Page Cache Attacks

Daniel Gruss[1], Erik Kraft[1], Trishita Tiwari[2], Michael Schwarz[1],
Ari Trachtenberg[2], Jason Hennessey[3], Alex Ionescu[4], Anders Fogh[5]
[1] *Graz University of Technology,* [2] *Boston University,* [3] *NetApp,* [4] *CrowdStrike,* [5] *Intel Corporation*

Abstract

We present a new hardware-agnostic 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, and our attacks thus work across cores and CPUs. Our side-channel permits unprivileged monitoring of some memory accesses of other processes, with a spatial resolution of 4 kB and a temporal resolution of 2 µs on Linux (restricted to 6.7 measurements per second) and 466 ns on Windows (restricted to 223 measurements per second); this is roughly the same order of magnitude as the current state-of-the-art cache attacks. We systematically analyze our side channel by demonstrating different local attacks, including a sandbox bypassing high-speed covert channel, timed user-interface redressing attacks, and an attack recovering automatically generated temporary passwords. We further show that we can trade off the side channel's hardware agnostic property for remote exploitability. We demonstrate this via a low profile remote covert channel that uses this page-cache side-channel to exfiltrate information from a malicious sender process through innocuous server requests. Finally, we propose mitigations for some of our attacks, which have been acknowledged by operating system vendors and slated for future security patches.

1 Introduction

Modern processors are highly optimized for performance and efficiency. A large share of these optimizations is based upon *caching* - taking advantage of temporal and spatial locality to minimize slower memory or disk accesses. Indeed, caching architectures typically fetch or prefetch code and data into fast buffers closer to the processor.

Although side-channels have been known and utilized primarily in military contexts for decades [41, 78], the idea of cache side-channel attacks gained more attention over the last twenty years [3, 40, 53]. Osvik et al. [51] showed that an attacker can observe the cache state at the granularity of a cache set using Prime+Probe, and later Yarom et al. [77] showed this with cache line granularity using Flush+Reload. While different cache attacks have different use cases, the accuracy of Flush+Reload remains unrivaled.

Indeed, virtually all Flush+Reload attacks target pages in the so-called page cache [30, 33, 34, 35, 42, 77]. The page cache is a pure software cache implemented in all major operating systems today, and it contains virtually all pages in use. 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 [24]. The operating system uses the page cache to store frequently used pages in memory, this obviating slow disk loads whenever a process needs to access said pages. There is a large body of works exploiting Flush+Reload in various scenarios over the past several years [30, 33, 34, 35, 42, 77]. There have also been a series of software (side-channel) cache attacks in the literature, including attacks on the browser cache [5, 20, 36, 37, 72] and exploiting page deduplication [2, 6, 26, 52, 55, 68, 75, 76]; however, page deduplication is mostly disabled or limited to deduplication within a security domain today [46, 56, 71].

In this paper, we present a new attack on the operating system page cache. We present a set of local attacks that work entirely without any timers, utilizing operating system calls (`mincore` on Linux 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 the software-cache side channel by comparing it to hardware-cache side channels. Like the recent DRAMA attack [54, 73], our side-channel attack works across cores and across CPUs with a spatial granularity of 4 kB. For comparison, the spatial granularity of the DRAMA attack is 2 kB on dual-channel systems up to and including the Haswell processor architecture, and 1 kB on more recent dual-channel systems. The temporal granularity of the DRAMA attack is around 300 ns, whereas the temporal granularity of our attack is 2 µs on Linux (restricted to 6.7 measurements per second) and 466 ns on Windows (restricted to 223 measurements per second). Hence, 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 informed us that they will follow our recommendations with security patches to mitigate our attack.

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 which is agnostic to specific hardware configurations.
3. We present a set of local attacks which can compete with state-of-the-art microarchitectural attacks.
4. We present 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 and Windows. Section 7 presents (timing-free) local page cache attacks. Section 8 presents 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 Side-Channel Attacks

The suggestion of cache attacks harks back to the timing attacks of Kocher [40]. Osvik et al. [51] presented a technique with a finer granularity called Prime+Probe. Yarom et al. [77] 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 [39] and Meltdown [43]. 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, 20, 36, 37, 72]. 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. [68] presented the first page-deduplication attack, which detected programs running in co-located virtual machines. Subsequently, several other page-deduplication attacks were demonstrated [26, 52, 75, 76]. Today, page deduplication is either completely disabled for security reasons or restricted to deduplication within a security domain [6, 46, 55, 56].

