Eviction Notice: Reviving and Advancing Page Cache Attacks

Sudheendra Raghav Neela, Jonas Juffinger, Lukas Maar, Daniel Gruss
Graz University of Technology

***Abstract*—Page cache attacks are hardware-agnostic and can have a high temporal and spatial resolution. With mitigations deployed since 2019, only Evict+Reload-style timing measurements remain, but suffer from a very low temporal resolution and a high impact on system performance due to eviction.**

In this paper, we show that the problem of page cache attacks is significantly larger than anticipated. We first present a new systematic approach to page cache attacks based on four primitives: flush, reload, evict, and monitor. From these primitives, we derive five generic attack techniques on the page cache: Flush+Monitor, Flush+Reload, Flush+Flush, Evict+Monitor, and Evict+Reload. We show mechanisms for all primitives that operate on fully up-to-date Linux kernels, bypassing existing mitigations. We demonstrate the practicality of our revived page cache attacks in three scenarios, showing that we advance the state of the art by orders of magnitude in terms of spatial and temporal attack resolution: First, the channel capacity with our fastest attack (Flush+Monitor) achieves an average capacity of 37.7 kB/s in a cross-process covert channel. Second, for low-frequency attacks, we demonstrate inter-keystroke timing and event detection attacks across processes, with a spatial resolution of 4 kB and a temporal resolution of 0.8 µs, improving the state of the art by 6 orders of magnitude. Third, in a website-fingerprinting attack, we achieve an F_1 score of 90.54 % in a top-100 closed-world scenario. We conclude that further mitigations are necessary against the page cache side channel.

I. Introduction

Caches play a central role in hardware and software system performance by hiding memory latency. They typically buffer data based on temporal (e.g., recently used data) and spatial locality (e.g., prefetching code and data in spatial proximity). Side channels have been known for decades [38], [77], with generic techniques identified in the past two decades, including Evict+Time [51], Prime+Probe [51], [53], Flush+Reload [76], Evict+Reload [25], and Flush+Flush [24]. While these techniques have been described primarily for CPU hardware caches, they have been generically applied in various contexts [1], [15], [16], [20], [22], [44], [54].

Several works investigated timing side channels in software caches [17], [22], [45], [73]. Gruss et al. [22] mounted the first attack on the operating system's page cache. Several other works focused on other timing differences in the kernel [9], [31], [40], [45], [46], [52], [67], [78]. Boskov et al. [5] used the Linux `mincore` system call (syscall) to exploit the page cache across Docker containers. Schwarzl et al. [66] showed that page cache attacks can be valuable in a template attack, combining multiple channels with different spatial resolutions. Two of these [5], [22] monitor the page cache using `mincore`, another [66] uses the `preadv2` syscall, and the last [34] targets specific SSD features for fast eviction and reloads, with the reload being the source of information leakage. However, the `mincore` syscall has been patched since 2019 to mitigate these cross-process and cross-container attacks: `mincore` now only reports the number of owned pages a process has cached [37] but not per-page residency information. Hence, even if a victim loads a page into the page cache, the attacker does not see a change in the information `mincore` provides.

Schwarzl et al. [66] showed that an attacker can still mount a page cache attack using an Evict+Monitor-style attack: The attacker continuously evicts the target page and uses `preadv2` with the `RWF_NOWAIT` flag. If the page is already in RAM, `preadv2` & `RWF_NOWAIT` returns data; otherwise, the syscall & flag return early without bringing data into the RAM. A limitation of prior work [5], [22], [66] is the fragility of eviction strategies employed, which may make the system unreliable due to the dynamically allocated non-page-cache pages used during eviction. Furthermore, as reported by Schwarzl et al. [66], eviction is slow on recent Linux kernels, typically in the range of multiple seconds.

In this paper, we show that the page cache attack surface is significantly larger than anticipated and existing mitigations can be bypassed. Even worse, we show that we can not only revive previously mitigated page cache attacks but also increase their temporal resolution by 6 orders of magnitude. The foundation of our paper is a systematic approach dividing the interaction with the Linux page cache into four primitives: flush, reload, evict, and monitor. For each primitive and mechanism, we determine the origin (e.g., kernel functions) that causes the leakage or manipulation of page-cache residency.

Based on this analysis, we (1) systematically backtrack for flush, which operations on different file systems can lead to a page cache flush operation; (2) for reload, we perform a systematic and exhaustive manual analysis of all system calls accepting file descriptors or file paths whether they lead to kernel functionality that can be used as a reload primitive; (3) for monitor, we discover three independent paths through the kernel that lead to equally exploitable behavior; and (4) for evict, the mechanism which was best studied so far, we propose a small extension to the state of the art using a

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240006
www.ndss-symposium.org

dynamic adjustment to rapidly changing memory pressure. We show that for each primitive, an attacker has viable mechanisms to bypass existing mitigations.

We extend our systematic view of the attack primitives by deriving five generic attack techniques on the page cache, classifying both known techniques and those that have not been used in page cache attacks yet: Evict+Reload [34], [66], Flush+Reload, Evict+Monitor [5], [22], [66], Flush+Flush, and Flush+Monitor. Our attack techniques are hardware-agnostic and work provided the victim has access to the target file. For our Flush+Flush attack, we observe that unprivileged page-cache flush operations leak the page cache status with their execution time. Since our attack techniques work entirely on the software level, they are agnostic to CPU count, core scheduling, and other hardware configurations.

We evaluate our generic techniques in three attack scenarios: First, we evaluate the channel capacity in a cross-process scenario and show that the fastest of the five techniques (Flush+Monitor with `cachestat`) achieves an average capacity of $37.7\,\text{kB/s}$ with less than $3 \times 10^{-4}\%$ error, which is 5.3 times faster than prior work [22]. Second, we demonstrate that we can monitor events in co-located processes across container boundaries following the threat model of Boskov et al. [5]. We mount a UI redressing attack, an inter-keystroke timing attack, detect events inside programs, and detect the launch of shells in co-located containers. The spatial resolution of our attack is $4\,\text{kB}$, which is inherent to page cache attacks. We achieve a temporal resolution of $0.8\,\mu\text{s}$ with more than 1 million measurements per second, which is 6 orders of magnitude higher resolution than state-of-the-art attacks using `mincore` prior to any mitigations. Third, we evaluate our attacks in a website-fingerprinting attack, with three of our attacks achieving an F_1 score of more than $85\,\%$ and our best attack (Flush+Monitor) achieving $90.5\,\%$ in a closed-world scenario across the top 100 websites.

Overall, our attacks show that ad hoc mitigations are insufficient to sustainably mitigate the information leakage from the page cache. We discuss existing mitigation approaches that primarily focus on the leakage side and why they are insufficient to prevent the information leakage presented in this work. We conclude that further fundamental mitigations are necessary to reduce the side-channel attack surface of the page cache.

Contributions. In summary, our main contributions are:

1) We classify page-cache attack techniques into a system of four primitives—flush, reload, evict, and monitor—and derive five generic attack techniques, classifying both known techniques and techniques that have not been used for page cache attacks yet: Flush+Monitor, Flush+Reload, Flush+Flush, Evict+Monitor, and Evict+Reload.
2) We systematically backtrack *flush* on different file systems, exhaustively manually analyze all system calls accepting file descriptors or paths for *reload*, precisely analyze kernel code execution for *monitor*, resulting in a set of mechanisms that can be used to build full attacks on up-to-date Linux kernels that bypass existing mitigations.
3) We measure the capacity of all attack techniques in cross-process covert channels. Our fastest attack (Flush+Monitor) achieves an average capacity of $37.7\,\text{kB/s}$ with less than $3 \times 10^{-4}\,\%$ error, 5.3 times faster than prior work.
4) In our evaluation, we present a keystroke timing attack (average 5.5 keys/second), a UI redressing attack with a temporal resolution 6 orders of magnitude higher than state of the art, event detection inside programs, a cross-container app launch detection, and a website-fingerprinting attack with our best attack achieving $90.5\,\%$ in a closed-world scenario across the top 100 visited websites.

Outline. Section II provides background on page cache attacks. Section III introduces our attack primitives and Section IV our threat model. Section V constructs generic attack techniques and Section VI evaluates them in different scenarios. Section VII discusses mitigations. Section VIII concludes.

Responsible Disclosure. In January and April 2025, we disclosed our findings to the Linux kernel security team. The `cachestat` syscall was mitigated in February 2025 and was assigned CVE-2025-21691.

II. Background

In this section, we provide a brief overview of the page cache and side-channel attacks.

A. Operating System Page Cache

Since hard disk drives (HDDs) and solid-state disks (SSDs) have comparably high access latency, modern operating systems implement a page cache to buffer recently and frequently accessed memory. The page cache generally buffers all files regardless of how they are used, including memory-mapped files and files accessed via file operations and including shared library and common binary pages that are mapped multiple times by different processes to save memory. Writing to a page in the page cache either requires write permissions or follows copy-on-write semantics.

Linux maintains two doubly-linked lists for pages in the page cache: the `active_list` for pages recently accessed and the `inactive_list` for pages that are not recently accessed. Pages are moved from the active to the inactive list using a variant of the LRU replacement algorithm [11], [12], [13], [32]. Linux maintains a third list for pages that have been recently evicted. Ideally, the page cache occupies all available but otherwise unused memory to minimize disk I/O.

B. Side-Channel Attacks

The principle of side-channel attacks was first scientifically explored by Kocher [36]. Following this seminal work, research initially focused on cryptographic implementations and the exploitation of timing differences stemming from CPU caches [3], [51], [53], [72]. Over the last two decades, this research has yielded generic cache attack techniques independent of the underlying concrete cache technology used. The most prominent cache attack techniques are Prime+Probe by Osvik et al. [51], and Flush+Reload by Yarom et al.

[76]. Numerous variants have been presented with Evict+Reload [25] and Flush+Flush [24] as noteworthy examples.

Caches exist throughout the hardware-software stack, leading to significant performance gains as well as side-channel leakage across all layers, including various software caches [17], [22], [26], [31], [45], [73]. For cryptographic algorithms, constant-time code [36] is a principle to prevent timing leakage stemming from implementation (e.g., various caches); however, it is not realistic for general-purpose code [65].

Besides the page cache [22] (see Section II-C), many system-level resources have been attacked with side channels. Jiang et al. [31] and Chen et al. [9] exploited contention on file sync operations and write buffers to build covert channels. Lee et al. [40] and Maar et al. [45] exploited timing side channels in the Linux slab allocator. Shen et al. [67] presented a covert channel based on kernel locking primitives [78].

C. Page Cache Attacks and Related Works

Knowing which pages are in the page cache can provide tremendous performance benefits for certain use cases [68]. Hence, Linux provides several mechanisms to check whether a page is resident in the page cache. Prior work exploited the `mincore` system call for Linux [5], [22]. The syscall provides per-page information for mapped pages, informing the caller which of its mapped pages is currently in the page cache. This information is equivalent to the information gained by the reload operation in Flush+Reload. The difference between `mincore` and a reload is that the former is not destructive, *i.e.*, it does not bring a page into the page cache.

