherwise slow disk loads; pages that contain data accessible to multiple programs, such as disk-backed pages (e.g., program binaries, shared libraries, other files, etc.), are shared among all processes regardless of privilege and permission boundaries [27]. There is a large body of works exploiting Flush+Reload in various scenarios over the past several years [34, 39–41, 48, 90]. There have also been a series of software (side-channel) cache attacks in the literature, including attacks on the browser cache [5, 23, 42, 43, 83] and exploiting page deduplication [2, 6, 30, 59, 63, 78, 87, 88]; however, page deduplication is mostly disabled or limited to deduplication within a security domain today [52, 64, 82].

In this paper, we present a new attack that directly targets the software-based operating system page cache, and is thus hardware agnostic. We present a set of local attacks that work entirely without timers, utilizing operating system calls (`mincore` on Linux and Android, and `QueryWorkingSetEx` on Windows) to elicit page cache information. We also show that page cache metadata can leak to a remote attacker over a network channel, producing a stealthy covert channel between a malicious local sender process and an external attacker.

We comprehensively evaluate and characterize our software-cache side channel by comparing it to known hardware-cache side channels. To this end, our attacks are similar in effect to DRAMA attacks [62, 85], in that they work across cores and across CPUs; the temporal granularity of the DRAMA attack is around 300 ns, whereas the temporal granularity of our attack is 2 µs on Linux (≤ 6.7 measurements per second) and 466 ns on Windows (≤ 223

measurements per second). In other aspects, our attacks are similar to controlled-channel attacks on TEEs [81, 89], in that the information leaked is whether or not a page was used within a certain time frame, implying a spatial granularity of 4 kB. We conclude that our attack can compete with the current state-of-the-art in microarchitectural attacks.

Finally, we present several ways to mitigate our attack in software, and observe that certain page replacement algorithms reduce the applicability of our attack while simultaneously improving the system performance. In our responsible disclosure, both Microsoft and the Linux security team acknowledged the problem and plan to follow our recommendations with security patches to mitigate our attack, which has also been codified as CVE-2019-5489.

To summarize, we make the following contributions:

(1) We present a novel attack targeting the page cache.
(2) We present a high-speed covert channel and a set of local attacks which are hardware-agnostic and can compete with state-of-the-art microarchitectural attacks.
(3) We present timing-based attacks in form of a remote attack which can leak information across the network.

We begin in Section 2 with background information on hardware caches, cache attacks, and software caches, followed by our threat model in Section 3. Section 4 overviews our attack. Section 5 presents a novel method to spy on the page cache state. Section 6 shows how page cache eviction can be done efficiently on Linux, Android, and Windows. Section 7 presents hardware-agnostic local page cache attacks. Section 8 presents timing-based remote page cache attacks. Section 9 discusses different countermeasures against our attack. Section 10 concludes our work.

2 BACKGROUND

We begin with a brief discussion of hardware and software cache attacks, followed by some background on the operating system page cache that we exploit.

2.1 Hardware and Software Cache Attacks

The suggestion of cache attacks harks back to the timing attacks of Kocher [46]. Osvik et al. [58] presented a technique with a finer granularity called Prime+Probe. Yarom et al. [90] presented Flush+Reload, which is still today the cache attack technique with the highest accuracy (virtually no false negatives or false positives) and a finer granularity than most other attacks (one cache line). Consequently, Flush+Reload is also used in other applications, including the covert channel in Spectre [45] and Meltdown [49]. Flush+Reload requires shared memory with the victim application. However, all modern operating systems share code and unmodified data of every program and shared library (and any unmodified file-backed page in general) across privilege boundaries and applications.

Caches also exist in software, caching remote data, data that has been retrieved from slow or offline storage, or precomputed results. Some of these caches have very specific use-cases, such as browser caches used for website content; other caches are more generic, such as the page cache that stores a large portion of code and data used. Caches make use of the principle of locality to retain common computations closer to the processor, and consequently they can leak information about the cache contents.

For example, browser caches leak information about browsing history and other possibly sensitive user information [5, 23, 42, 43, 83]. Requested resources may have different access times, depending on whether the resource is being served from a local cache or a remote server, and these differences can be distinguished by an attacker. As another example of a software-based side channel, page-deduplication attacks exploit page deduplication across security boundaries. A copy-on-write page fault reveals the fact that the requested page was deduplicated and that another process must have a page with identical content. Suzaki et al. [78] presented the first page-deduplication attack, which detected programs running in co-located virtual machines. Subsequently, several other page-deduplication attacks were demonstrated [30, 59, 87, 88]. Today, page deduplication is either completely disabled for security reasons or restricted to deduplication within a security domain [6, 52, 63, 64].

2.2 Operating System Page Cache

Virtual memory creates the illusion for each involved process of running alone on the system. To do this, it provides isolation between processes so that different processes may operate on the same addresses without interfering with each other. Each virtual memory page may be mapped by the operating system, with varying properties, to an arbitrary physical memory page.

When multiple processes map a virtual page to the same physical page, this page is part of *shared memory*. Shared memory typically may arise out of inter-process communication or, more broadly, to reduce physical memory consumption. For example, if shared library and common binary pages on the hard disk are mapped multiple times by different processes, they map to the same pages in physical memory.

Any page that might be used by more than one process may be mapped as shared memory. However, if a process wants to write to such a page, it must first secure a private copy of the page, so as not to break the isolation between processes. The efficiency savings come because a great many pages are never modified and, instead, remain shared among multiple processes in a read-only state.

The operating system page cache is a generalization of the above memory sharing scenario, and, in fact, all modern operating systems (e.g., Windows, Linux, Android, and OS X) implement a page cache. The page cache contains not only shared pages, but all pages that are memory mapped files, any file read from the disk, and (depending on the system) possibly other pages such as anonymous pages or shared memory [27]. The operating system keeps track of which pages in the page cache are clean (*i.e.*, their data is unmodified from the disk version) and which are dirty (*i.e.*, modified since they were first loaded from the disk). Ideally, the page cache incorporates all available memory, minimizing the disk I/O.

The introduction of a page cache disrupts the traditional functioning of the operating system under a page fault. Without a page cache, the operating system reserves a free physical page frame, loads the data from the disk into that physical page frame, and then maps a virtual page to the physical page frame accordingly. If there are no available physical page frames, the system swaps out pages to the disk using an operating system-dependent page-replacement algorithm. In Linux, this algorithm had traditionally been based on

a variant of Least Recently Used (LRU) [12], and LRU-related data structures can still be found throughout the kernel code. More recent Linux versions implement an improved variant called CLOCK-Pro [44] along with several adaptions [13]. Within this improved framework, Linux moves pages among multiple lists (an inactive list, an active list, and a recently evicted list). In contrast to Linux, Windows uses the *working-set* model of page caching to introduce more fairness among processes competing for memory [9, 19, 20]. The page replacement algorithm used on Windows was based on Clock or pseudo-random replacement [26, 68] in older Windows versions, and today is likely a variant of the Aging algorithm [8].

With a page cache, the operating system endeavors to make full use of all physical page frames, and a page-replacement algorithm is still needed for evicting page cache pages (swapping is less relevant on modern operating systems [17, 18, 37]). Also pages from KVM virtual machines are cached in the host-side page cache if the machine is configured to use a write-back caching strategy [21].

Both Linux and Windows provide mechanisms for checking whether a page is resident in the page cache - the `mincore` system call for Linux (and Android), and the `QueryWorkingSetEx` system call for Windows.

3 THREAT MODEL

Our threat model is based on the threat model for Flush+Reload [34, 39–41, 48, 90].

Specifically, we assume that attacker and victim have access to the same operating system page cache. On Linux, we also assume that the attacker has read access to the target page, which may be any page of any attacker-accessible file on the system. This assumption is satisfied, for example, when attacker and victim are

- processes running under the same operating system, or
- processes running in isolated sandboxes with a copy-on-write file system mapping (e.g., Firejail [24], (s)chroot, etc.).

On Windows, read access to the target page is not necessary for our attack, *i.e.*, our attack works on **non-shared** pages.

Our local attacks are timing-free, in that they do not rely on hardware timing differences. Our remote attack leverages timing differences between memory and disk access, measured on a remote system, as a proxy for the required local information.

4 HIGH-LEVEL VIEW OF THE ATTACK

Our attack fundamentally relies on the attacker's capability to distinguish whether a page is in the page cache or not. In the local attack we are agnostic to the underlying hardware, *i.e.*, we do not exploit any timing differences although this would be practically possible on virtually all systems. Thus, we use the `mincore` system call on Linux and Android for this purpose and the `QueryWorkingSetEx` system call on Windows. The `mincore` system call returns which pages of a memory range are present in memory (*i.e.*, in the page cache) and which are not. Likewise, the `QueryWorkingSetEx` system call returns a list of pages that are in the current working set of a process, and thus are present in the page cache.