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.

Indeed, 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, and OS X) implement a page cache. The page cache contains 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 [24]. 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, allowing the operating system to minimize 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 the Least Recently Used (LRU) paradigm [11], and LRU-related data structures can still be found throughout the kernel code. More recent Linux versions implement an improved variant called CLOCK-Pro [38] along with several adaptions [12]. 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 [8, 16, 17]. The page replacement algorithm used on Windows was based on Clock or pseudo-random replacement [23, 60] in older Windows versions, and today is likely a variant of the Aging algorithm [7].

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 [14, 15, 32]). 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 [18].

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

3 Threat Model

Our threat model is based on the threat model for Flush+Reload [30, 33, 34, 35, 42, 77].

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 shared files (e.g., Firejail [21]).

On Windows, read access to the target page is not necessary for our attack.

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.

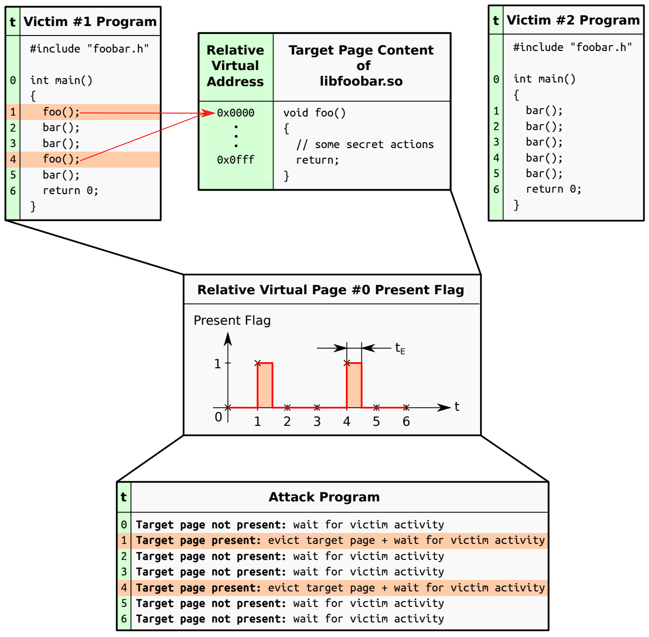
4 High-Level View of the Attack

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 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 [28, 66, 71] or consume too much time and impose system pressure [27]. 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 a working-set-based eviction strategy that efficiently accesses groups of other pages more frequently than the page to evict.

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 [47], 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



```
t  Victim #1 Program
     #include "foobar.h"

0    int main()
     {
1      foo();
2      bar();
3      bar();
4      foo();
5      bar();
6      return 0;
     }
```

```
Relative Virtual Address   Target Page Content of libfoobar.so
0x0000                     void foo()
   :                       {
   :                         // some secret actions
0x0fff                       return;
                           }
```

```
t  Victim #2 Program
     #include "foobar.h"

0    int main()
     {
1      bar();
2      bar();
3      bar();
4      bar();
5      bar();
6      return 0;
     }
```

```
t  Attack Program
0  Target page not present: wait for victim activity
1  Target page present: evict target page + wait for victim activity
2  Target page not present: wait for victim activity
3  Target page not present: wait for victim activity
4  Target page present: evict target page + wait for victim activity
5  Target page not present: wait for victim activity
6  Target page not present: wait for victim activity
```

Figure 1: Attack overview.

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 [51, 77] and page-deduplication attacks [52, 68]. 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 [47] (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 from a shared library is loaded into the page cache, as this is exactly the time of the access by the victim program. Thus, the shared library containing the target page an attacker wants to observe accesses to has to be mapped into the attacker's address space. This is possible using `mmap` on Linux and either `LoadLibraryEx` or `CreateFileMappingA` and `MapViewOfFile` on Windows.

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.

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 which are discussed as follows.