Gruss et al. [22] demonstrated the first page-cache attacks. The attacks they mounted could be called Evict+Monitor, using page cache eviction followed by `mincore`, a monitoring primitive. This syscall was thus patched in response [37]. Gruss et al. [22] achieved a channel capacity of 7 kB/s (Linux) to 273 kB/s (Windows) and mounted side-channel attacks e.g., on keystrokes, with a temporal resolution of 149 ms. Boskov et al. [5] exploited the same interface in cross-container attacks. Since the mitigation patches, `mincore` only returns valid data if the process has `CAP_SYS_ADMIN` permissions, or if it can modify the underlying file [37]. Consequently, using `mincore` to monitor activity in shared libraries or binaries has been mitigated.

Instead of `mincore`, Schwarzl et al. [66] showed that an attacker can also use the `preadv2` system call with the `RWF_NOWAIT` flag. When a page is not in the page cache, this syscall & flag returns early *without* bringing the page into the page cache, thereby leaking whether the page resides in the cache. Their approach, Evict+Monitor in our classification, requires constant eviction to restore the initial state, lowering the temporal resolution to 2 s or more. This resolution is too low for keystroke timing attacks and UI-redressing attacks.

An alternative to these syscalls is timing information [22], [34], e.g., via reload. The attacker simply accesses the page, loading it into the page cache. By measuring the time it takes to access the page, the attacker can infer whether the page was in the page cache or loaded from the disk. This approach is destructive and, thus, the performance is substantially lower, in the range of multiple seconds for eviction.

Page Cache Eviction. The core of prior page cache attacks is page cache eviction. Gruss et al. [23] described the first eviction strategy for the page cache in the context of Rowhammer attacks. Instead of exhausting memory via dynamically allocated memory, they use memory mappings of read-only files that remain equally evictable as other pages in the page cache. Hence, their eviction is less fragile than previous approaches but still takes 2.68 s per eviction.

Gruss et al. [22] tried to overcome this limitation by introducing three eviction sets: The first eviction set comprises pages already in the page cache and used by other processes. A thread runs constantly to keep these pages among the most recently accessed. The second eviction set is disk-backed pages brought to the page cache when accessed. These pages are accessed rarely and randomly to maintain a set of eviction candidates that are more likely to be evicted than most pages from the initial eviction set. The third eviction set contains dynamically allocated, *i.e.*, non-evictable pages. This set is used to increase memory pressure up far enough to evict the target pages from the page cache but far smaller than the number of evictable pages in the first two eviction sets.

As an additional optimization Gruss et al. [22] use `madvise` with the `MADV_DONTNEED` for the target page and `MADV_WILLNEED` for all other pages, as well as `posix_fadvise` with `POSIX_FADV_DONTNEED` for the target page. With this optimization they report a much lower runtime of 149 ms for their eviction. Juffinger et al. [34] showed that eviction can perform faster (22 ms) on SSDs with specific hardware features, *viz.* host-memory buffer.

In Section III, we describe that using `posix_fadvise` with `POSIX_FADV_DONTNEED` is equivalent to a flush operation, with the caveat that the pages are not mapped. However, it is likely that flushing performed by Gruss et al. [22] was ineffective as it was operated on a mapped page; in Section III-B, we explain the reason why flushing mapped pages is ineffective. Therefore, we believe Gruss et al. [22] in many cases mounted only an Evict+Monitor. The runtime of Evict+Monitor is much higher due to eviction runtimes in the range of a hundred milliseconds, while the runtime of Flush+Monitor is in the order of microseconds (see Section V).

III. Attack Primitives

In this section, we discuss mechanisms, *viz.* syscalls and techniques, to implement each of the abstract primitives and analyze potential convergence points inside the Linux kernel as well as alternative routes depending on file systems or available system calls. Throughout the paper, we instantiate the primitives with suitable mechanisms available to the attacker. Noteworthy, prior works on page cache attacks [22] were mitigated in Linux in 2019, specifically with a focus on the `mincore` system call [37]. We show that with various alternative mechanisms to implement our attack primitives (reload, monitor, evict, and flush), we can bypass these mitigations

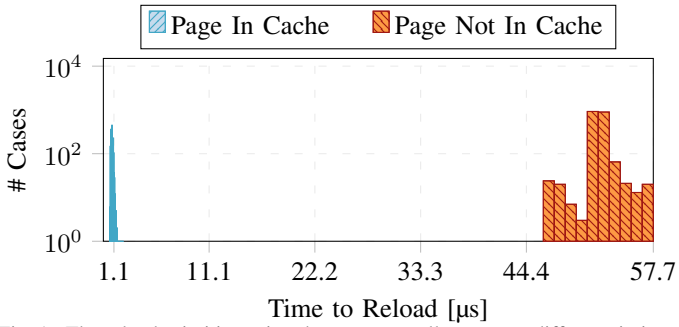

Fig. 1. The reload primitive using the `read` syscall exposes a different timing depending on whether a target page is currently in the page cache or not.

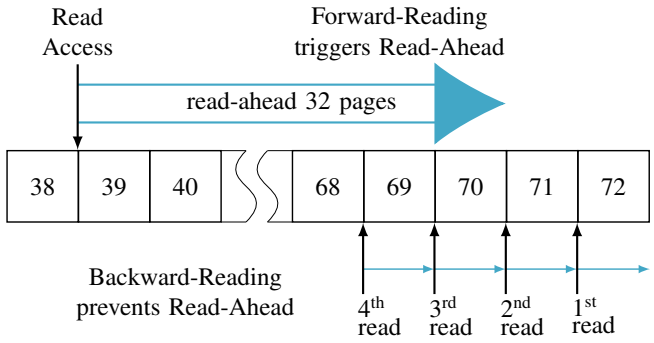

Fig. 2. Read operations trigger the kernel's read-ahead mechanism. Reading the file backward does not trigger this mechanism, resulting in stable timings and avoiding issues with monitoring multiple pages per file.

and, as we later show, improve attack performance by orders of magnitude. In Section V, we combine these primitives to build attack techniques for increasing the temporal resolution of attacks on the page cache. All experiments in this section and the following sections were performed on an AMD Ryzen 7 7700X (Zen 4) with 32 GB of DDR5 memory running Ubuntu 22.04.4 LTS (Linux kernel 6.8.0) and an ext4 file system.

A. Reload

The reload primitive determines whether a page is present in the page cache by measuring the time it takes a process to read a page. A long reload time indicates that the page was not in the page cache, while a short reload time indicates the opposite. A straightforward reload mechanism is the `read` syscall. In Figure 1, we show the significant difference in execution time of `read` when a page is in the page cache compared to when it is not. On our machine, using the `read` syscall to read a page takes over fifty times longer when the page is not cached (210 000 cycles (n=2 × 10^6, $\sigma_{\bar{x}}$=68 000)) than when being present in the page cache (3 900 cycles (n=2 × 10^6, $\sigma_{\bar{x}}$=900)) (46.66 µs versus 0.86 µs). To ensure the page is not in the page cache, we remove the page from the page cache using the flush primitive (see Section III-B).

Bypassing Read-Ahead. As an optimization to increase performance, Linux, by default, reads 32 pages ahead upon any file access [8], akin to the CPU's prefetcher. Similar as on the CPU-level, with prefetching and Flush+Reload, this limits what an attacker can observe, *i.e.*, the attacker cannot monitor multiple locations within the prefetching range. Prior work [22], [66] mentions this read-ahead mechanism as a limitation and a source of noise, as it inadvertently destroys information regarding the presence of the subsequent 32 pages in the page cache. This limitation was overcome either by applying more memory pressure [22], or by passing `MADV_RANDOM` to `madvise` indicating a random read order while being cognizant that this may not always work [5], [66]. We investigated this issue and confirmed that read-ahead can interfere with measurements. In our work, we find two novel methods to entirely bypass the read-ahead mechanism:

1) **Backward-Reading**: By reading the file *backward*, the read-ahead mechanism can be bypassed, as illustrated in Figure 2. We observe that the read-ahead mechanism does not affect backward reading in any of our experiments on Linux. As this backwards-reading naturally undermines the read-ahead mechanism, thus severely reducing performance, it provides the attacker with precise timing differences that enable the reload primitive.
2) **`readahead` Syscall**: This syscall is an explicit request to the read-ahead mechanism to bring pages from disk. We notice a difference in the time it takes to bring pages when they are in cache (∼0.46 µs) versus when they are to be fetched from disk (∼2.64 µs). We also find that `posix_fadvise` + `POSIX_FADV_WILLNEED` is similarly plumbed in the kernel, leading to similar timings as `readahead`. Although `readahead` is a faster reload mechanism than `read`, it quickly becomes unreliable under memory pressure and some system activity.

Manual Exhaustive System Call Analysis. We analyze all system calls that accept a file descriptor or a file path as an argument to determine which can be used as a reload mechanism. Given the complex data structures and variety of flags available for files, we opted for a manual analysis by a human expert instrumenting the syscalls to automatically generate call traces that reveal which kernel functions are used, *i.e.*, which system calls interact with the page cache.

Automated methods, e.g., static analysis using sparse [7], smatch [2], or coccinelle [39], are challenging to employ for a multitude of reasons: function and operation table pointers typically resolve at run time; configuration options often lead to divergent paths; filesystems interact with different subsystems resulting in vastly different behavior; hardware-specific config options programmed variably (and in turn macros expanding in separate, hardware-specific ways); unclear runtime permission interactions with user-chosen or distribution-default security subsystems; distribution patches whose code must be considered; numerous fallback functions; syscalls having more than one definition depending on configuration preferences along with runtime options. Altogether, these reasons lead to a search space explosion overwhelming automation attempts.

Therefore, we use a manual and tool-assisted system to determine which syscalls accepting file descriptors can be used as a reload mechanism. We first generate tags for the

TABLE I
LINUX SYSCALLS THAT BEHAVE AS RELOAD MECHANISMS

Syscall	Reload Time [μs] From Cache	Reload Time [μs] From Disk	Kernel Function
`copy_file_range`	3.64	50.07	`filemap_splice_read`
`mmap` access	0.97	142.72	`filemap_map_pages`
`posix_fadvise` + `POSIX_FADV_WILLNEED`	0.45	2.63	`page_cache_ra_unbounded`
`pread`	0.91	46.90	`filemap_read`
`preadv`	1.03	46.90	`filemap_read`
`preadv2`	0.98	46.91	`filemap_read`
`read`	0.86	46.66	`filemap_read`
`readahead`	0.46	2.63	`page_cache_ra_unbounded`
`readv`	0.84	67.79	`filemap_read`
`sendfile`	3.42	49.51	`filemap_splice_read`
`splice`	0.67	48.46	`filemap_splice_read`

```
x64_sys_call()
  __x64_sys_read()
    ksys_read()
      vfs_read()
        rw_verify_area()
        ext4_file_read_iter()
          generic_file_read_iter()
            filemap_read()
              filemap_get_pages()
```

Listing 1. The path taken by the `read` syscall.