Bringing the page cache into a known state is not trivial, as it behaves like a fully associative cache. Previous approaches for page cache eviction can lead to out-of-memory situations [32, 75, 82] or consume too much time and impose system pressure [31]. This is not practical when evicting pages often, e.g., multiple times per second. Hence, they have not been used in published side-channel attacks so far, but only to support other attacks, e.g., relocation of a page for Rowhammer. For Linux, we devise an eviction strategy that forms working sets of pages and accesses them frequently. As the page to evict is accessed less frequently (never by the attacker), it is evicted in the process.

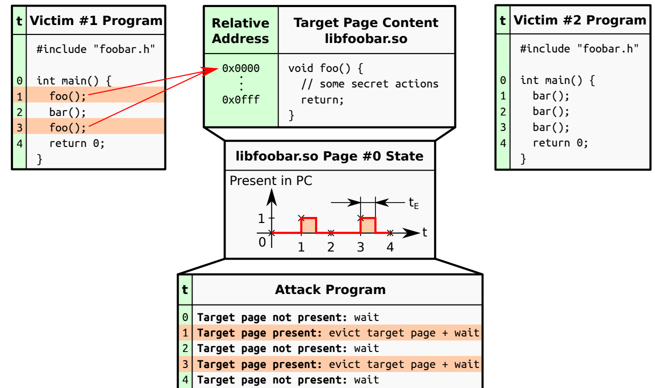

Figure 1: Attack overview.

On Windows, our attack is much more efficient than on Linux. On Linux, the page cache is directly influenced by all processes. In contrast, Windows has per-process working sets [53], and the page cache is influenced indirectly through these working sets. Hence, for Windows, we present an attack which evicts pages only from the working set of the victim process, but not from the page cache (*i.e.*, not from DRAM), *i.e.*, causing no additional disk accesses. Although both attack variants follow the same attack methodology, we have to distinguish between the Linux and Windows variant at several places in the remainder of the paper.

In contrast to hardware cache attacks and page-deduplication attacks, our local attacks are non-destructive, allowing us to repeat measurements. Measuring whether a memory location is cached or not manipulates the state such that the information is not available anymore at a later point in both hardware cache attacks [58, 90] and page-deduplication attacks [59, 78]. However, it is not the case for our local attack. As we rely on the `mincore` and `QueryWorkingSetEx` system calls, we can arbitrarily check whether the page is in the page cache (on Linux) or the process working set [53] (Windows). These checks are non-destructive as they neither modify nor influence the state of the page cache or the process working set with respect to the target memory location.

Our attack is illustrated in Figure 1. The attacker wants to measure when the function `foo()` is called by a victim program. The attacker determines the page which contains the function `foo()`. By observing when the page is in the page cache, the attacker learns when `foo()` was called.

Our attack continuously runs through the following steps: Initially, the target pages are in the page cache (on Linux) respectively the working set of the victim process (on Windows). After the eviction, the page is not in the page cache (Linux) or process working set (Windows) anymore. The attacker can now continuously probe when the page is added back in. As soon as the page is found in

the page cache (Linux) or the process working set (Windows), the attacker logs the memory access and evicts the page again.

In the following sections, we detail the two main steps of the attack, *i.e.*, determining the page cache state (defining the temporal resolution) and performing the page cache eviction (defining the maximum frequency at which the attack can be performed).

5 DETERMINING THE PAGE CACHE STATE

In this section, we discuss how to determine the page cache state. Note that although our attack starts with the page cache eviction, following the attack description is easier when understanding how to determine the page cache state first.

The attacker wants to determine when a specific page in a victim process is loaded into the page cache, as this is exactly the access time of the victim program. On Linux, the binary containing the page targetted by an attacker has to be mapped into the attacker's address space, using `mmap`. On Windows the same can be achieved via `LoadLibraryEx` or `CreateFileMappingA` and `MapViewOfFile`, but this is not necessary for our attack, as we detail in Section 5.1.

To map the shared library, the user only requires read-only access to the file containing the target page. As the attacker process works on its own mapping of the shared library, all addresses are observed relative to the start of the shared library. Hence, security mechanisms such as Address Space Layout Randomization (ASLR) have no effect on our attack on Linux. On Windows attack even allows to break ASLR in other processes, as we detail in Section 5.1.

To determine whether or not a page is in the page cache, we rely on the operating-system provides APIs to query the page cache. On Linux, this API is provided by the `mincore` system call. `mincore` expects the base address and length of a memory area and returns a vector indicating for each page whether it is in the page cache or not. On Windows, there are two variants we discuss.

5.1 Windows Process Working-Set State

On Windows, every process has a working set which is a very small subset of the page cache. We cannot query the page cache directly as on Linux but instead we focus on the working set. While this makes determining the cache state more complex, the following eviction is much easier and faster (cf. Section 6.2). On Windows, we rely on the `QueryWorkingSetEx` system call. This function takes a process handle and an array specifying the virtual addresses of interest as arguments. It returns a vector of structures which, if the page is part of the working set, contain various information about the corresponding pages. We devise two different variants to determine whether or not a page is in the working set of a process based on the return value of the `QueryWorkingSetEx` system call.

Variant 1: Low Share Count and Attacker-Readable.
The `ShareCount` represents the number of processes that have this page in their working set. It is one of the members in the structure returned by `QueryWorkingSetEx`. Unfortunately, the value is capped to 7 processes, *i.e.*, if more processes have the page in their working set, the number remains 7. However, as the working-set size is limited to 1.4 MB by default, this rarely happens for a page. In fact, most pages in the page cache have a `ShareCount` of 0 due to the small working-set sizes. With this variant, we do not need **any permissions** for other processes and only default permissions on our own process. Hence, we can mount the attack even across users without restrictions.

Variant 2: High Share Count or Not Attacker-Readable.
If the `ShareCount` is 7 or larger, we cannot gain any information by calling `QueryWorkingSetEx` on our own process. Instead, we can use `QueryWorkingSetEx` directly on the victim process. In contrast to the official documentation [53], the `QueryWorkingSetEx` system call only requires the `PROCESS_QUERY_LIMITED_INFORMATION` permission. By default, the attacker process has this permission for handles of other processes of the same user but also for some processes with a **higher** integrity level (as part of the generic execute access) [54]. This allows us to use `QueryWorkingSetEx` to break ASLR in these other processes. Even worse, this allows us to spy on pages that are **not shared**, *i.e.*, not attacker-readable at all, `QueryWorkingSetEx` still leaks information whether the page is currently in the page cache or not. Hence, we can use `QueryWorkingSetEx` to directly break ASLR in the victim process and also determine directly whether the target page is in the working set of the victim process.

5.2 Spatial and Temporal Granularity

One limitation of our attack is the coarse spatial granularity of 4 kB, *i.e.*, one page. This is identical to TLB- [28, 29] and DRAM-based attacks [62, 85], but also to controlled-channel attacks on TEEs [81, 89]. If a target region contains other frequently used data, the signal-to-noise ratio decreases in these attacks same as in ours. However, this just increases the number of measurements an attacker has to perform.

The temporal granularity of the DRAMA attack is constrained by the time it takes to run one or two rounds of Flush+Reload, which is around 300 ns [62, 85]. The temporal granularity of our attack is constrained by the time the system call consumes, which we observed to be 2.04 µs on average for `mincore` with a standard error of 20 ns, and 465.91 ns on average for `QueryWorkingSetEx` with a standard error of 0.20 ns on our Intel i7-6700K. Hence, on Linux, it is only 6.8 times lower than the DRAMA attack, and on Windows only 55 % lower than the DRAMA attack. Thus, our attack can be used as a reasonable replacement for the hardware-dependent DRAMA attack. However, as we describe in Section 6, the eviction limits how often an attacker can measure, *i.e.*, 6.7 times per second on Linux and 223 times per second on Windows.

5.3 `procfs` and Timing

As an alternative to `mincore` on Linux, we also tried to mount the same attack using information from `/proc/self/pagemap`. However, `/proc/self/pagemap` only shows the information from the page translation tables. As operating systems commonly use lazy page mapping, the page is in practice not mapped into the attacker process and thus, the information in `/proc/self/pagemap` does not change. Furthermore, as a response to Rowhammer attacks [75], access to `/proc/self/pagemap` was first restricted and nowadays it is often not accessible by unprivileged processes.