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. In contrast to the official documentation [47], 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 and even for some processes with a higher integrity level (as part of the generic execute access) [48]. 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. 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, *i.e.*, the attacking process must have the `PROCESS_QUERY_LIMITED_INFORMATION` permission for the victim process handle. As `QueryWorkingSetEx` takes virtual addresses, we need to figure out the virtual address. This is not a problem if pages from shared files are targeted (e.g., shared libraries) as they are typically mapped to the same virtual address in different processes. However, if the pages are not shared, *i.e.*, not attacker-readable, `QueryWorkingSetEx` still leaks information if the virtual address is known. Hence, we can use `QueryWorkingSetEx` to 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 a recent attack on TLB entries [25] and similar to the DRAMA attack [54, 73] on a single-channel DDR3 system, which has the same spatial granularity (one 4 kB page). The spatial granularity of the DRAMA attack increases with the number of banks, ranks, channels, and processors. It is 2 kB on dual-channel systems up to Haswell, and 1 kB on more recent dual-channel systems. If a target region contains other frequently used data, the signal-to-noise ratio decreases in our attack just as it does for the DRAMA attack. 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 [54, 73]. 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. 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 Alternatives to `mincore` and `QueryWorkingSetEx`

As an alternative to `mincore` on Linux, we also investigated whether it is possible to mount the same attack using `procfs` information, namely `/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 [66], access to `/proc/self/pagemap` was first restricted and nowadays it is often not accessible by unprivileged processes.

As a more generic alternative to `mincore` and `QueryWorkingSetEx`, we investigated the timing of pagefaults as another source of information. Accessing a page may trigger a pagefault. Measuring the time it takes to handle the pagefault reveals whether it was a soft pagefault, mapping a page already present in the page cache, or a regular pagefault, loading data from the disk. The timing differences we observed there are easy to distinguish, with 1 to 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 [27, 31]. Holen et al. [31] 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. [27] 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 [45, 54, 77].

Hence, we can conclude that page cache eviction, as done in previous work, is 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).

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. Hence, our idea is to mostly access pages which are already in the page cache (to keep them there) and also access a few non-cached pages in order 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 all available to an unprivileged attacker using system calls like `mmap` and `mincore`. The attacker creates a long list of all pages currently in the page cache. The attacker also creates a list of further pages that could be loaded into the page cache to increase memory pressure. Both lists can be updated occasionally to reflect changes in the system memory use. The attacker adapts the amount of pages accessed in these two lists to achieve efficient cache eviction.

This is done by creating 3 eviction sets:

Eviction Set 1. These are pages already in the page cache, used by other processes. To keep them in the page cache, a thread continuously accesses these pages while also keeping the system load low by using `sched_yield` and `sleep`. Consequently, they are among the most recently accessed pages of the system and eviction of these pages becomes highly unlikely.

Eviction Set 2. These are pages not yet in the page cache. Using `mincore`, we can check whether the target page was evicted, and stop the eviction immediately, reducing the eviction runtime. 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. If swapping is disabled, we use another eviction set, namely non-evictable pages, e.g., dynamic content. These pages are only created and filled with content, but never again read or written. As they cannot be swapped, 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 previous 2 eviction sets.

Alternative Approaches and Optimizations. We investigated whether the file system influences the attack performance. For our tests, we used `ext4` as a file system. We compared the attack performance by running our attack on `XFS` and `ReiserFS`. However, we only found negligible timing differences.

We also investigated whether the use of the `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 was 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 rate at which events can be observed in practice is lower. The reason is that, if the event occurs, eviction is necessary, and thus, the temporal resolution for events with a higher frequency is limited to 149 ms on average. This still allows capturing more than 6 keystrokes per second, enough to capture keystrokes accurately for most users [64]. In this case, the temporal resolution of the DRAMA attack is 6 orders of magnitude higher [54, 73].

The temporal resolution is also 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 [26]. 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 must also 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 [30, 33, 34, 35, 42, 77]. 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 [45]. 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 [46, 56, 71]. On Windows, we do not have this limitation, *i.e.*, we can also attack dynamically allocated memory on Windows.

Due to the nature of the exploited side channel, our attack comes with clear limitations. 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 [30, 77].

Another limitation which frequently poses a problem in hardware cache attacks is prefetching [30, 77]. 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. [30] noted that this limits their attack to a small number of memory locations per page. The same limitations apply to our work, *i.e.*, monitoring multiple pages within a 32-page window can be noisy. However, we found that this still leaves a multitude of viable attack targets. 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 [45].