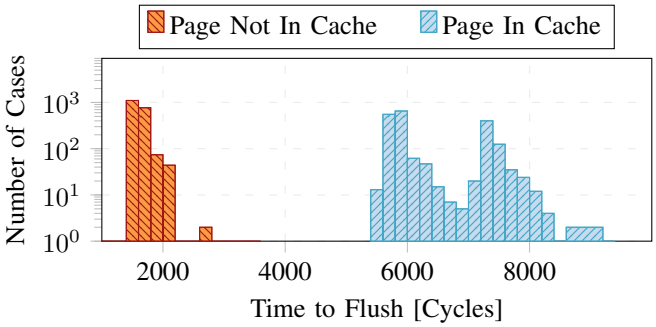

Fig. 3. The flush primitive using `posix_fadvise` with `POSIX_FADV_DONTNEED` exposes a different timing depending on whether the target page is currently in the page cache or not.

entire kernel using ctags [14], resulting in 7.9 million tags. From the Linux x86 syscall table [42], we generate signatures for every syscall, resulting in 408 syscalls with 509 total function signatures; there are duplicates due to the syscall ABI. Filtering for syscalls accepting file descriptors results in 175 signatures including duplicates; de-duplicating generates 119 total syscalls. Filtering out syscalls which require a "special" file descriptor (`pipefd`, `socketfd`, `epollfd`, `dirfd`, `pidfd`, `fanotify`), or requires privileges, or capabilities, we reduce to 46 remaining syscalls.

We write minimal code for each syscall accounting for flags accepted by the syscall and flush the testing file (see Section III-B) prior to every recording. Furthermore, we use `ftrace` [63] to record call graphs for determining which kernel functions introduce the timing differences when loading a page from the page cache versus when loading it from disk. As an example, we show the path taken by `read` in Listing 1. The `read` syscall calls `filemap_read` in its execution, a MM API call which "read[s] data from the page cache" or brings it into the page cache if the data is not present [41].

Out of 46 system calls, we found eleven system calls that can be used as reload mechanisms, as shown in Table I. For the eleven syscalls, we find three Linux MM API calls — `filemap_read`, `filemap_splice_read`, `page_cache_ra_unbounded`— and one internal function — `filemap_map_pages`— that these syscalls call in their call trace. Two of these MM API calls — `filemap_read`, `filemap_splice_read`— further converge to `filemap_get_pages`. Note that the page cache is implemented at the MM API level and any functions beyond can be considered as already inside the page cache implementation, *i.e.*, the interaction with the page cache is already evident at this point. Finally, there is an abundance of alternative paths to reach the same reload primitives, e.g., through drivers and various file systems, that provide the same functionality to an attacker. Given this abundance of alternative paths, we conclude that both for exploiting and mitigating a reload mechanism, it is more promising to focus either on the MM API on the one side or the syscall interface on the other side but not the numerous alternative paths in between them.

Reload mechanisms are inherently destructive: while they leak whether a page is in the page cache, they also modify the page cache state by loading the specific page into the page cache. That is, after using the reload primitive, any previous page cache state is irreversibly lost. With the page cached, victim accesses cannot be observed anymore. Hence, using a reload requires repeated flushing or eviction after every reload to reset the page cache state.

B. Flush

Flush is an unprivileged and deterministic primitive that removes pages from the page cache. Prior work [5], [22] used `posix_fadvise` with `POSIX_FADV_DONTNEED` along with `madvise` with `MADV_DONTNEED` to hasten a page's

```
x64_sys_call()
  __x64_sys_fadvise64()
    ksys_fadvise64_64()
    generic_fadvise()
      __filemap_fdatawrite_range()
      lru_add_drain()
      mapping_try_invalidate()
        mapping_evict_folio()
```

```
/* Remove an unused folio from the page-cache */
long mapping_evict_folio(...) {
  ...
  /* ... if any page in the folio is mapped */
  if (folio_ref_count(folio) >
      folio_nr_pages(folio) +
      folio_has_private(folio) + 1)
    return 0;
  ...
}
```

Fig. 4. The flush primitive, `posix_fadvise` with `POSIX_FADV_DONTNEED`, cannot flush a mapped page. The function `mapping_evict_folio` prevents the removal of a mapped page from the page cache.

eviction from the page cache. Unlike prior work, we do not use `madvise` with `MADV_DONTNEED` simultaneously as we find that mapping a file prevents it from being flushed. In Figure 4, we show why `posix_fadvise` with `POSIX_FADV_DONTNEED` does not flush a mapped page.

Prior work has not reported (and likely also not observed) this behavior, as they worked with mapped files to be able to call `madvise` [66] or `mincore` [5], [22]; thus, flushing has not been employed in side-channel attacks on the page cache so far. This discovery allows us to construct the flush primitive (`posix_fadvise` with `POSIX_FADV_DONTNEED` only) which speeds up the attack presented by Schwarzl et al. [66] by six orders of magnitude (see Section V-B).

The execution time of the flush primitive is shown in Figure 3. We can see that not only is the flush primitive a fast and reliable method to remove pages from the page cache, but it also leaks whether a page was cached in its execution time. This behavior was first discovered by Gruss et al. [24] in the context of CPU caches. On our machine, we find that flushing pages not in the page cache requires 1620 cycles (n=2×10^6, σ=120), while flushing pages in cache takes 6340 cycles (n=2×10^6, σ=760). We exploit this timing difference to determine whether a page is present in the page cache with our Flush+Flush attack in Section V-E.

Backtracking Flush for Attacks on Different File Systems. We pursued a similar approach to find flush mechanisms as for the reload mechanisms. Through this approach, we indeed discovered one flush mechanism in our setup: The kernel function responsible for the flushing behavior is `mapping_try_invalidate` (see Figure 4). In contrast to the reload mechanisms, we find that there are not many alternative paths to reach the flush primitive. In fact, this function is only called twice in the kernel, once by `posix_fadvise` and another

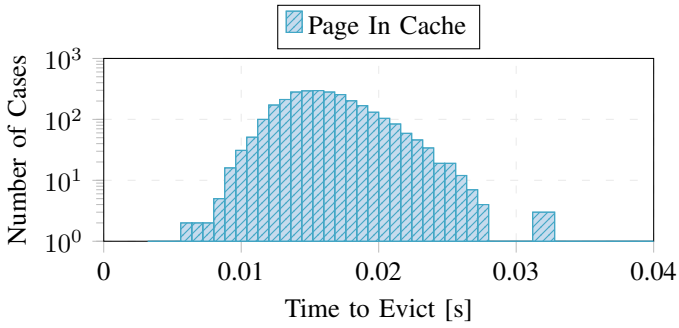

Fig. 5. The evict primitive, using memory pressure with multiple eviction sets, is significantly slower than flushing pages from the page cache. An attacker can use the evict primitive in Evict+Reload or Evict+Monitor attacks.

by `invalidate_mapping_pages`. The latter function is predominantly only called by filesystems and drivers, some of which use it to implement direct IO writes (e.g., Btrfs, ext2, ext4, GFS2) while others employ the function to implement custom page-cache clearing methods (e.g., NILFS2, CephFS, EROFS). The only two non-filesystem and non-driver usages of `invalidate_mapping_pages` are via `posix_fadvise` (retry attempt if `mapping_try_invalidate` failed) and a privileged sysctl interface to drop the page cache.

C. Evict

By accessing a large number of pages, an attacker can induce sufficient memory pressure to forcefully evict a target page from the page cache [22]. Since the entire page cache acts as a fully-associative cache, there is no way to target a subset of the cache or even a *single* targeted page. Instead, many unrelated pages are also evicted alongside the targeted page. Eviction is a non-deterministic and difficult-to-control primitive as the kernel decides which pages are evicted when low on memory using a multi-generational LRU algorithm [13].

We use the page cache eviction technique presented by Gruss et al. [22], albeit without the `posix_fadvise` optimization that we identified as a flushing operation (see Section III-B). The spread of the evict primitive's execution time is shown in Figure 5. With the following three eviction sets, it takes 16 ms (n=3000, σ=3.4) on average to evict a targeted page on a system with roughly 500 MiB free, as reported by the `free` command.

Eviction Set 1 are pages frequently used by processes and their eviction would slow the processes down. To avoid slowing the system, a thread constantly reads this set of pages. These pages are marked with `MADV_WILLNEED`, and CPU usage is kept low by combining `sched_yield` and sleeps.

Eviction Set 2 are pages not yet in cache. They are randomly accessed to evict pages present in the page cache. On our system, this set totals to 70 GB. These pages are also marked with `MADV_WILLNEED`, and CPU usage is kept low by combining `sched_yield` and sleeps.

Eviction Set 3 are pages serving as a baseline memory pressure, preventing regions of memory from being used. These pages are not to be swapped out, which can be achieved

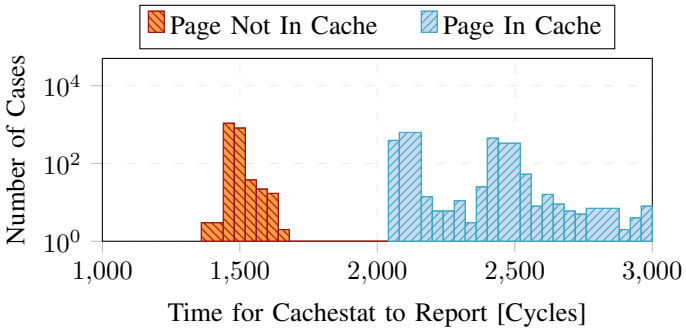

Fig. 6. The mechanism behind the monitor primitive, `cachestat`, requires less than 3 000 cycles to report whether a page is in the page cache, making it a very fast source of information leakage.

by using the `mlock` syscall. This syscall prevents up to $1/8$ of RAM from being swapped out on Debian-based systems by default; however, an attacker can easily overcome this limit by spawning more processes. This eviction set substantially reduces the amount of free memory and lowers the overall time to perform eviction.

Eviction Set 4: Dynamic Memory Pressure. Eviction with the three eviction sets works well for the scenarios described by Gruss et al. [22]. However, we observe that the low and static amount of free memory can lead to stability issues if the memory usage varies widely for the different targeted activities. A low static memory pressure results in unsuccessful evictions and increased execution times until a successful eviction. For instance, when fingerprinting websites based on their usage of `libxul`, the shared library used by Firefox (see Section VI-D), we notice that specific sites like openai.com occupy considerable amounts of memory (3 GB), while others like wikipedia.com occupy far less (90 MB). Hence, atop the three previously explained eviction sets, we find it necessary to introduce a fourth eviction set that dynamically responds to a program's memory footprint.

We design the fourth eviction set to allocate and free 64 MiB chunks of memory as needed. When the victim's pages are in the page cache, the thread allocates a 64 MiB region. Similarly, the thread de-allocates all regions when the victim's pages have been evicted. This information on whether the target pages are in the page cache is known to the thread either via the reload or monitor primitives, which we discuss in Section III-A and Section III-D, respectively. With this fourth eviction set, an attacker can efficiently and dynamically apply memory pressure for various attacks.

D. Monitor

The better alternative to the reload primitive is the monitor primitive, as it is not destructive: unlike the reload primitive, monitor merely reports the presence of a page in the page cache without bringing it into memory if it is not present. In the past, Gruss et al. [22] used the `mincore` syscall as the mechanism for the monitor primitive, which requires files to be mapped. However, this source of leakage has been patched since Linux kernel version 5.2 [37] and hence, it cannot be used in page cache attacks anymore. In Table II, we overview all monitor mechanisms that work at the level of pages or files.