Timing of pagefaults is a generic alternative to `mincore` and `QueryWorkingSetEx`. Accessing a page may trigger a pagefault. Measuring the time it takes to handle the pagefault reveals whether it was a soft pagefault, just mapping a page from the page cache, or a

regular pagefault, loading data from the disk. The timing differences we observed are easy to distinguish, with 1–2 orders of magnitude between the two cases. In our remote attack we exploit these timing differences. However, this makes page cache eviction more difficult as the accessed page is now the least-recently used one.

Finally, as stated in Section 3, our local attacks are entirely attack hardware-agnostic. Hence, we cannot use any timing differences in our local attacks.

6 PAGE CACHE EVICTION

In this section, we discuss how page cache eviction can be implemented efficiently on Linux and Windows systems. Page cache eviction is the process of accessing enough pages in the right way such that a target page is evicted. We show that we improve over state-of-the-art eviction algorithms by 1 to 2 orders of magnitude, enabling practical side-channel attacks through the page cache for the first time.

Less efficient variants of page cache eviction have been used in previous work [31, 36]. Holen et al. [36] generates a large amount of data, simply exhausting the physical memory. Using this approach it takes 8 s or more to evict a target page on Linux. Furthermore, when reproducing their results we observed severe stability issues, constantly leading to crashes and system lock-ups during eviction. The technique presented by Gruss et al. [31] takes 2.68 s on Linux to evict a target page. On Windows, their technique is slower, with an average execution time of 10.1 s. State-of-the-art microarchitectural side-channel attacks have a higher temporal resolution by more than 6 orders of magnitude [51, 62, 90]. Hence, we can conclude that page cache eviction, as done in previous work, would be far too slow for side-channel attacks with a relevant frequency. We solve this problem by combining the technique from Section 5 with efficient page cache eviction on Linux (Section 6.1) and process working-set eviction on Windows (Section 6.2), allowing page cache attacks with a decent frequency. Our technique for Linux is also applicable to Android, as Android is based on Linux and also exposes the required functions to user-space applications.

6.1 Efficient Page Cache Eviction on Linux

The optimal cache eviction for the attacker would evict only the target page of the victim, without affecting other cached pages. Our idea is to constantly access a large number of pages which are already in the page cache (to keep them there) and then access a few non-cached pages to evict the target page.

In a feasibility analysis, we measured how many pages an attacker can locate inside the page cache. On our test system, we had 1 040 542 files accessible to the attacker program, amounting to 77 GB of disk space. We found that less than 1 % of the files had pages in the page cache, still amounting to 68 % to 72 % of the total page cache pages. This information is available to an unprivileged attacker using system calls like `mmap` and `mincore`.

The attacker then creates a list of all pages currently in the page cache and a list of further pages that could be loaded into the page cache to increase memory pressure. Both lists are updated occasionally to reflect changes in how the system is utilized currently. To also reflect the current memory load, the attacker adapts how many pages in each of the two lists are accessed to achieve efficient cache eviction.

Thus, the attacker creates 3 eviction sets:

Eviction Set 1. The pages in this set are already in the page cache, used by other processes. To keep them in the page cache, a thread continuously accesses these pages. The system load is kept low by using `sched_yield` and `sleep` frequently. Consequently, the pages in this set are among the most recently accessed pages of the system and eviction of these pages is highly unlikely.

Eviction Set 2. The pages in this set are not yet in the page cache. Pages in this eviction set are randomly accessed, to avoid repeated accesses and thus any similarity to the pages in eviction set 1 for the replacement algorithm.

Eviction Set 3. We use another eviction set, namely non-evictable pages, e.g., dynamic content. The attacker has to make sure these pages cannot be swapped, e.g., by disabling swapping. These pages are only created and filled with content, but never again read or written. Hence, they block a certain amount of memory, reducing the required eviction-set size. This reduces the runtime of the eviction significantly. Still, this introduces no stability issues, as we always keep a large amount of pages ready for immediate eviction, *i.e.*, the other eviction sets.

Alternative Approaches and Optimizations. For our tests, we used `ext4` as a file system. We investigated the influence of the file system type, by running our attack also on `XFS` and `ReiserFS`. We didn't find a relevant influence of the file system type on the attack performance.

We also investigated whether `madvise` and `posix_fadvise` system calls on Linux can improve the attack performance. These system calls allow a programmer to provide usage hints for a given memory or file range to the kernel. The advice `MADV_DONTNEED` indicates that the process will not access the specified pages any time soon again, whereas the advice `MADV_WILLNEED` indicates that the process will soon access the specified pages again. Thus, the operating system will evict the corresponding pages from the page cache. We found that marking the target page as `MADV_DONTNEED` and all eviction set pages as `MADV_WILLNEED` was often ignored by the kernel, which ignores these hints unless the process exclusively owns the pages (`madvise`) or when no other process has the file mapped (`posix_fadvise`). Still, this allows to use `posix_fadvise` on files regardless how frequently they are accessed, e.g., via `read()`, as long as they are not mapped. Hence, we are able to mount a covert channel by using `posix_fadvise` on a file which is not mapped by any (other) process, instead of eviction.

6.1.1 Evaluation. We measured the precision and recall of our eviction by monitoring a periodic event which was triggered every second. The page cache eviction using all 3 eviction sets simultaneously achieves an average runtime of 149 ms ($\sigma = 1.3$ ms) on average and an F-Score of 1.0

Hence, while the temporal resolution of our attack is generally 2.04 µs on Linux, the maximum rate at which events can be observed can be lower. The reason is that, if (and only if) the event occurs, eviction is necessary, and thus, we cannot perform any measurements within these 149 ms needed for eviction. This still allows capturing more than 6 keystrokes per second, enough to capture keystrokes accurately for most users [73].

The temporal resolution is significantly higher than that of page-deduplication attacks. The frequency at which page deduplication happens is lower the more memory the system has, and has in use, and the less power the device should invest in deduplication. In practice deduplication happens every 2 to 45 minutes, depending on the system configuration [30]. Hence, our attack has an at least 800 times higher temporal resolution than the best page-deduplication attacks.

Limitations. One obvious limitation of our approach is that the target page has to be in the page cache. However, as detailed in Section 2.2, virtually all pages used by user programs end up in the page cache, even dynamically allocated ones.

On Linux, the page also must be accessible to the attacker, e.g., file-backed memory such as binary pages, shared library pages, or other files. This is exactly the same requirement (and limitation) of Flush+Reload attacks [34, 39–41, 48, 90]. Other microarchitectural attacks, e.g., Prime+Probe, may not have this requirement but usually have other similarly constraining requirements, such as knowledge of the physical address which is difficult to obtain in practice [51]. Page-deduplication attacks also do not have this limitation, but they face other limitations such as a significantly lower temporal resolution and, more recently, that page deduplication is mostly disabled or limited to deduplication within a security domain [52, 64, 82]. On Windows, we do not have these limitations: We can attack any page in the victim process, including data pages, heap pages, stack pages, and of course any executable and shared library pages.

Like other cache attacks, the side channel experiences noise if the target location is not only used by the event the attacker wants to spy on but also other events. This is the same limitation as for any other cache side-channel attack [34, 90]. Another limitation for hardware cache attacks is prefetching [34, 90]. Unsurprisingly, software again implements the same techniques as hardware. When accessing the SSD, the Linux kernel reads ahead to increase the performance of file accesses. If not specified otherwise, the readahead window is 32 pages large, cf. `/sys/block/sda/queue/read_ahead_kb`. This is similar to the adjacent line prefetcher and the streaming prefetcher in hardware. Whenever a cache miss occurs, the adjacent line prefetcher always fetches the sibling cache line region into the cache, *i.e.*, the adjacent 64 B. Whenever a second cache miss within a page occurs, the streaming prefetcher reads ahead of the cache miss and reads up to 512 B (*i.e.*, 8 cache lines) into the cache. Gruss et al. [34] noted that this limits their attack to a small number of memory locations per page. The same limitations apply to our work, *i.e.*, *simultaneously* monitoring multiple pages within a 32-page window can be noisy. However, this does not significantly reduce the abundance of side channel targets for cache attacks. To avoid triggering the prefetcher, we add the pages surrounding the target page to the eviction set 1, *i.e.*, we reduce their chance of being evicted, in order to avoid all noise from prefetching, as no other page from this range will be accessed.

Finally, the attacker process can, of course, only perform measurements and evictions when it is scheduled. Hence, scheduling can introduce false negatives into our attack. Again, this is also the case for hardware cache attacks and it can be compensated for [51].

Compared to previous work, we improve the state-of-the-art for page cache eviction by a factor of more than 16 and additionally avoid cache eviction in most cases (cf. Section 5). With these two building blocks, we are able to mount practical attacks as demonstrated in the following sections. The ideal target for our attack is a function or data block which is used at frequencies below 8 times per second, allowing our attack to yield a temporal resolution of up to 2 µs. Furthermore, the noise is the lowest if the target code or data resides on a page which is not much used by other functionality.