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, but where a temporal resolution of 2 µs can leak a sufficient amount of information. Furthermore, our ideal target resides on a page which is mostly accessed for this function or data block, and not for unrelated functions or data.

6.2 Process Working-Set Eviction on Windows

As previous page cache eviction techniques [27, 31] are too slow to mount generic side-channel attacks, we pursue a different approach on Windows. Windows has per-process working sets [47], which (by default) are constrained to a size between 100 kB and 1.4 MB [47]. 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 [48]. 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 [48]. 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 [47]. 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` on the other process [47]. 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 [54, 73]. The eviction on Windows via `VirtualUnlock` consumes 4.48 ms, limiting the temporal resolution for high-frequency events to 4.48 ms on average. Thus, locally the temporal resolution of the DRAMA attack is 4 orders of magnitude higher than the temporal resolution of the DRAMA attack [54, 73]. Again, this is fast enough for inter-keystroke timing attacks [49, 64].

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 [63]. 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 [49, 64].

Limitations. Our attack on Windows has clear limitations, mainly introduced by the permissions required by the attacker. More specifically, the `SetProcessWorkingSetSize` system call requires the `PROCESS_SET_QUOTA` permission on the process handle [47]. 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. Processes with a higher integrity level, e.g., processes running with Administrator privileges, cannot be attacked using this system call [48]. The `VirtualUnlock` only works on our own process and requires no permissions. Also, noise is again a limitation, which exists for both the Linux and the Windows variant of our attack, but this is again also true for any other cache side-channel attack [30, 45, 77]. In our tests, we were always able to reliably evict the page from the victim's working set indicating very low error rates.

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 [69]). 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 [14, 15, 32]. Disabling swapping allows for a better comparison with related work which also focuses on such recent systems [34, 42, 54, 73].

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 Firejail sandboxes [21]. 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 [29, 44, 45, 54]. 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 [45].

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 also evaluated the covert channel in a cross-sandbox scenario using Firejail [21]. Firejail was configured to prevent all outgoing inter-process communication, deny all network traffic, and only allow read access to the file system. We did not observe any influence from running the covert channel in isolated Firejail sandboxes. This is not specific to Firejail but works identically on other sandbox and container solutions that utilize the host system page cache, e.g., 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. [54] reported an error rate of 0.4 % for their DRAMA covert channel, albeit with a channel capacity of 74.5 kB/s which is much slower than our Windows-based covert channel, but faster than our Linux-based covert channel. Wu et al. [74] presented a cross-CPU covert channel which achieves 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 [29]. 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 Authentication UI Redress Attack

In this section, we present a user-interface redress attack [4, 9, 22, 50, 57, 62] 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 [22]. 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 7.3). 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 [30] 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.

Mounting the same attack on Windows 10 works even better. Here, the latency is only 465.91 ns, 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 very 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.

7.3 Keystroke Timing Attack

In this section, we present an inter-keystroke-timing attack [30, 49, 59, 67, 79] on keyboard input in the root authentication window on Ubuntu 18.04. To mount a keystroke timing attack, we first identify pages that are loaded into the page cache when the user presses a key using a template attack [30] (cf. Section 7.2). 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 of a password entry, one on Linux (Section 7.3) and one on Windows 10 (Section 7.3) 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 [49, 64]. For Windows, the temporal resolution is much higher, far below the timing variations of a human [49, 64], allowing us to reliably detect and report

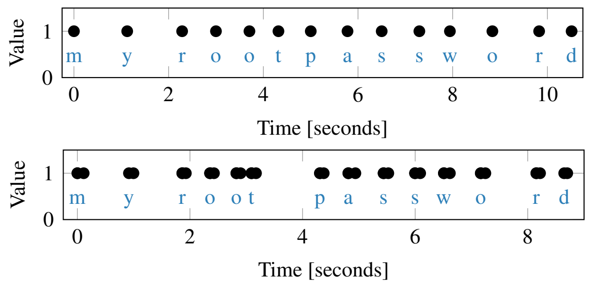


Figure 2: Values returned by the page cache side channel during a password entry on Linux (top) and while typing in an editor on Windows (bottom). On Windows we observe key up and key down events due to the page selected and the high attack frequency achievable. In both cases, there is no noise between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

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. Running the attacker binary inside a Firejail sandbox [21] had no measurable influence on the accuracy of the attack.