TABLE II
LINUX SYSCALLS THAT LEAK PAGE-LEVEL OR FILE-LEVEL INFORMATION ABOUT PAGE-CACHE RESIDENCY

Syscall	Page-Level	File-Level	Prior work
`cachestat`	●	●	—
`preadv2` + `RWF_NOWAIT`	●	●	[66]
`mincore` (before mitigation)	●	●	[5], [22]
`inotify_add_watch` + `IN_ACCESS`	○	●	[64]

Symbols: The syscall leaks (●) or does not leak (○) page-cache residency.

Schwarzl et al. [66] used the `preadv2` syscall with the `RWF_NOWAIT` flag as a monitor mechanism. When a page is present in the page cache, this syscall performs a read; otherwise, it returns an `EAGAIN` error when the page is not in the page cache. This behavior is intended [43]. The `preadv2` syscall with `RWF_NOWAIT` takes 4 977 cycle (n=2 × 10^6, σ=3700) when a page is present in the page cache, while it takes 3 370 cycle (n=2 × 10^6, σ=985) when the page is not in page cache. Ruggia et al. [64] use the Android `FileObserver` class, which internally uses `inotify` [35], as a *file*-level monitor mechanism.

In Linux kernel version 6.5, a new syscall, `cachestat`, was introduced to "query the page cache state of large file sets" [55]. The `cachestat` syscall reports five page-cache statistics about a range in a file, of which we require only the number of cached pages. By specifying length and offset, an attacker can get information about a single page or multiple continuous pages with a single invocation of `cachestat`.

The execution time of the monitor primitive is shown in Figure 6. Evaluating `cachestat` as a monitor primitive, we find that it requires only 2280 cycles (n=2 × 10^6, σ=204) for a page in the page cache and 1470 cycles (n=2 × 10^6, σ=71) for a page not in cache. While there is a timing difference, the information on whether a page is present in the page cache is already contained in the return value, *i.e.*, there is no additional value in exploiting the timing difference. However, if future versions of the interface are hardened against page cache attacks, it is crucial not to overlook the timing difference.

We can retrofit prior page cache attacks that used a monitor primitive with `cachestat` [5], [22], [66] and, thus, re-enable them. Compared to `mincore`, `cachestat` requires a file descriptor instead of a mapped address. This is a significant advantage, as a file does not have to be mapped, allowing for reliable flush operations (see Section III-B). The `cachestat` syscall being fast and non-destructive makes monitor-based attack techniques (*i.e.*, Flush+Monitor and Evict+Monitor) the fastest and most reliable.

The `cachestat` syscall has been introduced as a faster variant of `mincore`, which we verify with two experiments. First, we measure the timestamps required by both syscalls.

Second, we analyze the accesses to the Translation Lookaside Buffer (TLB), indicating the number of memory accesses performed. In our experiments, we run `cachestat` and `mincore` on a 1 GiB file and a memory-mapped region, respectively. Each experiment is repeated 10 times for reliability. The results demonstrate that `cachestat` reduces timestamps by approximately 45 % and TLB accesses by around 30 %, showcasing a significantly faster syscall.

System Call Convergence Analysis. We analyze whether the three different monitor mechanisms converge to a single path in the kernel. Surprisingly, the monitor mechanisms we analyzed do not converge to a single path in the kernel to determine page cache residency. The `cachestat` syscall implements `filemap_cachestat`, which in turn directly operates on folios. The `mincore` syscall implements `mincore_walk_ops` operations, which walks over the page tables. The `preadv2` syscall with `RWF_NOWAIT` flag internally sets the `IOCB_NOIO` flag, which in turn makes `filemap_get_pages` return early with an error before the read can occur if a page is not present in the page cache, which is determined by a folio operation. Therefore, using the same approach as for flush primitives, we do not discover further monitor mechanisms. Instead, discovering further monitor primitives may require a different approach, as they appear to have distinct and independent paths to the page-cache residency information.

IV. Threat Model

We assume a standard threat model applied for similar side channels, e.g., Flush+Reload on CPU caches [25], [76] and prior page cache attacks [22]. In this threat model, the attacker has unprivileged local code execution on the target system, albeit under a separate user with no access to files and data belonging other users. The attacker requires read-only access to files that are installed for all users in the system, *i.e.*, shared libraries, executables, and shared data files. Based on this standard threat model, all primitives from Section III can be used by an attacker. The attacker's goal is to learn page-use information from other users' processes.

Cross-Container. Like prior work [5], [82], we mount attacks in a cross-container threat model, enabled by file and library sharing across containers. We specifically target Docker containers in which three of four primitives can be employed: flush, evict, and reload. Only the mechanism behind monitor, `cachestat`, is disabled by the default seccomp policy [48].

Target Page. We assume the attacker knows files and page numbers of interesting attack targets. The attacker can obtain this information in an offline analysis on a victim program by employing templating [66] to profile its file accesses.

V. Combining Primitives: Attack Techniques

In this section, we present five attack techniques based on the four primitives from Section III. Each attack strategy combines of two primitives, as shown in Figure 7. One primitive is used to remove a page from the page cache, *i.e.*, bring the page cache into a state where victim accesses can be observed, and the second leaks whether a specific page is currently in the page cache, *i.e.*, it is used to monitor the activity of a victim process. The techniques we present follow the standard naming pattern for attack techniques from microarchitectural attacks on CPU caches. Flush+Reload, Evict+Reload, and Flush+Flush follow the exact same working principle as their microarchitectural variants [24], [25], [76], Evict+Monitor and Flush+Monitor are logical extensions using the monitor primitive. Unless specified, we use `read` as reload and `cachestat` as monitor mechanism.

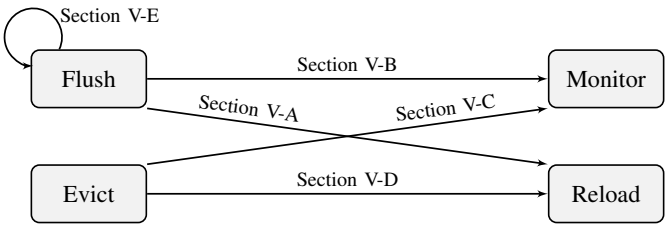

Fig. 7. Combining the primitives yields five different attack techniques. We cover all attack techniques in sub-sections of Section V.

A. Flush+Reload

Flush+Reload is a well-established side-channel attack on CPU caches [21], [27], [28], [30], [76], [81]. The attack principle can also be applied to the page cache by flushing a page and measuring the time it takes to reload the page. A short reload time indicates that the page was cached in the meantime, while a long reload time means it was not. In Section III-A, we found that the reload primitive takes 46.66 µs to bring a page into the page cache as compared to 0.86 µs if a page is already present in the page cache. This significant difference is easy to distinguish. As discussed in Section III-A, our reload primitive bypasses the kernel's read-ahead mechanism by using backward-reading.

The core Flush+Reload sequence comprises of three steps. First, the attacker flushes the target set of pages from the page cache. In the second step, the attacker waits for a short while for the victim to access pages from this set. Finally, the attacker times how long a reload operation takes on these pages to detect whether they were accessed.

Like Flush+Reload on processor caches, Flush+Reload on the page cache is susceptible to different sources of noise, like scheduling or changing CPU frequency. The CPU frequency can be stabilized by running a busy loop on the sibling SMT thread. Pinning a process to a specific core is possible without any privileges. The most significant source of noise is the scheduler interrupting the attacker while timing the reload. Similar to prior work [25], we find that yielding after every Flush+Reload sequence tends to be an effective method to reduce the chance that the kernel interrupts the attacker. While this makes the Flush+Reload loop slightly slower (from 58 µs to 64 µs), it makes Flush+Reload substantially more reliable. Note that the timing thresholds we report are dependent on the system configuration, e.g., CPU frequency, and similar as in other side-channel attacks (e.g., native Flush+Reload on

the CPU's hardware caches [76]) the thresholds have to be determined on the victim system. The similarities also extend to the read-ahead mechanism, which behaves very similar to the CPU's prefetcher [25], which is a known potential limitation for an attacker.

B. Flush+Monitor

Combining flush and monitor results in the least noisy attack strategy since both primitives are deterministic syscall interfaces. Moreover, the mechanisms behind both flush and monitor are faster than all the other primitives, with the upper limit defined by the time it takes to flush a page in the page cache, which is $0.8\,\mu s$ on our system (see Section III-B). This makes the Flush+Monitor attack strategy noiseless, information-rich, and fast.

The core sequence is similar to the steps of Flush+Reload. First, the attacker flushes the target set of pages. Second, the attacker waits for a short while. Third, instead of timing a reload operation, the attacker uses the monitor primitive to monitor whether the page is in the page cache or not. As the monitor operation is non-destructive, the attacker can loop over this third step until a cache hit is observed. Consequently, one round of the attack takes only $0.3\,\mu s$ while there is no victim access and $2.2\,\mu s$ for any round with victim access.

Schwarzl et al. [66] mount an Evict+Monitor page-cache attack with `preadv2` and the `RWF_NOWAIT` flag as the monitor mechanism. They require gaps of $\sim 2\,s$ for detecting keystrokes due to the very long time required for page-cache eviction. However, our *Flush+Monitor* attack requires only $0.6\,\mu s$ in the same setup while there is no victim access and $2.3\,\mu s$ for any round with a victim access. Thus, our new flush primitive improves the temporal resolution and sampling rate of the attack by Schwarzl et al. [66] by 6 orders of magnitude.

The monitor primitive does not suffer from any of the three sources of noise that reload suffers, nor is the monitor primitive affected by any read-ahead mechanism since the mechanism (`cachestat`) does not read any pages. Hence, it is the fastest and most reliable attack. In Section VI, we consistently observe Flush+Monitor-based attacks to outperform all other attack techniques.

C. Evict+Monitor

Considering that the flush primitive may be patched or unavailable in a concrete attack scenario, we also explore Evict+Monitor as an alternative to Flush+Monitor. While eviction is less reliable than flushing, we can use the monitor primitive to detect when the eviction was successful. Eviction is also much slower than flushing.

The core sequence is identical to Flush+Monitor, with three steps: *eviction*, waiting, and monitoring. Again, the monitor operation is non-destructive, allowing the attacker to loop over this third step until a cache hit is observed. Consequently, one round of the attack still takes only $0.068\,s$ for a page in the page cache compared to $2.1\,\mu s$ for a page not in cache. Prior work predominantly mounted Evict+Monitor [5], [22], [66].

D. Evict+Reload

If both flush and monitor are not available, an attacker can still resort to Evict+Reload, which only requires memory accesses for eviction and memory accesses for reloading. With eviction and reloading being less reliable than flush and monitor, the attack is also less reliable. This unreliability is due to timing variations and thresholds that have to be determined experimentally, but also due to eviction only having a certain success rate. Furthermore, the attack is made more challenging by constantly reloading victim pages, as this behavior prevents these pages from being evicted (due to the LRU).

The core sequence is identical to Flush+Reload, with three steps: *eviction*, waiting, and reloading. As the reload operation is destructive, the attacker has to increase the time between rounds, making Evict+Reload much slower, with one round of the attack taking $0.13\,s$ for a page in the page cache compared to $0.32\,ms$ for a page not in cache.