6.2 Process Working-Set Eviction on Windows

As previous page cache eviction techniques [31, 36] are too slow to mount generic side-channel attacks, we pursue a different approach on Windows. Windows has per-process working sets [53], which (by default) are constrained to a size between 100 kB and 1.4 MB [53]. Hence, we evict a page from the process working set rather than from the page cache. Our results show that the runtime of the eviction is on par with eviction in hardware cache attacks.

We use process working-set eviction in both, covert channels and side-channel attacks. For a covert channel, the sender can add pages to the working set, e.g., by accessing them. To evict pages, we use an unintended behavior of `VirtualUnlock` that comes from a programming error [54]. Calling `VirtualUnlock` on a page which is not locked evicts it directly from the working set. For reasons of backward-compatibility, the behavior was never changed [54]. Additionally, pages which are only read in one of the processes can be locked, so that they are never removed from the working set. This way, arbitrary information can be encoded into the `ShareCount` of the page cache pages – up to 3 bits exist, which allows 7 sharers. Hence, we can transmit arbitrary information without any special privileges (as long as the receiver is not constrained by an App Container). The default maximum working-set size is 1.4 MB. As the page size is 4 kB, that is, there are at most 345 page slots in the working set by default [53]. Hence, we can exploit self-eviction (from the working set) for the side channel, which can happen frequently with a little heavy memory pressure because of the small working-set size. Pages that are not accessed are evicted from the working set, but remain in RAM and mapped in the process. However, we can speed up eviction by reducing the victim process' working-set size using `SetProcessWorkingSetSize` and even evict all pages from the working set on the other process [53]. The lowest possible value for the maximum working-set size is 13 pages (52 kB).

6.2.1 Evaluation. We found that `VirtualUnlock` has a success rate of 100 % over several million tests. The average time to evict a page from the process working set with `VirtualUnlock` is 4.48 ms with a standard error of 3.6 µs.

Similarly to Linux (cf. Section 6.1), the higher runtime of the eviction has a local influence on the temporal resolution of our attack. Generally, the temporal resolution of our attack on Windows is 466 ns, which is only 55 % lower than the temporal resolution of the DRAMA attack [62, 85]. The eviction on Windows via `VirtualUnlock` consumes 4.48 ms during which no second measurement can be taken. This is by far fast enough for inter-keystroke timing attacks [56, 73].

While prefetching posed a relevant limitation on Linux, it is no problem on Windows. On Windows, features like SuperFetch fetch memory into the page cache, acting like an intelligent hardware

prefetcher or speculative execution. Indeed, SuperFetch speculatively prefetches pages from the disk into the main memory, based on similar past usage, e.g., same time of day, same sequence of applications started [72]. However, these pages are not added to the working set of any process. Thus, our side channel remains entirely unaffected by these Windows features. This makes the side channel very well suited for inter-keystroke timing attacks [56, 73]. Still, future work should investigate whether the speculative nature of SuperFetch can be exploited by other means, such as timing, e.g., leaking valuable information about the user behavior.

Limitations. `VirtualUnlock` works on our own process and requires no permissions, yet it also evicts the corresponding pages from all other processes. Thus, it only works for read-only shared pages, e.g., file-backed pages. The `SetProcessWorkingSetSize` system call requires the `PROCESS_SET_QUOTA` permission on the process handle [53]. By default, the attacker process has this permission for handles of other processes of the same user running on the same or a lower integrity level. Hence, by default, processes with a higher integrity level, e.g., Administrator processes, must be attacked with eviction if both `VirtualUnlock` and `SetProcessWorkingSetSize` are not applicable [54].

7 LOCAL ATTACKS

In this section we present and evaluate our local attacks. The temporal resolution naturally scales with the performance of the system. We perform all performance evaluations on recent systems with multiple gigabytes of RAM, with off-the-shelf mid-class consumer SSDs (e.g., transfer rates above 250 MB/s [79]). For our tests on Linux, we have swapping disabled. This is recommended with recent processors (e.g., Haswell or newer) and to reduce disk wear [17, 18, 37], and it is also by default the case on Android (cf. Section 7.8 for the experiments on Android). Disabling swapping allows for a better comparison with related work which also focuses on such recent systems [40, 48, 62, 85]. Running our attacks inside a sandbox did not have any measurable effect on the performance of our attacks, e.g., running it inside a Firejail sandbox [24] or other cgroups-based sandbox which share the host system page cache.

7.1 Covert Channel

To systematically evaluate the page cache side channel, we adapt different state-of-the-art hardware cache attacks to it and demonstrate that they achieve a comparable performance. In this section, we cover the first example, a covert channel between two processes additionally isolated by running them in different sandboxes. The strongly isolated sender process sends a secret file from a restrained environment to a receiver process which can forward the data to the attacker.

As evicting a page is comparably slow (cf. Section 6), and checking the state of a page is comparably fast (cf. Section 5), it is optimal to reduce the number of evictions. Hence, it is more efficient to transmit multiple bits at once. We took this into account for the design of our covert channel. We follow the basic principle of hardware cache covert channels [33, 50, 51, 62]. First, a large shared file (e.g., a shared library) is mapped read-only into the address space of the sender and receiver process. As described in Section 4, we use `mmap` for this purpose on Linux. On Windows, we use `CreateFileMappingA` and `MapViewOfFile` for the same purpose.

The covert channel works by accessing or not accessing specific pages. We use two pages to transmit a 'READY' signal and one page to transmit an 'ACK' signal. The remaining pages up to the end of the file are used as data transmission bits. The two 'READY' pages are used alternately to avoid any race conditions in the protocol between the transmission of two subsequent messages. On Windows, we use two 'READY' pages and two 'ACK' pages, for the two transmission directions.

The present state of each page of the mapped file (cf. Section 5) corresponds to one bit of the message. Hence, the size of the file defines the maximum message size of a single transmission. To avoid the prefetcher, we only allow a single access in a region of 32 pages. If the file has a size S, the (maximum) message size is computed as $w = \frac{S}{4096 \cdot 32}$ bits. For instance, on Linux, Firefox' `libxul.so` or Chromium's `chromium-browser` binaries are more than 100 MB large. Similarly, large files can also be found on Windows.

These large files allow transmitting more than 3200 bits in a single message including the 3 pages required for the control channels. To avoid the introduction of noise, the attacker can skip noisy pages, *i.e.*, pages which are also accessed by other system activity. By combining pages from multiple shared libraries, the attacker can easily find a significantly higher number of pages that can be used for transmissions, leading to very large message sizes w. The pages are numbered from $0, 1, .., i, .., w$, *i.e.*, it is not relevant which file they belong to. Instead of a static list of files to check, the attacker could also use a dynamic approach and a jamming-agreement protocol [51].

To exchange a message, the sender first checks the present state of the 'ACK' page (cf. Section 5). If the 'ACK' page is present, the sender knows the receiver is ready for the next transmission. The sender then evicts (cf. Section 6) any pages that are mapped, e.g., from previous transmissions. After that, the sender reads the next w bits (w is the message size) from the secret to transmit. If the i-th bit is set, page i page is accessed. Otherwise, page i is not accessed. As soon as the sender is done with accessing the data transmission pages, it accesses the currently to-be-set 'READY' page, to signal the receiver to start reading the message.

On the other side, the receiver first waits until a 'READY' page is present. As soon as it is set, the receiver reads the message by analyzing the present state of the pages of the memory mapped files. After that, the receiver accesses the 'ACK' page again to inform the sender that it is ready for the next message.

While above protocol is implemented with `mmap`, `mincore` (cf. Section 5), and page cache eviction (cf. Section 6.1) on Linux, we use a slightly different mechanism on Windows as we only work with working-set eviction (cf. Section 6.2). On Windows, we lock pages in the working set which should always remain in the working set, *i.e.*, the 'READY' and 'ACK' bit pages of the sender and the receiver process on the corresponding receiving side. Additionally, we increase the minimal working-set size so that none of the pages we use are removed from the working set. We temporarily add pages into the working set by accessing them and remove pages surgically from the working set by calling `VirtualUnlock`. Hence, the covert channel information is perfectly (no information loss) stored

in the page cache in the ShareCount for the shared pages. Using QueryWorkingSetEx the receiving side can read the ShareCount and decode the information that was encoded in the page cache.

Performance Evaluation. We tested the implementation by transmitting random messages between two processes. The test system was equipped with an Intel i5-5200U processor, 8 GB DDR3-1600 RAM, and a 256 GB Samsung SSD.