7.4 PHP Password Generation

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, 19, 80]. This is known as a bad practice and considered insecure, not least due to side-channel attacks [80]. During our research we found that the popular phpMyFAQ framework [58] 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 ini-

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

tialization of the PRNG. Based on this, we can reconstruct any password generated based on the same PRNG initialization, as the password generation algorithm is also publicly available.

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 [80].

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.

7.5 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 small proof-of-concept program. The attacker passes the string through an API to the victim process. By using our 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 this was the case, 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.

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

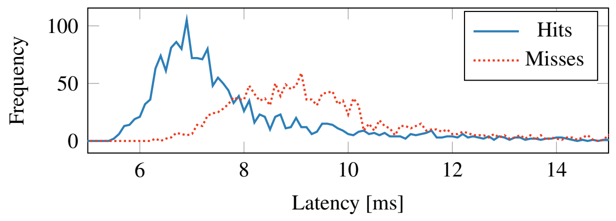


Figure 3: Timing histogram of the remote covert channel with a 100 kB file (25 pages).

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 [65, 70], 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 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

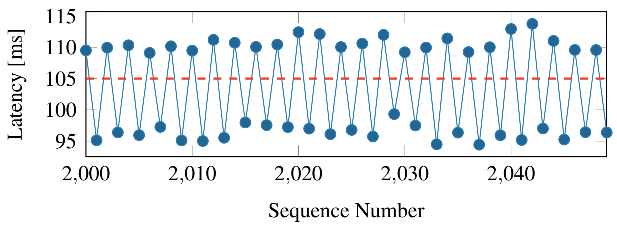


Figure 4: Transmitting a sequence of alternating '0's and '1's by accessing a 10 MB **file (2560 pages). A threshold can distinguish the two cases.**

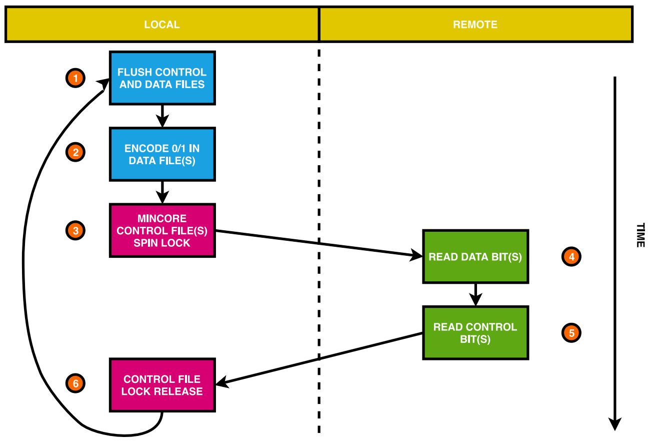

LOCAL
REMOTE
1
FLUSH CONTROL AND DATA FILES
2
ENCODE 0/1 IN DATA FILE(S)
3
MINCORE CONTROL FILE(S) SPIN LOCK
READ DATA BIT(S)
4
READ CONTROL BIT(S)
5
6
CONTROL FILE LOCK RELEASE
TIME

Figure 5: Illustration of the web server covert channel.

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 trying to transmit (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 [10, 13, 65]. 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 [65]. 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 [70]. However, in practice it is difficult to evict the cache remotely and eviction can be tricky without the information from the local system. Furthermore, 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 7.4.

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 modifying the operating system implementation.

Privileged Access. The `QueryWorkingSetEx` and `mincore` system calls are the core of our side-channel attack. Requiring a higher privilege level for these system calls stops our attack. The downside of restricting access to these system calls is that existing programs which currently make use of these system calls might break. Hence, we analyzed how frequently `mincore` is called by any of the software running on a typical Linux installation. We used the Linux `perf` tools to measure over a 5 hour period whenever the `sys_enter_mincore` system call is called by any application.[2] During these 5 hours a user performed regular operations on the system, *i.e.*, running various work-related tools like Libre Office, gcc, Clion, Thunderbird, Firefox, Nautilus, and Evince, but also non-work-related tools like Spotify. The system was also running regular background tasks during this time frame. Surprisingly, the `sys_enter_mincore` system call was not called a single time. This indicates that making the `mincore` system call privileged is feasible and would mitigate our attack at a very low implementation cost.