E. Flush+Flush

An interesting alternative if neither reload nor monitor are available is Flush+Flush, which has been explored as a side-channel attack on CPU caches [24]. We apply the same principle to the page cache, with the Flush+Flush sequence essentially being a single loop measuring the time it takes to flush a page from the page cache. At the same time, this resets the page cache state such that the subsequent victim access can be observed. Interestingly and similar to CPU caches [24], a high execution time ($1.4\,\mu s$) corresponds to a cache hit, whereas a low execution time ($0.36\,\mu s$) corresponds to a cache miss (see Section III-B). This difference is easily distinguishable, e.g., using a threshold of $0.88\,\mu s$. Similar to the monitor primitive, flushing does not trigger the read-ahead mechanism of the kernel (see Section III-A). A single round of the attack can be performed in $1.2\,\mu s$ when there is no victim access and $2.5\,\mu s$ when there is a victim access.

VI. Evaluation

In this section, we evaluate our five attack techniques on a variety of attacks. First, we evaluate the maximum channel capacity using a covert channel with each attack technique, reaching a transmission speed of up to $37.7\,kB/s$ with Flush+Monitor. We show how our attack techniques can be used to observe low-frequency events like starting processes, unrelated containers detecting shell launches in other container, or individual keystrokes typed by a user. Last, we present a website fingerprinting attack with an F_1 score of up to $90.5\,\%$.

A. Experimental Setup

We evaluated all attacks on an AMD Ryzen 7 7700X 8-Core CPU with $32\,GB$ DDR5 memory running Ubuntu 22.04.4 LTS (kernel 6.8.0) without swap and also verified them on an Intel Core i5-11300H with $16\,GB$ DDR4 memory running Arch Linux (kernel 6.12.4-arch1-1), which yields similar results as the attacks are hardware agnostic. We tested both with the default `mlock` limit of $1/8$ of RAM, e.g., with 8 processes, but also tested the higher limit of $1/4$ of RAM, e.g., with 4

TABLE III
COVERT CHANNELS WITH OUR FIVE ATTACK TECHNIQUES.

Attack Technique	Error Rate	Channel Capacity
Flush+Monitor	$< 3 \times 10^{-4}\%$	37.7 kB/s
Flush+Flush	1.8 %	33.9 kB/s
Flush+Reload	$< 2.9 \times 10^{-3}\%$	4.3 kB/s
Evict+Monitor	$< 8.9 \times 10^{-3}\%$	1.4 kB/s
Evict+Reload	$6 \times 10^{-4}\%$	1.2 kB/s

processes instead. Both approaches yield similar results as the processes are mostly idle and simply occupy memory.

The only special requirement for monitor-based attacks is that the system must run at least Linux version 6.5 since this is the version when `cachestat` was introduced. Since the `cachestat` syscall does not yet have an interface function in the libc, we manually called `cachestat` using the syscall function with its assigned number, 451.

For eviction-based attacks, we assume a base memory pressure from the first three eviction sets such that there is 4 GB of free memory. This assumption aligns with recent work using page cache eviction [33].

B. Covert Channel

To evaluate the maximum achieved channel capacity with two colluding communicating parties, we build a covert channel with each of our five attack techniques. Our fastest covert channel using Flush+Monitor reaches up to 37.7 kB/s with an error rate less than $3 \times 10^{-4}\%$. This is significantly faster than comparable covert channels in previous work, such as the one by Gruss et al. [22], who attained 7 kB/s.

On channels with transmission errors, we use the binary symmetric channel model to compute the true channel capacity T as $T = C \cdot (1 + ((1-p) \cdot \log_2(1-p) + p \cdot \log_2(p)))$. Here, C is the raw bit-rate and p the bit-error rate.

1) Threat Model: Our threat model consists of two processes on the same machine without a legitimate communication channel. Both processes share read access to two sets of files on the system. These could be, for example, shared libraries, which are readable by every process on the system. One set is used for synchronization, and it can be any size — we call this the synchronization file. The other set of files is used for the actual transmission, and a larger set allows sending more bits at once — this is the transmission file. In our experiments, we use a transmission file size of 1 GiB. Unlike countless prior works [47], [54], [58], [62], [75], our covert channel does not need a high-precision clock shared between the two processes.

Finally, we assume that the transmitted data is represented by a similar amount of `0` bits and `1` bits. This is required to even out the highly unequal times it takes to read a page not in cache (50 µs) versus a page in cache (2 µs). All data can be converted to be balanced with small overheads, for example, using 8b/10b encoding [74]. We use a randomly generated stream of bits for our evaluation.

2) Flush-based Covert Channels: In a flush-based covert channel, the sender reads the whole transmission file to load all pages of the file into the page cache and flushes all pages corresponding to `0` bits in the data to send, other page correspond to `1` bits. Once complete, the sender reads the synchronization file to signal the receiver to read the message.

Flush+Monitor. The receiver monitors the synchronization file using `cachestat`, and upon seeing it in cache, it gets the cache status of every page of the transmission file. Our Flush+Monitor covert channel reaches 37.7 kB/s with an error rate less than $3 \times 10^{-4}\%$. Both the flush and monitor primitives are non-variable in nature, *i.e.*, a flush always happens, and a monitor always tells the page-cache state, leading to a noise-free and robust covert channel. Our transmission file size is 1 GB; with a much larger file whose size is comparable to free memory, it is possible for the kernel to begin evicting pages, causing the receiver to incorrectly perceive these as `0` bits.

Flush+Reload. The receiver times the reload primitive, `read`, to obtain whether the synchronization file is in the page cache. As this is destructive, the receiver has to flush the synchronization file after each measurement. If the sender reads the synchronization file between the read and flush of the receiver, the receiver misses the signal. Therefore, the sender performs the read 100 times. Afterwards, the receiver uses backward-reading technique presented in Section III-A to read the transmission file to learn the message.

Our Flush+Reload covert channel reaches 4.3 kB/s with an error rate less than $2.9 \times 10^{-3}\%$. We measure such a stark difference in speed between the Flush+Monitor and Flush+Reload covert channels because `cachestat` is significantly faster than `read` (1875 cycles vs 120 000 cycles).

Flush+Flush. The receiver uses the flush-timing side channel to obtain the cache state. After signaled, the receiver flushes the transmission file page-by-page to reconstruct the message. Our Flush+Flush covert channel reaches a raw transmission speed of 34.6 kB/s with an error rate of 1.8 %. This results in a true channel capacity of 33.9 kB/s

3) Evict-based Covert Channels: In an evict-based covert channel, the sender attempts to completely evict the transmission and synchronization files before communicating.

Evict+Monitor. We find that it takes all four eviction sets roughly 13 s (n=30, σ=0.8 s) on average to evict the entire transmission file, regardless of the transmission file's size from 128 MiB to 2 GiB. Eviction set 4 allocates 64 MiB chunks; although it could be faster to evict files if the chunk size was larger, we find that it often ends up being too big, slowing the system, and triggering the out-of-memory (OOM) killer. With 64 MiB chunks, we could always reliably evict the file without slowing the system down or running OOM.

The sender invokes eviction set 4 and ensures that the transmission file is evicted with `cachestat` before freeing eviction set 4 to release memory pressure. Afterwards, for every 1 the sender reads the corresponding page backwards through the file. Then, the sender reads the synchronization file to communicate that it is done. The receiver is identical to the one in Flush+Monitor. Our Evict+Monitor channel reaches

TABLE IV
FLUSH+MONITOR COVERT CHANNEL ON DIFFERENT HARDWARE

CPU	OS	Kernel	RAM	Capacity
Intel Ultra 7 265K	Ubuntu 24.04	6.14.0	8 GB	73.1 kB/s
Intel i9 13900KF	Ubuntu 22.04	6.8.0	64 GB	71.5 kB/s
Intel i5-11300H	Debian 13	6.12.43	16 GB	55.0 kB/s
Intel i5-8265U	Ubuntu 22.04	6.8.0	16 GB	35.9 kB/s
AMD Epyc 7313P	Ubuntu 22.04	6.11.0	64 GB	31.8 kB/s

TABLE V
FLUSH+MONITOR COVERT CHANNEL ON DIFFERENT FILESYSTEMS

File System	btrfs	ext4	ext3	ext2	ntfs
Channel Capacity	64.3 kB/s	54.99 kB/s	43.8 kB/s	44.4 kB/s	48.3 kB/s

Intel i5-11300H running Debian 13 (kernel 6.12.43) and 16 GB memory; Monitor mechanism: `cachestat`

1.4 kB/s ($< 8.9 \times 10^{-3}\%$ error rate), including the eviction time that all four eviction sets take at the beginning.

Evict+Reload. Unlike the Evict+Monitor covert channel, Evict+Reload cannot use monitor to determine whether a file is in cache. Instead, the sender must rely only on reload to determine whether a file is in cache. For this, the sender chooses N equidistant pages as "probing pages" of the file. These probes are reloaded to determine whether any part of the file is in the page cache. If even one probe is in the page cache, the sender must increase the memory pressure by employing eviction set 4. If all probes are not in the page cache, the sender assumes that the file is no longer in the page cache and stops applying memory pressure.

The sender uses multiple sets of N probing pages, as reading one set to determine the file's cache state makes its pages recently used and unlikely to be evicted. Therefore, after checking the cache state and increasing memory pressure, a new, not recently-used set is used to check the cache state. The sender repeats this, at most 64 times with $N = 16$ probe pages each. The sets are laid out to prevent read-ahead using backwards-reading (see Section III-A).

After evicting the file, the sender reads the corresponding page for every 1 to transmit, backwards through the file. Finally, it reads the synchronization file to signal the receiver. The receiver reloads pages of the synchronization file using the backward-reading technique. Upon seeing it in the page cache, it reads the transmission file. The receiver ignores the pages used by the sender to check if the transmission file is cached. Our Evict+Reload channel reaches a raw speed of 1.2 kB/s ($< 6 \times 10^{-4}\%$ error rate), including the eviction time that all four eviction sets take at the beginning.

Hardware Agnosticism. All our attack techniques are hardware and filesystem agnostic. In Table IV, we present the true capacity of our Flush+Monitor covert channel using `cachestat` on five different hardware. Table V presents the same covert channel on five files systems, all performed on the same machine. Systems with kernel versions before 6.5 do not support `cachestat`. However, it is still possible to employ `preadv2` with `RWF_NOWAIT` as a monitor mechanism

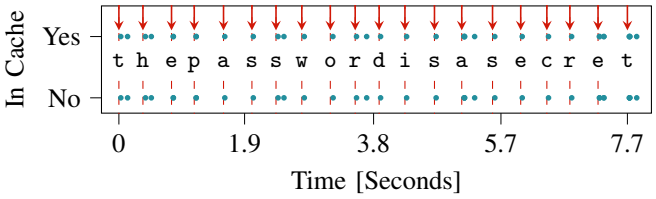

Fig. 8. For 7.7 s, the string "`thepasswordisasecret`" was typed into gedit while a malicious program employing the Evict+Monitor attack strategy monitored page 32 of `libgedit-41.so`, a shared library on which gedit depends. Each red, dashed line represents a key press while each blue dot is a change in the thirty-second page's presence in the page cache.

(on kernel versions after 4.14) for a Flush+Monitor covert channel while other non-monitor-based attacks still can be utilized.