For the tests on Linux, we used Ubuntu 16.04 with kernel version 4.4.0-101-generic. We observed transmission rates of up to 9.69 kB/s with an average transmission rate of 7.04 kB/s with a standard error of 0.18 kB/s. We did not observe any influence by the core or CPU scheduling, which is not surprising, as both the system calls and the page cache eviction can equally run on any core or CPU. We observed a bit-error rate of less than 0.000 03 %. We evaluated the performance of the covert channel across sandboxes using a cgroups-based sandbox (e.g., Firejail [24]), preventing all outgoing inter-process communication, denying all network traffic, and only allowing read access to the file system. We did not observe any interference from running the covert channel in Firejail or other sandboxes and containers using the host system page cache, e.g., (s)chroot, or Docker, if configured accordingly.

For the tests on Windows, we used two different hardware setups with fully updated Windows 10 installations. On the Intel i5-5200U system, we observed transmission rates of up to 152.57 kB/s with an average transmission rate of 100.11 kB/s with a standard error of 0.79 kB/s and a bit-error rate below 0.000 006 %. On a second system, an Intel i7-6700K with a SanDisk Ultra II 480GB SATA SSD (running Ubuntu 19.04 with a 4.18.0-11-generic kernel), we observed transmission rates of up to 278.16 kB/s with an average transmission rate of 273.44 kB/s with a standard error of 0.23 kB/s, again with a bit-error rate below 0.000 006 %.

For a performance comparison in a similar cross-CPU scenario, Pessl et al. [62] reported an error rate of 0.4 % for the DRAMA covert channel, albeit with a channel capacity of 74.5 kB/s which is much slower than our covert channel on Windows, but faster than our covert channel on Linux. Wu et al. [86] presented a cross-CPU covert channel with a channel capacity of 93.25 B/s. Hence, our Linux covert channel outperforms this one by two orders of magnitude and our Windows covert channel even by three to four orders of magnitude. In particular, the covert channel on the i7-6700K test system can even compete with Flush+Reload and Flush+Flush covert channels which require specific hardware (Intel processors) and shared memory [33]. Thus, we conclude that our covert channel can very well compete with state-of-the-art hardware-component-based covert channels. Yet, our covert channel works regardless of the presence of these leaking hardware components.

7.2 Breaking ASLR on Windows 10

In this section, we break ASLR in other processes through the use of the page cache side channel. Breaking ASLR undermines the main line of defense against memory-error attacks [2] in both local and remote attacks [76]. Indeed, different memory errors require different regions of the process to be derandomized for exploitation. In the case of Windows 10, we note that the operating system randomizes the location of executable, heap, and stack per process, but the locations of shared libraries are the same across all processes.

For our attack, we use QueryWorkingSetEx directly on the victim process. This only requires the PROCESS_QUERY_LIMITED_INFORMATION permission, which the attacker process does not only have for handles of other processes of the same user, but also for some processes with a **higher** integrity level, cf. Section 5.1. As we only want to locate cached pages, it does not make sense to evict pages; the slow part of our attack, namely eviction, is not necessary. Iterating over the entire user address space with a 4 kB granularity (*i.e.*, 2^{35} pages) takes 4.4 h, with a test for a single address taking 465.91 ns on average (standard error of 0.20 ns) on our i7-6700K. However, in practice the entropy of ASLR is much lower, leaving us with smaller address ranges to search. The working set of every process almost always contains some code and data pages, heap pages, and stack pages as they are essential to the process execution. Windows 10 uses 24 bits of entropy for the heap, 17 to 19 bits of entropy for the executable, and up to 25 bits of entropy for the stack pages (with an additional 8 bits of displacement within the stack) [2, 29, 55]. All this means that we can skip most locations and only search through the corresponding randomized region. With 465.91 ns on average (standard error of 0.20 ns) per address check, we find that it takes 15.6 s to locate the stack, 8 s to locate the heap, and less than 1 s to locate the executable on our i7-6700K. Microsoft acknowledged the issue and issued a fix for the ASLR break.

7.3 Observing Low Frequency Events

In this section, we present the basic idea of observing system events via the page cache side channel. The information obtained is used for various attacks in the subsequent subsections. Due to the similarity to controlled-channel attacks on TEEs [81, 89], the first question is of course whether we can mount the same attacks without privileges on regular applications. However, as controlled-channel attacks artificially slow down the victim enclave, they can have an arbitrarily high timing resolution. While these attacks work equally with page cache attacks when artificially slowing down the victim, this is not a realistic attack scenario, as the attacker needs elevated privileges to slow down the victim. Page cache attacks can trivially detect running applications, which is directly reflected through the corresponding program binary's presence in the page cache. Beyond that, we focus on lower frequency events in browsers, mail clients, password safes, web servers, and other applications. There is an abundance of information leaks allowing the complete supervision of any (user) activity on the system.

For browsers, we analyzed what information can be leaked from Firefox (63.0-20181023213305 4.18.0-11-generic). We found that we can observe the opening of new browser windows (among many others: /usr/lib/firefox/firefox, pages 54 and 55), and whether a video starts playing (libmozavcodec.so, pages 48 to 63). Furthermore, we noticed differences in the side channel when playing videos from different sources: Youtube (streaming webm) uses most pages of libmozavcodec.so up to page 416 while playing a video, Dailymotion was not distinguishable from directly playing an mp3 or mp4 file only e.g., pages 48 to 56 were in use when starting a video, and Pornhub (streaming mp4) used more pages while playing a video e.g., 64 to 80 and 240 to 256. We can also deduce to some extent which type of website is currently opened: /usr/lib/firefox/libmozavutil.so leaks via pages 0

to 23 whether a media-containing website like Twitter, Facebook, or Youtube is opened in any (loaded) tab. Other, more static or non-media pages (Google, Google Maps, Github, Yahoo) do not require the corresponding code from this library. We can infer when a PNG file is rendered (across various applications) when spying on `/usr/lib/x86_64-linux-gnu/libpng16.so`. `libxul.so` is the largest shared library used by Firefox and exposes an abundance of events, such as scrolling, marking text, mouse clicks, opening links, as well as opening and closing tabs and windows. The same library is also used by Thunderbird, and there allows us to monitor when emails come in, when users click on emails, delete emails, and write emails. Future work should systematically analyze the leakage in `libxul.so` via templating [34] or machine learning techniques [35, 84, 93].

While this type of leakage is mostly a privacy issue, personal and context-aware information is also essential for extortion and phishing attacks [7, 38, 71]. In particular the information on which video site was visited for how long could be used to make extortion attacks more compelling [60]. Furthermore, UI redress attacks, as shown in the following section, can benefit from our information leaks, e.g., to open fake overlay windows at exactly the right time.

7.4 Authentication UI Redress Attack

In this section, we present a user-interface redress attack [4, 10, 25, 57, 65, 70] which relies on our side channel as a trigger. The basic idea is to detect when an interesting window is opened and to place an identically looking fake window over it. This can be so stealthy that even advanced users do not notice it [25]. However, to achieve this, the latency between the original window opening and the fake window being placed over it must be very low. Fortunately, our side channel provides us with exactly this capability, regardless of any other information leakage. Note that the operating systems authentication windows may be protected. However, other password prompts, e.g., for password managers, browsers, and mail clients, are usually unprotected and can be targeted in our attack.

We use our side channel to detect when a root authentication window on Ubuntu 16.04 is displayed. We detect this with a latency of 2.04 µs on average, and it does not take us longer to make our fake window visible and move it on top of the real window. The user now types in the root password in our fake window. Depending on the attacker capabilities, the attacker can either forward the password to the real window or simply close the fake window after the password was entered. In the latter case, the user would see the original authentication window afterwards and likely think that the password was rejected on the first try, e.g., because of a typing error occurred.

To identify binary pages which are used when spawning the root authentication window, we performed an automated template attack (cf. Section 7.5). Note that the template attack is performed on an attacker-controlled system with identical software installed. Hence, the attacker can take arbitrary means (e.g., side-channel attacks or a debugger) to find interesting memory locations that can be exploited on the victim system. The attacker first runs a debugger-based or cache-based template attack [34] to identify binary regions that handle the corresponding event. In a second run, the attacker templates with our page cache side-channel attack. In our specific case, the result of the templating was that the strongest leakage is page 2 in the binary file `polkit-gnome-authentication-agent-1`. Hence, on the victim system, the attacker simply uses the previously obtained templates to mount the attack.

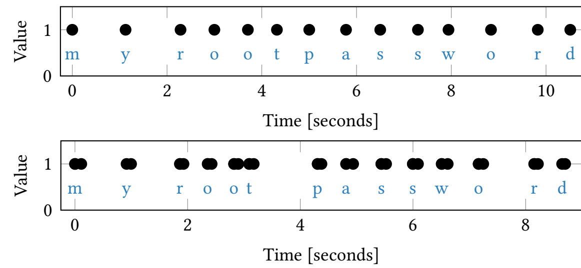

Figure 2: Values returned by the page cache side channel during a password entry on Linux (top) and while typing in an editor on Windows (bottom). On Windows we observe key up and key down events due to the page selected and the high attack frequency achievable. In both cases, there is no noise between the keystrokes.