On Windows, there are multiple possible solutions to mitigate our attacks by adapting the privileges required for the system calls we use. First of all, it is questionable why a process can obtain working-set information of another process via `QueryWorkingSetEx`. Especially, as this contradicts the official documentation [47]. Second, the share count information could 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 all our attack variants on Windows.

We responsibly disclosed our findings to Microsoft, and they acknowledged the problem and will roll out these changes in Windows 10 19H1. Specifically, Windows will require `PROCESS_QUERY_INFORMATION` for `QueryWorkingSetEx` instead of `PROCESS_QUERY_LIMITED_INFORMATION` to prevent lesser privileged processes from directly obtaining working set information. Microsoft also follows our second recommendation of 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 much 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 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 much worse than on Linux and optimizing the eviction appeared to be far more tricky. One reason for this is that with working-set-based algorithms, processes cannot directly influence the eviction probability for pages owned by or shared with other processes [8, 16, 17]. On Linux, we are only able to evict pages efficiently because we can trick the page replacement algorithm into believing our target page would be the best choice for eviction. 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 [8, 16, 17, 61].

Working-set algorithms deplete these denial-of-service situations and they also increase the general system performance by making more clever choices for eviction candidates [8, 16, 17]. Hence, switching to working-set algorithms on Linux, as on Windows [61], makes our attack less practical. We can also transfer this insight to hardware caches: If hardware caches would use replacement algorithms that guarantee fairness in a similar way, attacks like Prime+Probe would not be possible anymore, because the attacker would rather evict its own cache lines, rather than the one required by the victim process. This is a larger change, but it might make remote attacks that rely on page cache eviction less practical.

10 Conclusion

We have demonstrated a variety of local and remote attacks against the page cache used in modern operating systems, thereby highlighting a new source for side- and covert channels that is hardware and timing agnostic. On the local front, we have demonstrated a high-speed cross-sandbox covert channel, a UI redressing attack triggered by a side channel, a keystroke-timing side channel, and password-recovery side channel from a vulnerable PHP script. On the remote front, we have shown that forgoing hardware agnosticism permits a low profile covert channel from a local malicious sender, and a higher profile side channel. 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 pro-

[2]We use `sudo perf stat -e 'syscalls:sys_enter_mincore' -a sleep 18000` for this purpose.

cesses. Stronger permissioning, as we recommend, will help against some of our local attacks.

Acknowledgments

We want to thank James Forshaw for helpful discussions on COM use cases and Simon Gunacker for early explorative work on this topic. Daniel Gruss and Michael Schwarz were supported by a generous gift from ARM and also by a generous gift from Intel. 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. 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] ARGYROS, G., AND KIAYIAS, A. I forgot your password: Randomness attacks against php applications. In *USENIX Security Symposium* (2012).

[2] BARRESI, A., RAZAVI, K., PAYER, M., AND GROSS, T. R. CAIN: silently breaking ASLR in the cloud. In *WOOT'15* (2015).

[3] BERNSTEIN, D. J. Cache-Timing Attacks on AES, `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf` 2004.

[4] BIANCHI, A., CORBETTA, J., INVERNIZZI, L., FRATANTONIO, Y., KRUEGEL, C., AND VIGNA, G. What the app is that? deception and countermeasures in the android user interface. In *S&P* (2015).

[5] BORTZ, A., AND BONEH, D. Exposing private information by timing web applications. In *WWW* (2007).

[6] BOSMAN, E., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *S&P* (2016).

[7] BRUNO, L. What page replacement algorithms does the windows 7 os uses?, `https://social.technet.microsoft.com/Forums/ie/en-US/e61aef24-38fd-4e7e-a4c1-a50aa226818c` Feb. 2013.

[8] CARR, R. W., AND HENNESSY, J. L. Wsclock-a simple and effective algorithm for virtual memory management. *ACM SIGOPS Operating Systems Review 15*, 5 (1981), 87–95.

[9] CHEN, Q. A., QIAN, Z., AND MAO, Z. M. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *USENIX Security Symposium* (2014).

[10] COCK, D., GE, Q., MURRAY, T., AND HEISER, G. The last mile: An empirical study of timing channels on sel4. In *CCS* (2014).

[11] CORBATO, F. J. A paging experiment with the multics system. Tech. rep., Massachusetts Institute of Technology, Cambridge, 1968.