On an Intel 7 Ultra 155U running Ubuntu 18.04.6 (kernel 5.4.0-84-generic) with 32 GB of memory, our Flush+Monitor covert channel has a capacity of 69.1 kB/s, while Flush+Reload reaches 5.1 kB/s. On an Intel i5-11300H running Debian 12 (kernel 6.1.0-39) with 32 GB of memory, the Flush+Monitor covert channel has a capacity of 37.3 kB/s, while Flush+Reload reaches 2.68 kB/s.

C. Observing Low Frequency Events

In this section, we show that our attack techniques can be used to observe singular low-frequency events like keystrokes for inter-keystroke timings, the starting of different processes for UI redress attacks, or actions within programs like Discord.

Like prior work [22], [66], we template pages of binaries and shared libraries to find code pages that are executed on the events we want to observe. Afterwards, we constantly remove the target page(s) from the cache and check if it was brought back into the cache. We know that the event code is executed when the page containing it is loaded back into the cache by the program we want to observe.

1) Threat Model: We assume an attacker is on the same machine and, for some attacks, inside a Docker container. The attacker wants to spy on user behavior, e.g., keystrokes, when applications are started, or events inside applications. The attacker has access to shared libraries and binaries but no other permissions. We assume there are no other paths or vulnerabilities that could be exploited.

2) Inter-Keystroke Timing: In this section, we present an inter-keystroke timing attack [22], [49], [59], [60], [62], [69], [79] using Evict+Monitor on `gedit`, a text editor. Keypress-timing information does not directly leak specific keys, but rather indirectly [69], [79]. We can only execute the inter-keystroke timing attack by means of eviction, as no pages corresponding to keys can be flushed. As the source of leakage, we target page 32 of `libgedit-41.so`. This page was determined by gradually increasing memory pressure while pressing keys and monitoring `libgedit` to see which pages are reloaded. Page 32 was the least noisy page corresponding to keystrokes.

With 401 human keystrokes pressed in 72 s (average: 5.5 keys/s), our Evict+Monitor attack loop observed an F_1 score of 96.7%. In Figure 8, we show the recording of an Evict+Monitor attack loop while a script using `xdotool` typed characters into gedit. The script typing the keys also recorded the precise timestamps of each key press, shown as red dashed lines. Changes in the page state of `libgedit` are depicted as blue dots. To ensure that all the keys are not pressed with uniform time between each key press, the script added a random delay between each key press. During this attack, the free space in memory as reported by the `free` command was between 450 MiB to 550 MiB.

As seen in Figure 8, there are thirteen single-observations (single in-cache observations) and seven double-observations (two rapid and successive in-cache observations). These double-observations are trivial to filter out as the minimum difference in time between two keystrokes (0.35 s) is greater than the difference in time between two observations corresponding to the same key (0.15 s). It is possible to misinterpret two different keystrokes as a double-observation (false negative) should the person be fast at typing. To obtain better inter-keystroke timings, we can collect typing traces multiple times, as shown in prior work [49], [65].

3) UI Redress Attack: In this section, we demonstrate that we can detect when a process started reliably with very short delay using Flush+Monitor. This can be exploited for a UI redress attack on a password authentication window where a user enters their password [4], [10], [18], [22], [34], [50], [61]. An attacker can detect such a window by detecting the presence of `pkexec` in the page cache. The `pkexec` program of PolicyKit [56] uses an authentication agent, e.g., UI password window, to authorize a user. Once `pkexec` has been detected in the page cache, an attacker can draw an identical-looking and fake password-authentication window over the real window. All seven pages of `pkexec` can be flushed when the program is not in use, and all pages are brought into cache when executing the program. On average, we detect the first page of `pkexec` in 800 ns (n=100, σ=60 ns).

Another target file to detect the start of `pkexec` is `libpwquality.so.1.0.2`, a shared library used by `pkexec`. We find that `libpwquality` is only used by three programs on a default Ubuntu installation: `sddm` or `gdm` (display managers), `sudo`, and `polkit-agent-helper-1` (spawned by `pkexec`). By monitoring the seventh page of `libpwquality` with Flush+Monitor, we detect the `pkexec` in 900 ns (n=100, σ=60 ns) on average.

4) Cross-Container Shell-Launch Detection: Boskov et al. [5] demonstrated page-cache attacks across Docker containers using Evict+Monitor. As noted in Section IV, the monitor primitive's mechanism, `cachestat`, is disabled by default in Docker containers [48]. Despite the lack of this primitive, we show that a container can still reliably use the Flush+Reload attack strategy on shared libraries to detect events in co-located containers. In particular, we show that a container can detect when another container starts a shell by flushing & reloading pages 45, 46, and 47 of the `libtinfo.so.6` shared library.

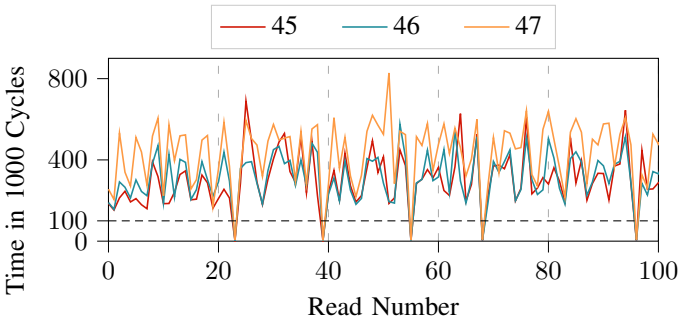

Fig. 9. A malicious Docker container can use Flush+Reload on pages 45, 46, and 47 of `libtinfo.so.6` to detect whether another Docker container launches. The reload-time for these pages considerably dips when another container launches — 4 000 cycles versus 350 000 cycles.

When shells are run, e.g., `bash` at container launch, they load `libtinfo` for terminal-related information.

We use a default Ubuntu 22.04 Docker container for our evaluation and find that it can flush and reload the three pages of `libtinfo` to successfully determine whether a shell was started in another container. In Figure 9, we plot the reload-times of pages 45, 46, and 47 of `libtinfo` as measured by a malicious container while a benign containers launches a shell five times. Reload times are consistently above 100 000 cycles, averaging around 350 000 cycles, while the shell is not launched. When the container spawns a shell, the time to reload drops significantly, averaging 4 000 cycles. With a simple threshold of 100 000 cycles, the malicious container detects all five times the benign container launched a shell.

Only relying on `libtinfo` comes with the risk of detecting false negatives. For example, if the user starts multiple `tmux` sessions, each session spawns a shell. However, this can be overcome by monitoring multiple shared libraries used when a container launches — effectively fingerprinting a container launch. A set of shared libraries can be determined via an offline templating analysis [66].

5) Events Inside Programs: In this section, we briefly show that we can also detect events inside running applications. In a more complex continuation of this attack, we perform website fingerprinting in the next section. Schwarzl et al. [66] attacked electron apps by combining CPU caches and the page cache. With our high-frequency page cache flushing primitive, we can perform attacks on electron apps with only the page cache. We target Discord 0.0.78, which requires and uses `/usr/share/discord/libffmpeg.so`. By constantly running Flush+Monitor on `libffmpeg`, we find three interesting observations: First, pages 174 and 887 are used whenever an embedded YouTube video starts playing. Second, Pages 190 to 221 are momentarily brought into cache when an internally-shared video plays, before being removed. Last, pages 936 to 974 are used whenever a sound is played, such as the sounds for deafen, mute, join voice chat.

TABLE VI
WEBSITE FINGERPRINTING RESULTS FOR EACH ATTACK TECHNIQUE.

Attack Technique	F_1 Score
Flush+Monitor	90.5 %
Flush+Reload	88.1 %
Flush+Flush	86.1 %
Evict+Monitor	79.5 %
Evict+Reload	37.9 %

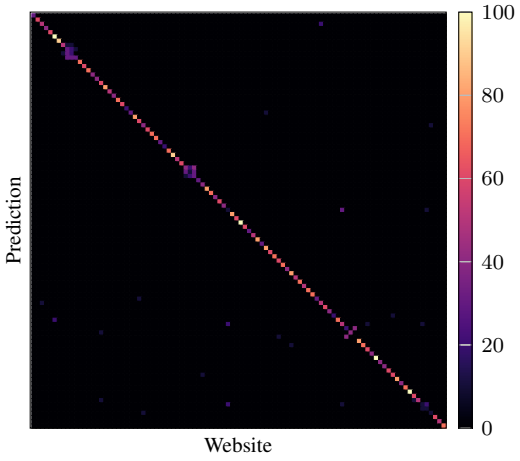

Fig. 10. Classification heat map of the website fingerprinting using Flush+Monitor. The F_1 score is 90.5 %.

D. Website Fingerprinting

In this section we present a website fingerprinting attack [33], [60], [71], [80], reaching an F_1 score of up to 90.5 % on the top 100 websites from the Alexa top 1 million list.

1) Evaluation Methodology: We exclude two of the top 100 websites due to issues with blocking and excessively slow loading times: jd.com and mediafire.com.

We target Firefox 133.0, which uses the shared library `libxul.so`, a file 41 205 pages large. Instead of keeping track of all 41 205 pages, we template the library in an offline analysis to narrow down the number of pages. As a part of this offline analysis, we launch each of the top 100 websites and monitor `libxul` using Flush+Monitor to find out which pages these websites use. We narrow down pages that are used by a maximum of any 80 websites, and this results in 7198 pages in our case, which is 17.47 % of the total library size. Therefore, we only track these 7 198 pages to determine which website the user is visiting. Any subsequent updates to `libxul` will require a reevaluation of the shared library.

For each of our five website-fingerprinting attacks, we open each website 30 times, recording the page cache behavior of our 7 198 target pages for 10 seconds, resulting in 3 000 measurements. We randomize the order in which websites are launched to mitigate possible unknown biases. After recording the data, we post-process it as described in Section VI-D2. This post-processing is the same for all attack techniques. Appendix A describes specific details of eviction-based website fingerprinting.

2) Data Processing: After recording the 7 198 pages of `libxul.so`, we convert the data into "activity strings". These strings contain the changes in the presence of each page in the page cache, '1' representing presence in the cache while '0' representing the opposite. If a page is accessed by the victim previously removed from page cache by the attacker and subsequently accessed by the victim again, the activity string here is "101". Since most websites only use few pages, we disregard any "0" activity strings and reduce them to an empty string, "", no activity. For classification, we pass these activity strings to a random forest classifier, the results of which shown in Table VI.

3) Results: We present our results in Table VI. Using flush to remove pages from page cache, we fingerprint websites with an F_1 score of at least 85 %. Out of the three flush-based attack techniques, Flush+Monitor performs the best with an F_1 score of 90.5 %, shown as a heat map in Figure 10. This is unsurprising as both the flush and monitor primitives are deterministic. While reload does provide a clear source of information (with a threshold of 200 000 cycles), there still are some fluctuations due to other system noise. With Flush+Reload we get an F_1 score of 88.1 %. Although using flush to leak whether a page is in cache has a close threshold (1 500 vs 5 000 cycles), we still get an F_1 score of 86.1 %.

The results for Evict+Monitor are comparable to the worst flush-based website fingerprinting attack, Flush+Flush. We see a sharp drop in F_1 score with Evict+Reload as the reloads add more pressure, making it harder to evict the target set of pages since these were recently accessed. Despite this, the F_1 score of Evict+Reload is still better than random guessing.