Mounting the same attack on Windows 10 works even better. Here, the latency is below 1 µs, which is clearly not perceivable for a human. Also, unsurprisingly, we found that fake windows can be created on Windows just as on Linux. Events like authentication windows and password prompts are well suited for our attack due to the low frequency in which they occur. This also makes the automated templating for leaking pages less noisy.

7.5 Keystroke Timing Attack

In this section, we present a higher frequency inter-keystroke-timing attack [34, 56, 67, 77, 92] on keyboard input in the root authentication window on Ubuntu 18.04. To mount a keystroke timing attack, we first template [34] (cf. Section 7.4) pages that are loaded into the page cache when the user presses a key. We target the Ubuntu 18.04 authentication window, where the user types in the root password. In the template attack, we identified page 14 of `libgksu2.so.0.0.2` as a viable target page.

Figure 2 shows two attack traces when typing a password, one on Linux (Section 7.5) and one on Windows 10 (Section 7.5) in `notepad.exe`. We obtain identical traces on Windows when running the attack on Firefox. Note that on Linux, for an extremely fast typing person, we could miss some keystrokes, *i.e.*, false negatives can occur. However, we can gather these traces multiple times and combine the information from multiple traces to recover highly accurate inter-keystroke timings [56, 73]. For Windows, the temporal resolution is much higher, far below the timing variations of a human [56, 73], allowing us to reliably detect and report all inter-keystroke timings including key down and key up events.

When running the side-channel attack on an idle system for one hour, we did not observe a single false positive, neither on Windows nor on Linux. This is not surprising, if the memory region is used by unrelated events we would have already seen such noise in the template phase. However, as the attacker can and will choose the memory region based on the templating, the attacker chooses memory regions which are not really used by any unrelated events.

Thus, in the optimal case, the selected memory region is completely noise-free. In such a case, there is no functionality in the operating systems that could lead to false positives due to spurious cache hits.

7.6 PHP Password Generation

A standard approach to benchmark and evaluate novel cache side channels, is to use them in an attack on weak cryptographic implementations which are known to be insecure [34, 58, 90, 94].

The PHP `microtime` function returns the current UNIX timestamp in microseconds. It is carelessly used by some frameworks to initialize the PHP pseudo-random number generator (PRNG) before it is used in cryptographic operations or to generate temporary passwords [1, 22, 94]. This is a practice known to be insecure, and very similar code has been attacked previously to benchmark side channels [94]. We found that the popular phpMyFAQ framework [66] still relies on this approach.[1]

We mount our page cache attack on the main PHP binary (7.0.4-7ubuntu2), on the function `zif_microtime`. This function is read-only and shared with any unprivileged process including the attacker. In our case, the function resides on page `0x1b9` (441) of the binary. By monitoring this page, we can determine the return value of `microtime` at the initialization of the PRNG. Based on this, we can reconstruct any password generated based on the same PRNG initialization, as the password generation algorithm is open source.

Due to the large variance on the runtime of PHP scripts, we only detected an access to the `microtime` function with an accuracy of ±1.5 ms. However, this is practical to brute force the range of remaining possible return values. On a newer PHP version (7.0.30-0ubuntu0.16.04.1), we observed an average difference of ±2.0 ms. Thus, we have to try around 4000 different passwords in the real-world attack. We confirmed that in 85 % of the test runs, the real password of the user was among the 4000 generated passwords from the attacker. Hence, also in this scenario, our page cache side channel can compete with state-of-the-art attacks [94].

Our attack also works on Windows. However, as the main source of noise is the varying runtime of PHP, the accuracy is not measurably better on Windows.

7.7 Oracle Attacks

Our side channel also allows implementing padding- or length-oracle attacks. For instance, a password or token comparison using `strcmp` forms a length oracle. If the attacker can place the string on the page boundary, the attacker can measure at which byte of the string the comparison terminated. By manipulating the string, the attacker can figure our the correct password or token.

We verified that this attack is practical in a proof-of-concept program. The attacker passes the string through an API to the victim process. With the page-cache- or working-set-based side channel we can determine whether the second page was loaded into the page cache or added to the working set. If it was, the attacker learns that the bytes on the first page were guessed correctly.

As the attacker can fully control the frequency of the measurements here and can repeat the attack, we observed no cases where we could not successfully leak the secret.

[1] We responsibly disclosed this vulnerability to the developers of phpMyFAQ, who issued a patch following our recommendation.

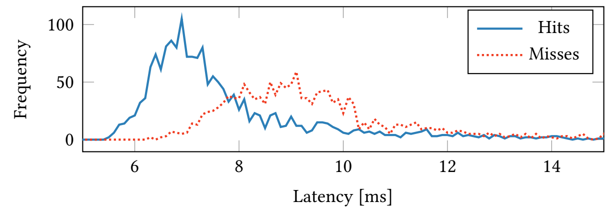

Figure 3: Timing histogram of the remote covert channel with a 100 kB file (25 pages).

7.8 Mobile Attacks

Android also provides the `mincore` function to unprivileged user applications, and we can also mount our side-channel attack on this operating system. For our evaluation, we used a Samsung Galaxy S9 running Android 8.1.0.

The difference between Android and Linux is that user-space applications are better isolated. Indeed, we cannot target user binaries, but only system libraries and system applications. By monitoring `libinputflinger.so`, we are able to detect all user inputs (e.g., touches), which can be used to mount a keystroke-timing attack (cf. Section 7.5). By monitoring `libcamera_client.so`, we are able to detect when the camera is opened, and when the user takes a photo

Besides libraries, system applications can also be targeted. We verified that we can see the startup and user interaction with system applications such as the calendar, NFC service, or the default on-screen keyboard.

8 REMOTE ATTACK

For our remote attack we have to distinguish soft pagefaults, *i.e.*, just mapping the page from the page cache, and regular pagefaults, *i.e.*, page cache misses, over a network connection. In this scenario, two physically separated processes wish to communicate with each other. The sender process runs on a server and has access to information that the attacker wants to have. However, it is unprivileged, firewalled, and possibly sandboxed, so it cannot reach any network resources or expose files for remote access. However, the server exposes multiple files to the public internet, e.g., over a web server. We also assume that the sender process has read permissions to these files, e.g., Apache has world-readable permissions on files in the web server root directory by default. The receiver process runs on a remote server, measuring the remote access latency to pages in these public files. Hence, the sender process can encode the information in the page cache state of these pages.

Page Cache Hits and Misses. Of course, a remote attacker cannot invoke `mincore` to check which pages are in cache, so the attacker needs to rely on timing. Hence, we first try to distinguish cache hits and misses over the network, similarly to the related work in [74, 80], by performing remote accesses with and without clearing the page cache. We also ensured that there was no other intermediary network caching or proxy caching active by passing appropriate HTTP headers to the server. Figure 3 shows the frequencies of remote access latencies for various cached and uncached accesses; the figure shows that cache hits can be distinguished from cache misses. Here, the mean access time was 8.4 ms for cache hits and 14.2 ms for cache misses to access a file with 25 pages (around

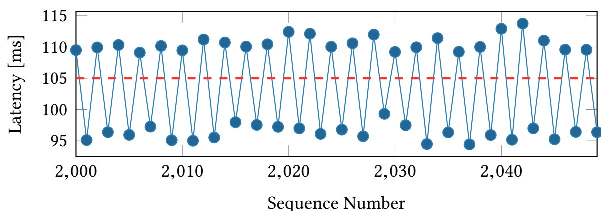

Figure 4: Transmitting a sequence of alternating '0's and '1's by accessing a 10 MB file (2560 pages). A threshold can distinguish the two cases.

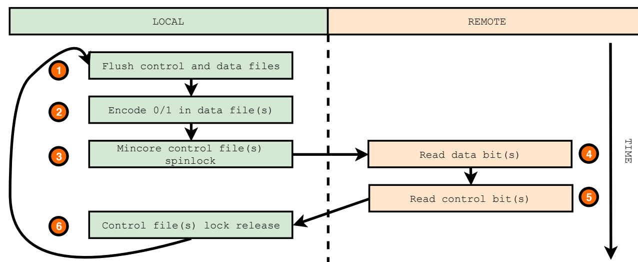

Figure 5: Illustration of the web server covert channel.

100 kB). The latency differences between cache hits and misses grow with the number of pages accessed. Hence, we use larger files for the subsequent remote attacks.