[12] CORBET, J. A clock-pro page replacement implementation, `https://lwn.net/Articles/147879/` Aug. 2005.

[13] CROSBY, S. A., WALLACH, D. S., AND RIEDI, R. H. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC) 12*, 3 (2009), 17.

[14] CROWTHERS, D. Can you get more space or speed from your ssd?, `https://www.tomshardware.com/reviews/ssd-performance-tweak,2911-4.html` June 2011.

[15] Ssdoptimization, `https://wiki.debian.org/SSDOptimization` Mar. 2018.

[16] DENNING, P. J. The working set model for program behavior. *Communications of the ACM 11*, 5 (1968), 323–333.

[17] DENNING, P. J. Working sets past and present. *IEEE Transactions on Software Engineering*, 1 (1980), 64–84.

[18] DJORDJEVIĆ, B., MAČEK, N., AND TIMČENKO, V. Performance issues in cloud computing: Kvm hypervisor's cache modes evaluation. *Acta Polytechnica Hungarica 12*, 4 (2015), 147–165.

[19] ESSER, S. Lesser known security problems in php applications. In *Zend Conference* (2008).

[20] FELTEN, E. W., AND SCHNEIDER, M. A. Timing attacks on web privacy. In *CCS* (2000).

[21] Firejail security sandbox, `https://firejail.wordpress.com/` 2018.

[22] FRATANTONIO, Y., QIAN, C., CHUNG, S. P., AND LEE, W. Cloak and dagger: from two permissions to complete control of the ui feedback loop. In *S&P* (2017).

[23] FRIEDMAN, M. B. Windows nt page replacement policies. In *Int. CMG Conference* (1999).

[24] GORMAN, M. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[25] GRAS, B., RAZAVI, K., BOS, H., AND GIUFFRIDA, C. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium* (2018).

[26] GRUSS, D., BIDNER, D., AND MANGARD, S. Practical memory deduplication attacks in sandboxed javascript. In *ESORICS* (2015).

[27] GRUSS, D., LIPP, M., SCHWARZ, M., GENKIN, D., JUFFINGER, J., O'CONNELL, S., SCHOECHL, W., AND YAROM, Y. Another Flip in the Wall of Rowhammer Defenses. In *S&P* (2018).

[28] GRUSS, D., MAURICE, C., AND MANGARD, S. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA* (2016).

[29] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[30] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[31] HOLEN, V. Experiments and fun with the linux disk cache, `https://www.linuxatemyram.com/play.html` 2017.

[32] HORN, C. Do we really need swap on modern systems?, `https://www.redhat.com/en/blog/do-we-really-need-swap-modern-systems` Feb. 2017.

[33] INCI, M. S., GULMEZOGLU, B., IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES* (2016), vol. 9813 of *LNCS*, Springer, pp. 368–388.

[34] IRAZOQUI, G., EISENBARTH, T., AND SUNAR, B. Cross processor cache attacks. In *AsiaCCS* (2016).

[35] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Lucky 13 strikes back. In *AsiaCCS'15* (2015).

[36] JACKSON, C., BORTZ, A., BONEH, D., AND MITCHELL, J. C. Protecting browser state from web privacy attacks. In *WWW* (2006).

[37] JIA, Y., DONG, X., LIANG, Z., AND SAXENA, P. I know where you've been: Geo-inference attacks via the browser cache. *IEEE Internet Computing 19*, 1 (2015), 44–53.

[38] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: An effective improvement of the clock replacement. In *USENIX ATC* (2005).

[39] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre attacks: Exploiting speculative execution. In *S&P* (2019).

[40] KOCHER, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[41] LAMPSON, B. W. A note on the confinement problem. *Communications of the ACM 16*, 10 (1973), 613–615.

[42] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[43] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., FOGH, A., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium* (2018).

[44] MAURICE, C., NEUMANN, C., HEEN, O., AND FRANCILLON, A. C5: Cross-Cores Cache Covert Channel. In *DIMVA* (2015).

[45] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., ALBERTO BOANO, C., MANGARD, S., AND RÖMER, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[46] MICROSOFT. Cache and Memory Manager Improvements, `https://docs.microsoft.com/en-us/windows-server/administration/performance-tuning/subsystem/cache-memory-management/improvements-in-windows-server` Apr. 2017.