VII. COUNTERMEASURES

Mitigating all page cache attacks is a challenging endeavor that may or may not be desirable for a concrete system. Information provided by interfaces like `mincore` and `cachestat` can have substantial performance benefits in real-world use cases [68]. Furthermore, it may be possible for the kernel to introduce a new syscall which behaves as a flush, reload, or monitor mechanism.

In the case of flush and reload, we discussed the converging kernel functions in Section III-B and Section III-A respectively. Therefore, for an effective mitigation of flush and reload, these kernel functions must be restricted at a file and process level. However, most systems of individual users may not require flush to be available for all unprivileged processes. As the only filesystem agnostic mechanism for flushing is `posix_fadvise` with `POSIX_FADV_DONTNEED`, it may even be a viable method to make the advice privileged or require a capability. Updating the `posix_fadvise` system call alone would mitigate all flush-based attacks (Flush+Reload, Flush+Monitor, and Flush+Flush).

As `cachestat` shows, new syscalls can easily reintroduce leakage that undermines prior mitigation efforts. The *prevention* of monitor mechanisms is challenging as there is no clear convergence to a kernel function (see Section III-D), and thus resorting to mitigating known mechanisms on a case-by-case

basis. Since, we find that the amount of time `cachestat` requires to report a page's presence in the cache also leaks information (see Section III-D), this timing difference must also be considered when designing and mitigating syscalls.

With flush, reload, and monitor mitigated, only evict is to be mitigated. Eviction is not trivial to mitigate and may persist as explored extensively in secure CPU cache research [6], [19], [29], [57], [70]. However, mitigating reload and monitor eliminates both evict-based attacks: Evict+Monitor and Evict+Reload.

VIII. Conclusion

In this paper, we showed that the problem of page cache attacks is significantly larger than anticipated. Based on a systematic approach with four primitives (flush, reload, evict, and monitor) and five attack techniques (Flush+Monitor, Flush+Reload, Flush+Flush, Evict+Monitor, and Evict+Reload), following the terminology of CPU cache attacks, we mount attacks on fully up-to-date Linux kernels, bypassing existing mitigations. We demonstrate fast covert channels, inter-keystroke timing and event detection attacks, outperforming prior mitigated attacks by orders of magnitude, and a website-fingerprinting attack with an F_1 score of $90.54\,\%$. Our work shows that mitigating page cache attacks is more challenging than anticipated: While ad-hoc measures like making specific system calls privileged are necessary, an abundance of different primitives and techniques remain available to an attacker, inherent to the page cache being a cache. Thus, the problem is currently open and future research needs to find solutions that go beyond specific system calls.

Acknowledgments

We thank our anonymous reviewers across conferences and our shepherd for their invaluable feedback and guidance. We thank Smriti Suresh, Stefan Gast, and Xufan Zhao for their inputs at various stages of this work. This research is supported in whole or in part by the the European Research Council (ERC project FSSec 101076409), the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85), the Austrian Science Fund (FWF project NeRAM 10.55776/I6054), and the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant numbers 888087). Additional funding was provided by generous gifts from Intel, Red Hat, and Google. 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] O. Acıiçmez, J.-P. Seifert, and c. K. Koç, "Predicting secret keys via branch prediction," in *CT-RSA*, 2007.

[2] D. Alden, "Finding locking bugs with Smatch," 2025. [Online]. Available: https://lwn.net/Articles/1023646/

[3] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[4] A. Bianchi, J. Corbetta, L. Invernizzi, Y. Fratantonio, C. Kruegel, and G. Vigna, "What the app is that? deception and countermeasures in the android user interface," in *S&P*, 2015.

[5] N. Boskov, N. Radami, T. Tiwari, and A. Trachtenberg, "Union Buster: A Cross-Container Covert-Channel Exploiting Union Mounting," in *International Symposium on Cyber Security, Cryptology, and Machine Learning*, 2022.

[6] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches," in *MICRO*, 2020.

[7] N. Brown, "Sparse: a look under the hood," 2016. [Online]. Available: https://lwn.net/Articles/689907/

[8] ——, "Readahead: the documentation I wanted to read," 4 2022. [Online]. Available: https://lwn.net/Articles/888715/

[9] C. Chen, J. Cui, G. Qu, and J. Zhang, "Write+Sync: Software Cache Write Covert Channels Exploiting Memory-disk Synchronization," *TIFS*, 2024.

[10] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks." in *USENIX Security*, 2014.

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

[12] J. Corbet, "A CLOCK-Pro page replacement implementation," 8 2005. [Online]. Available: https://lwn.net/Articles/147879/

[13] ——, "The multi-generational LRU," 2021. [Online]. Available: https://lwn.net/Articles/851184/

[14] ctags, *ctags*, 2023, https://docs.ctags.io/en/stable/man/ctags.1.html.

[15] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *MICRO*, 2016.

[16] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *ASPLOS*, 2018.

[17] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *CCS*, 2000.

[18] Y. Fratantonio, C. Qian, S. P. Chung, and W. Lee, "Cloak and dagger: from two permissions to complete control of the UI feedback loop," in *S&P*, 2017.

[19] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss, "Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks," in *USENIX Security*, 2023.

[20] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU." in *NDSS*, 2017.

[21] L. Groot Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme," in *CHES*, 2016.

[22] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, "Page Cache Attacks," in *CCS*, 2019.

[23] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *S&P*, 2018.

[24] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[25] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security*, 2015.

[26] C. Gu, Y. Zhang, and N. Abu-Ghazaleh, "I know What You Sync: Covert and Side Channel Attacks on File Systems via syncfs," in *S&P*, 2025.

[27] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice," in *S&P*, 2011.

[28] B. Gülmezoğlu, M. S. Inci, T. Eisenbarth, and B. Sunar, "A Faster and More Realistic Flush+Reload Attack on AES," in *COSADE*, 2015.

[29] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *MICRO*, 2017.

[30] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! A fast, Cross-VM attack on AES," in *RAID*, 2014.

[31] Q. Jiang and C. Wang, "Sync+Sync: A Covert Channel Built on fsync with Storage," in *USENIX Security*, 2024.

[32] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." in *USENIX ATC*, 2005.

[33] J. Juffinger, F. Rauscher, G. La Manna, and D. Gruss, "Secret Spilling Drive: Leaking User Behavior through SSD Contention," in *NDSS*, 2025.

[34] J. Juffinger, H. Weissteiner, T. Steinbauer, and D. Gruss, "The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer," in *DIMVA*, 2025.

[35] M. Kerrisk, "Filesystem notification, part 1: An overview of dnotify and inotify," 7 2014. [Online]. Available: https://lwn.net/Articles/604686/

[36] P. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.
[37] J. Kosina, "make mincore() more conservative," 2019. [Online]. Available: https://github.com/torvalds/linux/commit/134fca9063ad4851de767d1768180e5dede9a881
[38] B. W. Lampson, "A note on the confinement problem," *Communications of the ACM*, 1973.
[39] J. Lawall and G. Muller, "Coccinelle: 10 Years of Automated Evolution in the Linux Kernel," in *USENIX ATC*, 2018.
[40] Y. Lee, J. Kwak, J. Kang, Y. Jeon, and B. Lee, "PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique," in *USENIX Security*, 2023.
[41] Linux, "filemap_read," 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.8/core-api/mm-api.html#c.filemap_read
[42] ——, "64-bit system call numbers and entry vectors," 2025. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/entry/syscalls/syscall_64.tbl
[43] ——, *readv(2) — Linux manual page*, 2025, https://man7.org/linux/man-pages/man2/preadv.2.html.
[44] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss, "Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors," in *AsiaCCS*, 2020.
[45] L. Maar, S. Gast, M. Unterguggenberger, M. Oberhuber, and S. Mangard, "SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel," in *USENIX Security*, 2024.
[46] L. Maar, J. Juffinger, T. Steinbauer, D. Gruss, and S. Mangard, "KernelSnitch: Side-Channel Attacks on Kernel Data Structures," in *NDSS*, 2025.
[47] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.
[48] moby/profiles, "seccomp/default.json: Default Seccomp Policy," 2025. [Online]. Available: https://github.com/moby/profiles/blob/main/seccomp/default.json
[49] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.
[50] M. Niemietz, "Analysis of UI Redressing Attacks and Countermeasures," Ph.D. dissertation, Ruhr University Bochum, Germany, 2019.
[51] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.
[52] Y. Patel, C. Ye, A. Sinha, A. Matthews, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift, "Using Trāṭr to tame Adversarial Synchronization," in *USENIX Security*, 2022.
[53] C. Percival, "Cache Missing for Fun and Profit," in *BSDCan*, 2005.
[54] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security*, 2016.
[55] N. Pham, "[PATCH v13 2/3] cachestat: implement cachestat syscall," 2023. [Online]. Available: https://lore.kernel.org/lkml/20230503013608.2431726-3-nphamcs@gmail.com/
[56] polkit, *polkit*, 2025, https://polkit.pages.freedesktop.org/polkit/.
[57] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic Analysis of Randomization-based Protected Cache Architectures," in *S&P*, 2021.
[58] A. Purnal and I. Verbauwhede, "Advanced profiling for probabilistic Prime+Probe attacks and covert channels in ScatterCache," *arXiv:1908.03383*, 2019.
[59] M. Qiu, L. Chuang, D. Kim, H. Qu, T. Chen, and A. Kwong, "KeyTAR: Practical Keystroke Timing Attacks and Input Reconstruction," in *S&P*, 2026.
[60] F. Rauscher, A. Kogler, J. Juffinger, and D. Gruss, "IdleLeak: Exploiting Idle State Side Effects for Information Leakage," in *NDSS*, 2024.
[61] C. Ren, Y. Zhang, H. Xue, T. Wei, and P. Liu, "Towards Discovering and Understanding Task Hijacking in Android," in *USENIX Security*, 2015.
[62] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds," in *CCS*, 2009.
[63] S. Rostedt, "ftrace - Function Tracer," 2017. [Online]. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt
[64] A. Ruggia, A. Possemato, A. Merlo, D. Nisi, and S. Aonzo, "Android, Notify Me When It Is Time To Go Phishing," in *EuroS&P*, 2023.
[65] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks," in *NDSS*, 2018.
[66] M. Schwarzl, E. Kraft, and D. Gruss, "Layered Binary Templating," in *ACNS*, 2023.
[67] C. Shen, J. Zhang, and G. Qu, "MES-attacks: Software-controlled covert channels based on mutual exclusion and synchronization," in *DAC*, 2023.
[68] J. Snyder, "Re: [PATCH] mm/mincore: allow for making sys_mincore() privileged," 1 2019. [Online]. Available: https://lore.kernel.org/lkml/5c3e7de6.1c69fb81.4aebb.3fec@mx.google.com/
[69] D. X. Song, D. Wagner, and X. Tian, "Timing Analysis of Keystrokes and Timing Attacks on SSH," in *USENIX Security*, 2001.
[70] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu, "Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It," in *S&P*, 2021.
[71] R. Spreitzer, S. Griesmayr, T. Korak, and S. Mangard, "Exploiting data-usage statistics for website fingerprinting attacks on Android," in *ACM Conference on Security & Privacy in Wireless and Mobile Networks*, 2016.
[72] Y. Tsunoo, T. Saito, and T. Suzaki, "Cryptanalysis of DES implemented on computers with cache," in *CHES*, 2003.
[73] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *CCS*, 2015.
[74] A. X. Widmer and P. A. Franaszek, "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code," *IBM Journal of research and development*, 1983.
[75] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud," in *USENIX Security*, 2012.
[76] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.
[77] J. Young, "NSA tempest documents," *CRYPTOME*, 2002.
[78] J. Zhang, C. Shen, and G. Qu, "Mex+Sync: Software Covert Channels Exploiting Mutual Exclusion and Synchronization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.
[79] K. Zhang and X. Wang, "Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems," in *USENIX Security*, 2009.
[80] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels," in *USENIX Security*, 2023.
[81] X. Zhang, Y. Xiao, and Y. Zhang, "Return-oriented flush-reload side channels on arm and their implications for android devices," in *CCS*, 2016.
[82] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *CCS*, 2014.