8.1 Covert Channel Protocol

Figure 5 depicts how the two processes communicate over the covert channel. The local sender process is an unprivileged (possibly sandboxed) malware that encodes secret data from the victim machine into page cache hits and misses, and the remote receiver process decoding the secret data after measuring the remote access latency. For this, the sender process uses one file to encode data, and another file for synchronization (control file). The sender process first evicts both the data and control files from the file system cache (Step 1) using `posix_fadvise` on a rarely used file, *i.e.*, a file which is not currently locked in memory by another process. Note that the attacker could also use any other means of page cache eviction as described in Section 6. It then encodes one bit of information in the data file (Step 2) by either bringing it into the page cache by reading the file (encoding a '1'), or not bringing it into the cache (encoding a '0'). After encoding, the sender waits for the control file to be read by the remote process (Step 3). For this, the sender uses `mincore` on the control file in a loop, checking how many of the file's pages are in the page cache. In our case, the sender waits until 80 % of the file are cached, indicating that the remote attacker accessed it.

The receiver process measures the access latency, inferring the bits the sender process was transmitting (Step 4). In our experiments, the access time threshold that demarcated a '0' from a '1' was set to 105 ms for our hard drive experiments, as illustrated in Figure 4.

Immediately after the receiver process accessed the data file, it also accesses the control file (Step 5), to let the sender know the next bit can be transmitted now. The sender then continues at Step 1 again. This happens until the sender has transmitted all bits of secret information.

Evaluation. Our experimental setup involved two separate, but geographically close, machines, *i.e.*, a network distance of 4 hops. The victim machine was running the Linux Mint (kernel version 4.10.0-38) on an AMD A10-6700 with 8 GB RAM and a 977 GB hard drive. The victim machine exposed two files to the network, `data.jpg` (10 MB) and `control.jpg`, used as the data and control files respectively. The remote machine was also running Linux Mint (kernel version 4.13.0-37) on an Intel Core i7-7700 with 16 GB RAM and a 219 GB SSD.

For the evaluation, we transmitted 4000 bits from the local machine to the remote machine multiple times. The transmission took 517 s on average, which corresponds to an average bit rate of 7.74 bit/s and an average bit error rate of 0.2 %. This is a higher bit rate than several other remote covert channels [11, 16, 74]. The bit rate can be further increased by encoding information through more than one file, which is realistic given the vast number of files most web servers today have. To increase stealthiness, the attacker may choose to access the two files from different IPs, as the sender process is agnostic to this.

As our covert channel relies on timing differences, we also repeated our experiments on a machine with an SSD. Distinguishing a page cache hit from a page cache miss through timing over the network, could be more difficult as the timing differences can be smaller. To overcome this, we simply use a larger image file (30 MB, 7680 pages) to amplify the timing difference. However, this meant that the load latency threshold that demarcates a read hit and miss would need to be scaled up similarly from the previous experiment, and was set to 300 ms for the experiments on SSDs. Furthermore, we reduce the geographical distance between attacker and victim to 2 network hops. The victim server was running on a machine with Linux Mint on an Intel Core i7-7700 (kernel version 4.13.0-37) with 16 GB RAM and a recent off-the-shelf 219 GB SSD, and the attacker machine was the same as before. The transmission of 4000 bits, now takes 1298 s on average, giving us an average bit rate of 3.08 bit/s at an average bit error rate of 0.35 %. Hence, this remote timing covert channel is also possible on a machine with an SSD.

Our proof-of-concept implementation could be further optimized to yield a higher transmission rate, to mount the attack over a greater geographical distance, or to use smaller files, simply by repeating measurements for each single bit [74]. In our proof-of-concept we did not repeat any measurements to obtain a single bit, again indicating the high capacity of this remote covert channel.

8.2 Remote Side Channel

Similarly to our local side-channel attacks, we could also mount remote side-channel attacks exploiting the page cache. This information could be used to determine whether certain pages or scripts have been recently accessed [80]. However, in practice it is difficult to evict the cache remotely. A web server would need to provide access to enough files to occupy the entire page cache. The attacker then would need to constantly access this set of files. Controlling the working set via a huge number of remote file accesses will make the attack very conspicuous, though it may still be practically

effective for opportunity-based attacks (e.g., password reset pages) such as those presented in Section 7.6.

9 COUNTERMEASURES

Our side-channel attack targets the operating system page cache via operating system interfaces and behavior. Hence, it clearly can be mitigated by changing the operating system implementation, but these changes can be substantial.

Privileged Access. The `QueryWorkingSetEx` and `mincore` system calls are the core of our hardware-agnostic side-channel attacks. Requiring a higher privilege level or returning fake information for these system calls mitigates these attacks, but could also break existing programs that legitimately use these system calls. While we didn't observe any `sys_enter_mincore` system calls over multiple hours of every-day computer usage, it appears to play an important role in cloud service implementations [14]. Hence, rather than making `mincore` privileged, we recommend changing what information it returns or restricting the mapping types for which it works. This would mitigate our basic hardware-agnostic attacks (as acknowledged in our responsible disclosure). However, there are other interfaces which provide equivalent information, such as the `preadv2` system call with certain flags or accessing files opened with the `O_DIRECT` flag [15]. Furthermore, timing-based attacks remain unaffected, indicating that a generic mitigation of page cache attacks on Linux may require a fundamental re-design of the page cache.

On Windows, there are multiple possible solutions to mitigate our hardware-agnostic attacks by adapting the privileges required for the system calls we use. First of all, a process should not be able to obtain working-set information of another process via `QueryWorkingSetEx`, in particular not for processes with a higher integrity level, especially, as this contradicts the official documentation [53]. Second, the share count information should be omitted from the struct returned by `QueryWorkingSetEx` as it exposes information about other processes to the attacker. The combination of these two changes mitigates our hardware-agnostic attack variants on Windows. Again, timing-based attacks are still possible and mitigating them would require more fundamental changes.

We responsibly disclosed our attacks, which have in response been addressed in Linux and Windows, and was assigned CVE-2019-5489. For `QueryWorkingSetEx`, instead of `PROCESS_QUERY_LIMITED_INFORMATION`, Windows will require `PROCESS_QUERY_INFORMATION` to prevent lesser privileged processes from directly obtaining working set information, as well as omitting the share count information, to prevent indirect observations on working set changes in other processes.

It was also surprising that Windows allows changing the working-set size for another process. If this would be restricted, it would be more difficult to reliably evict across processes. The performance of our covert channel would decrease if `VirtualUnlock` did not have the "feature" that it removes pages from the working set of other processes if they are not locked.

Alternative approaches like page locking, signal burying, or disabling page sharing are likely not practical for most use cases or impose significant overheads.

Preventing Efficient Eviction while Increasing the System Performance. On Windows, we used working set eviction instead of page cache eviction as on Linux. We verified that the approach we used on Linux, *i.e.*, page cache eviction, also works on Windows. However, it performs worse than on Linux and optimizing the eviction is left as a challenge for future work. The reason for this lies in the fact that Linux uses a global page replacement algorithm, *i.e.*, an algorithm which does not distinguish between different processes. Global page replacement algorithms have been known for decades to allow one process to perform a denial-of-service on other processes [9, 19, 20, 69].

However, even working-set algorithms like those used by Windows [69] do not perfectly mitigate our attack. In the worst case, an attacker can always resort to evicting the entire cache and measuring the time to fault the page back in. A complete mitigation would need to eliminate either of these two attacker capabilities: bringing the cache into a known state, and measuring whether the state has changed.

10 CONCLUSION

We have demonstrated that the page cache in modern operating systems can be exploited in side-channel attacks. In hardware-agnostic attacks, we have demonstrated a high-speed cross-sandbox covert channel, an ASLR break on Windows 10, a UI redressing attack triggered by a side channel, a keystroke-timing side channel, and password-recovery side channel from a vulnerable PHP script. Even after applying the security changes we recommend, an attacker can resort to timing-based attacks. We demonstrate this with a covert channel exfiltrating data from a local malicious sender to a remote receiver. The severity of this attack surface is exacerbated by the variety of isolation techniques that share the page cache, including regular Unix processes, sandboxes, Function-as-a-Service platforms, managed language runtimes, web browsers, and even select remote processes. Indeed, mitigating this core problem appears to require a fundamental redesign of the functionality of the page cache.