[47] MICROSOFT. Programming reference for windows api, `https://docs.microsoft.com/en-us/windows/desktop/api/index` July 2018.

[48] MICROSOFT. Windows desktop applications, `https://msdn.microsoft.com/en-us/library/windows/desktop/aa906039.aspx` July 2018.

[49] MONACO, J. Sok: Keylogging side channels. In *S&P* (2018).

[50] NIEMIETZ, M., AND SCHWENK, J. UI Redressing Attacks on Android Devices. *Black Hat Abu Dhabi* (2012).

[51] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[52] OWENS, R., AND WANG, W. Non-interactive OS fingerprinting through memory de-duplication technique in virtual machines. In *International Performance Computing and Communications Conference* (2011).

[53] PERCIVAL, C. Cache missing for fun and profit. In *BSDCan* (2005).

[54] PESSL, P., GRUSS, D., MAURICE, C., SCHWARZ, M., AND MANGARD, S. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium* (2016).

[55] RAZAVI, K., GRAS, B., BOSMAN, E., PRENEEL, B., GIUFFRIDA, C., AND BOS, H. Flip feng shui: Hammering a needle in the software stack. In *USENIX Security Symposium* (2016).

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

[57] REN, C., ZHANG, Y., XUE, H., WEI, T., AND LIU, P. Towards discovering and understanding task hijacking in android. In *USENIX Security Symposium* (2015).

[58] RINNE, T. phpmyfaq - open source faq system for php and mysql, postgresql and other databases, `https://github.com/thorsten/phpMyFAQ` 2018.

[59] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS* (2009).

[60] RUSSINOVICH, M. Inside memory management, part 2, `https://www.itprotoday.com/management-mobility/inside-memory-management-part-2` Aug. 1998.

[61] RUSSINOVICH, M. E., SOLOMON, D. A., AND IONESCU, A. *Windows internals.* Pearson Education, 2012.

[62] RYDSTEDT, G., GOURDIN, B., BURSZTEIN, E., AND BONEH, D. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *4th USENIX Conference on Offensive Technologies* (2010).

[63] SCHMID, P. Windows vista's superfetch and readyboost analyzed, `https://www.tomshardware.com/reviews/windows-vista-superfetch-and-readyboostanalyzed,1532-2.html` Jan. 2007.

[64] SCHWARZ, M., LIPP, M., GRUSS, D., WEISER, S., MAURICE, C., SPREITZER, R., AND MANGARD, S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS* (2018).

[65] SCHWARZ, M., SCHWARZL, M., LIPP, M., AND GRUSS, D. Netspectre: Read arbitrary memory over network. *arXiv:1807.10535* (2018).

[66] SEABORN, M., AND DULLIEN, T. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *Black Hat Briefings* (2015).

[67] SONG, D. X., WAGNER, D., AND TIAN, X. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium* (2001).

[68] SUZAKI, K., IIJIMA, K., YAGI, T., AND ARTHO, C. Memory Deduplication as a Threat to the Guest OS. In *EuroSys* (2011).

[69] TALLIS, B. Best ssds: Q2 2018, `https://www.anandtech.com/show/9799/best-ssds` Mar. 2018.

[70] TIWARI, T., AND TRACHTENBERG, A. POSTER: Cashing in on the File-System Cache. In *ACM SIGSAC Conference on Computer and Communications Security* (2018).

[71] VAN DER VEEN, V., FRATANTONIO, Y., LINDORFER, M., GRUSS, D., MAURICE, C., VIGNA, G., BOS, H., RAZAVI, K., AND GIUFFRIDA, C. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *CCS'16* (2016).

[72] Van Goethem, T., Joosen, W., and Nikiforakis, N. The clock is still ticking: Timing attacks in the modern web. In *CCS* (2015).

[73] Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., and Gunter, C. A. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *CCS* (2017).

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

[75] Xiao, J., Xu, Z., Huang, H., and Wang, H. A covert channel construction in a virtualized environment. In *CCS* (2012).

[76] Xiao, J., Xu, Z., Huang, H., and Wang, H. Security implications of memory deduplication in a virtualized environment. In *International Conference on Dependable Systems and Networks (DSN)* (2013).

[77] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[78] Young, J. Nsa tempest documents. *CRYPTOME* (2002).

[79] Zhang, K., and Wang, X. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security Symposium* (2009).

[80] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS* (2014).