APPENDIX A
EVICTION-BASED WEBSITE FINGERPRINTING

To effectively evict our target pages, we have to dynamically apply memory pressure by means of eviction set 4. The memory consumption of Firefox is highly dependent on the opened website (see Section III-C), and thus it is insufficient only to apply a static pressure. Hence, we dynamically control memory pressure by employing eviction set 4.

To control memory pressure, we use the number of pages of our target file, `libxul`, that are not in the page cache as our base metric, determined by using the monitor primitive. When the percentage of pages not in cache is lower than the specified memory pressure, we allocate more memory to increase the memory pressure and evict more pages. Conversely, when the percentage is greater, we free memory. For example, a memory pressure of 10 % causes 10 % `libxul`'s pages to be evicted.

Naturally, a higher memory pressure causes the system to become unstable. We were able to reliably achieve a maximum memory pressure of 30 %. At memory pressures greater than 30 %, we consistently notice the Out-of-Memory (OOM) killer terminate our memory-pressure program. Therefore, we present the results below for eviction-based website fingerprinting attacks at 30 % memory pressure. To keep results

consistent between runs, we read the entire `libxul` file before beginning the recording.

Data Processing. We truncate the activity strings to length 8, since any more page cache state changes are artifacts of the eviction pressure. These artifacts have the unwanted effect of making the data very sparse, and thus, we find truncation to reduce noise. We also disregard any single-length activity strings ('0' or '1') as we found that these made the data noisy and the classification worse.

APPENDIX B
ARTIFACT APPENDIX

Page cache attacks have been thought to be mititgated or impractically slow since the mitigation of the Linux `mincore` syscall in 2019. In this paper, we not only revive practical attacks on the page cache, but also provide a systematic classification & understanding of primitives which interact with page cache. This understanding helps us advance page cache attacks, including speeding up previously known primitives by six orders of magnitude.

This artifact demonstrates the four primitives which we describe in our paper – flush, reload, evict, and monitor –, two methods to overcome the read-ahead mechanism, syscalls determined to be reload mechanisms, covert channels, and proof-of-concept attacks. Our attacks are hardware-agnostic and primarily depend on the versions of the Linux kernel used. We recommend evaluating our attacks on kernel versions above 4.14, ideally above 6.5. In February 2025, one of the monitor mechanisms, the `cachestat` syscall was mitigated; patches have been rolled out across Linux distributions, and the exact patch number can be checked to determine when it was patched. However, `cachestat` is not strictly required for our evaluation, as we demonstrate our attacks using another monitor mechanism, `preadv2 + RWF_NOWAIT`, introduced in kernel 4.14 and still working.

A. Description & Requirements

1) How to Access: Our Zenodo artifact is available at https://doi.org/10.5281/zenodo.17915256, made conveniently available at https://github.com/isec-tugraz/Eviction-Notice.

2) Hardware Dependencies: We tested all our attacks on x86 desktop / notebook CPUs. To keep evaluation times short, we suggest testing on Intel and AMD CPUs with 8 GB to 32 GB of RAM.

3) System Requirements: Linux kernel versions above 4.14 and ideally above 6.5. We developed and tested all our artifacts on Debian 13 and Debian-based Linux distributions (Ubuntu 22.04). Although our attacks should work on most Linux distributions, distribution-specific idiosyncrasies might make it challenging to mount attacks in the exact manner described in this artifact. One of the syscalls on which we rely as a monitor mechanism is `cachestat`. This syscall has been mitigated in February 2025, and no longer works in the cross-user threat model of our paper. Therefore, it may be necessary to downgrade the kernel for testing `cachestat` alone. However, we have another monitor mechanism, `preadv2 + RWF_NOWAIT`, which works with kernel versions above 4.14. Swapping should be switched off to keep evaluation time short. We suggest an ext4 filesystem and glibc version $\geq$ 2.34 is required. Finally, systems with `/tmp/` mounted as `tmpfs` will not be able to carry out the specific implementation of this artifact.

4) Attack Requirements: Creating another user is required to completely evaluate our attacks in the cross-user threat model. To create another user requires root permissions.

5) Software Dependencies: None

6) Benchmarks: None

7) Security, Privacy, and Ethical Concerns: Our artifacts do not perform any malicious operations by design. The only evaluation which *may* be destructive is eviction-based evaluation: if aggressive values are chosen, it may cause the system to become unstable or even crash, requiring a forced restart. For eviction-based operations, we recommend closing applications, saving important data, and disabling swapping. All other operations are non-destructive.

B. Artifact Installation & Configuration

To configure a system in which `cachestat` (monitor mechanism) works in a cross-user threat model, please check the Debian Security Tracker for when different distributions patched it and revert to that patched kernel. Our monitor-based attacks are configured to use an alternative with `preadv2` if setting up an unpatched kernel requires too much time.

C. Experiment Workflow

1) Create a new user: `sudo adduser testinguser`
2) Download the zipped artifact file and unzip its contents to `/path/`.
3) Run the script: `/path/helper-scripts/-create-testing-file.sh`
4) **Calibration**: Compile code in `/path/calibration/` and run to compute three thresholds.
5) Set these thresholds across:
   - `/path/primitives/Makefile`
   - `/path/covert-channels/flush-reload/Makefile`
   - `/path/covert-channels/flush-flush/Makefile`
   - `/path/attacks/docker/watcher/watcher-flush-_reload/Makefile`
6) **Primitives**: Compile code in `/path/primitives/` and evaluate all four primitives
7) **Reload Mechanisms**: Compile code in `/path/-reload-mechanisms/` and evaluate all eleven syscalls & methods.
8) **Covert Channels**: Compile code in the three directories of `/path/covert-channels` and evaluate all three covert channels.
9) **Attacks (1)**: Evaluate a Flush+Monitor attack to detect when a binary is run.
10) **Attacks (2)**: Build two docker containers (using docker compose) and evaluate a Flush+Reload attack to detect when a binary is run cross-container.

11) **Attacks (3)**: Using an apt version of Firefox, evaluate a Flush+Monitor attack to theoretically mount a website fingerprinting attack.

D. Major Claims

- (C1): Our work advances page cache attacks by systematically classifying flush, monitor, reload, and evict primitives. These four are outlined in Section III-A, III-B, III-C, III-D and are proven by experiment (E2).
- (C2): In our work, we find two novel techniques to overcome a well-known limitation with reload: the read-ahead mechanism. This is proven by experiment (E2) and discussed in Section III-A: the backward-reading technique and `readahead` syscall.
- (C3): Our work revives page cache attacks and improves state of the art by 6 orders of magnitude. Discussed in Section V-A, this is proven by experiment (E4) with a Flush+Monitor covert channel based on `preadv2` and `RWF_NOWAIT`.

E. Evaluation

For each experiment, we provide in-depth instructions in the artifact's singular `README` file. The structure of the `README` follows the experiment order in this appendix.

1) Experiment (E1): [Calibration] [10 human-minutes]: Determining timing thresholds

[Preparation] The testing file should be created with `create-testing-file.sh`. In `calibration/`, execute `make`.

[Execution] Run the three compiled binaries in `build/`, each with one argument: path to testing file. Run these binaries 5-10 times to gauge a good threshold.

[Results] The binaries report `FLUSH_THRESHOLD`, `READ_THRESHOLD`, and `READAHEAD_THRESHOLD`. Note a threshold for each which appears most of the time (a rough number is fine). These will be set in the Makefiles in experiment (E2).

2) Experiment (E2): [Primitives] [50 human-minutes]: Validating primitives

[Preparation] Set the determined thresholds from experiment (E1) in `primitives/Makefile`. Afterwards, run `make` and copy the build folder to `/tmp/build-primitives`. Ensure swapping is switched off.

[Execution] As testinguser, execute the binaries in `/tmp/build-primitives` to test flush, reload, monitor, and evict (detailed instructions in the `README`). Afterwards, test the read-ahead mechanism bypasses and eviction. **Warning**: Eviction may cause instability and possibly crashes.

[Results] Flush and evict should be able to remove pages from the page cache, while reload brings it back into the page cache. Monitor reports the presence of pages in cache. `monitor-cachestat` may fail if the mitigation is in place.

3) Experiment (E3): [Reload Mechanisms] [10 human-minutes]: Validating 11 syscalls and methods as reload mechanisms.

[Preparation] Run `make` in `reload-mechanisms`.

[Execution] Run `test-all-reload.sh`.

[Results] This experiment tests 11 syscalls and methods as reload mechanisms by flushing, reloading, and reloading again for each syscall and method. On average, there should be a notable timing difference between between `[Flushed]` and `[After Reload]`.

4) Experiment (E4): [Covert Channels] [20 human-minutes]: We demonstrate three out of five covert channels. Notably, Evict+Monitor and Evict+Reload are not provided to save time. Eviction is a challenging primitive on unknown systems, especially when run repeatedly. If eviction was made to work in experiment (E2), we claim that performing a reload or monitor preceded by an eviction is a rudimentary demonstration of an Evict-based covert channel.

[Preparation] Set the determined thresholds from experiment (E1) in each of Makefiles in `covert-channels/`. Afterwards, run `make` in all three directories and copy the build folder to `/tmp/covert-channels`. Generate a synchronization and communication file.

[Execution] One after the other, run each receiver as testinguser and the sender as the normal user. The Flush+Flush covert channel may be quite challenging for achieving a high accuracy as the threshold for flush is quite small. Nevertheless, we offer three suggestions in Flush+Flush section of the README to help achieving a higher accuracy.

[Results] Each receiver of the covert channel reports a percentage of data correctly transmitted. Both the sender and receiver report time taken to transmit and receive.

5) Experiment (E5): [Attacks] [45 human-minutes]:

[Preparation] Set the determined thresholds from experiment (E1) in the Makefile of `attacks/docker/watcher/watcher-flush_reload/Makefile`. Run `docker compose` in the `watcher` and `provoker` directories. Install an apt version of Firefox.

[Execution] First, run a flush and while-loop of monitor checking for the presence of `htop` in cache, and then run `htop`. Second, run the two docker containers. Start the program in watcher first watching `ls`, and execute `ls` in provoker. Finally, launch a Flush+Monitor loop on Firefox's `libxul.so`, launch Firefox, and visit 5 websites.

[Results] The first part reports `htop`'s presence in cache upon running `htop`. The second part reports a much lower timing when `ls` is executed cross-container. The third part reports a different range of values in `libxul.so`'s page count upon visiting different websites.