ACKNOWLEDGMENTS

We want to thank James Forshaw, Dmitrii Kuvaiskii, and Jon Masters for helpful comments and discussions and Simon Gunacker for early exploratory work on this topic. Ari Trachtenberg and Trishita Tiwari were supported, in part, by the National Science Foundation under Grant No. CCF-1563753 and Boston University's Distinguished Summer Research Fellowship, Undergraduate Research Opportunities Program, and the department of Electrical and Computer Engineering. Daniel Gruss and Michael Schwarz were supported by a generous gift from ARM, and also by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] George Argyros and Aggelos Kiayias. 2012. I Forgot Your Password: Randomness Attacks Against PHP Applications.. In *USENIX Security Symposium.*

[2] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R. Gross. 2015. CAIN: Silently Breaking ASLR in the Cloud. In *WOOT.*

[3] Daniel J. Bernstein. 2004. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] Antonio Bianchi, Jacopo Corbetta, Luca Invernizzi, Yanick Fratantonio, Christopher Kruegel, and Giovanni Vigna. 2015. What the app is that? deception and countermeasures in the android user interface. In *S&P*.
[5] Andrew Bortz and Dan Boneh. 2007. Exposing private information by timing web applications. In *WWW*.
[6] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2016. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P*.
[7] Garrett Brown, Travis Howe, Micheal Ihbe, Atul Prakash, and Kevin Borders. 2008. Social networks and context-aware spam. In *ACM conference on Computer supported cooperative work*.
[8] Luigi Bruno. 2013. What page replacement algorithms does the windows 7 OS uses? https://social.technet.microsoft.com/Forums/ie/en-US/e61aef24-38fd-4e7e-a4c1-a50aa226818c
[9] Richard W Carr and John L Hennessy. 1981. WSCLOCK-a simple and effective algorithm for virtual memory management. *ACM SIGOPS Operating Systems Review* 15, 5 (1981), 87–95.
[10] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. 2014. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks.. In *USENIX Security Symposium*.
[11] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The last mile: An empirical study of timing channels on seL4. In *CCS*.
[12] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Technical Report. Massachusetts Institute of Technology, Cambridge.
[13] Jonathan Corbet. 2005. A CLOCK-Pro page replacement implementation. https://lwn.net/Articles/147879/
[14] Jonathan Corbet. 2019. Defending against page-cache attacks. https://lwn.net/Articles/776801/
[15] Jonathan Corbet. 2019. Fixing page-cache side channels, second attempt. https://lwn.net/Articles/778437/
[16] Scott A Crosby, Dan S Wallach, and Rudolf H Riedi. 2009. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)* 12, 3 (2009), 17.
[17] Doug Crowthers. 2011. Can You Get More Space Or Speed From Your SSD? https://www.tomshardware.com/reviews/ssd-performance-tweak,2911-4.html
[18] Debian Wiki Team 2018. SSDOptimization. https://wiki.debian.org/SSDOptimization
[19] Peter J Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (1968), 323–333.
[20] Peter J. Denning. 1980. Working sets past and present. *IEEE Transactions on Software Engineering* 1 (1980), 64–84.
[21] Borislav Djordjević, Nemanja Maček, and Valentina Timčenko. 2015. Performance Issues in Cloud Computing: KVM Hypervisor's Cache Modes Evaluation. *Acta Polytechnica Hungarica* 12, 4 (2015), 147–165.
[22] Stefan Esser. 2008. Lesser known security problems in PHP applications. In *Zend Conference*.
[23] Edward W Felten and Michael A Schneider. 2000. Timing attacks on web privacy. In *CCS*.
[24] Firejail 2018. Firejail Security Sandbox. https://firejail.wordpress.com/
[25] Yanick Fratantonio, Chenxiong Qian, Simon P Chung, and Wenke Lee. 2017. Cloak and dagger: from two permissions to complete control of the UI feedback loop. In *S&P*.
[26] Mark B Friedman. 1999. Windows NT page replacement policies. In *Int. CMG Conference*.
[27] Mel Gorman. 2004. *Understanding the Linux virtual memory manager.* Prentice Hall Upper Saddle River.
[28] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium*.
[29] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS*.
[30] Daniel Gruss, David Bidner, and Stefan Mangard. 2015. Practical Memory Deduplication Attacks in Sandboxed JavaScript. In *ESORICS*.
[31] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another Flip in the Wall of Rowhammer Defenses. In *S&P*.
[32] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*.
[33] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
[34] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.
[35] Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2017. Cache-based application detection in the cloud using machine learning. In *AsiaCCS*.
[36] Vidar Holen. 2017. Experiments and fun with the Linux disk cache. https://www.linuxatemyram.com/play.html
[37] Christian Horn. 2017. Do we really need swap on modern systems? https://www.redhat.com/en/blog/do-we-really-need-swap-modern-systems
[38] Markus Huber, Martin Mulazzani, Edgar Weippl, Gerhard Kitzler, and Sigrun Goluch. 2011. Friend-in-the-middle attacks: Exploiting social networking sites for spam. *IEEE Internet Computing* 15, 3 (2011), 28–34.
[39] Mehmet S Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES (LNCS)*, Vol. 9813. Springer, 368–388.
[40] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *AsiaCCS*.
[41] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2015. Lucky 13 Strikes Back. In *AsiaCCS*.
[42] Collin Jackson, Andrew Bortz, Dan Boneh, and John C Mitchell. 2006. Protecting browser state from web privacy attacks. In *WWW*.
[43] Yaoqi Jia, Xinshu Dong, Zhenkai Liang, and Prateek Saxena. 2015. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing* 19, 1 (2015), 44–53.
[44] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement.. In *USENIX ATC*.
[45] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.
[46] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.
[47] Butler W Lampson. 1973. A note on the confinement problem. *Commun. ACM* 16, 10 (1973), 613–615.
[48] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*.
[49] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.
[50] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *DIMVA*.
[51] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*.
[52] Microsoft. 2017. Cache and Memory Manager Improvements. https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server
[53] Microsoft. 2018. Programming reference for Windows API. https://docs.microsoft.com/en-us/windows/desktop/api/index
[54] Microsoft. 2018. Windows desktop applications. https://msdn.microsoft.com/en-us/library/windows/desktop/aa906039.aspx
[55] Microsoft. 2018. x64 stack usage. https://docs.microsoft.com/en-us/cpp/build/stack-usage?view=vs-2017
[56] John Monaco. 2018. SoK: Keylogging Side Channels. In *S&P*.
[57] Marcus Niemietz and Jörg Schwenk. 2012. UI Redressing Attacks on Android Devices. *Black Hat Abu Dhabi* (2012).
[58] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.
[59] Rodney Owens and Weichao Wang. 2011. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *International Performance Computing and Communications Conference*.
[60] Mathias Payer. 2019. https://twitter.com/gannimo/status/1086185299225571328
[61] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.
[62] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*.
[63] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. 2016. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security Symposium*.
[64] Red Hat. 2017. *Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide*.
[65] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. 2015. Towards Discovering and Understanding Task Hijacking in Android. In *USENIX Security Symposium*.
[66] Thorsten Rinne. 2018. phpMyFAQ - Open Source FAQ system for PHP and MySQL, PostgreSQL and other databases. https://github.com/thorsten/phpMyFAQ
[67] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*.
[68] Mark Russinovich. 1998. Inside Memory Management, Part 2. https://www.itprotoday.com/management-mobility/inside-memory-management-part-2

[69] Mark E Russinovich, David A Solomon, and Alex Ionescu. 2012. *Windows internals*. Pearson Education.

[70] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. 2010. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *4th USENIX Conference on Offensive Technologies*.

[71] Fernando Sanchez, Zhenhai Duan, and Yingfei Dong. 2010. Understanding forgery properties of spam delivery paths. In *Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference (CEAS)*.

[72] Patrick Schmid. 2007. Windows Vista's SuperFetch and ReadyBoost Analyzed. https://www.tomshardware.com/reviews/windows-vista-superfetch-and-readyboostanalyzed,1532-2.html

[73] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. 2018. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*.

[74] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2018. NetSpectre: Read Arbitrary Memory over Network. *arXiv:1807.10535* (2018).

[75] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings*.

[76] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *CCS*.

[77] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. 2001. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*.

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

[79] Billy Tallis. 2018. Best SSDs: Q2 2018. https://www.anandtech.com/show/9799/best-ssds

[80] Trishita Tiwari and Ari Trachtenberg. 2018. POSTER: Cashing in on the File-System Cache. In *ACM SIGSAC Conference on Computer and Communications Security*.

[81] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. Sgx-step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution*.

[82] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS*.

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

[84] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. 2019. Unveiling your keystrokes: A Cache-based Side-channel Attack on Graphics Libraries. In *NDSS*.

[85] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. 2017. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*.

[86] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud. *IEEE/ACM Transactions on Networking* (2014).

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

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

[89] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*.

[90] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[91] J Young. 2002. NSA tempest documents. *CRYPTOME* (2002).

[92] Kehuan Zhang and XiaoFeng Wang. 2009. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium*.

[93] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2018. Analyzing Cache Side Channels Using Deep Neural Networks. In *ACSAC*.

[94] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS*.