

NoCAP: Practically Defending Page Cache Attacks

Sudheendra Raghav Neela, Marcell Haritopoulos, Hannes Weissteiner,
Fabian Rauscher, Stefan Gast, Daniel Gruss
Graz University of Technology

***Abstract*—Operating-system page caches are shared across users, containers, and processes, introducing hardware-agnostic side channels on all major operating systems. Recent attacks observe page cache residency through direct monitor interfaces, reload and flush timing, and indirectly through eviction pressure. However, straightforward defenses have been rejected as impractical: making all relevant interfaces privileged or changing their semantics would break long-standing userspace behavior, while making ordinary file accesses constant-time or removing the page cache altogether would impose prohibitive overheads. As a result, page-cache attacks remain effectively unmitigated in practice.**

In this work, we present Non-observable Cache Accesses for Page Caches — NoCAP —, a practical, userspace-preserving defense against page-cache side channels on Linux. NoCAP targets the root-cause leakage primitives of page-cache side channels: it treats direct monitor interfaces as explicit cross-domain information flows, hides reload timing with a per-process first-time miss strategy, and hides flush timing with an adaptive delay that preserves the behavior of `posix_fadvise` with `POSIX_FADV_DONTNEED`. We evaluate NoCAP on 8 Linux systems with asynchronous readahead and mmap readaround disabled across Intel and AMD consumer and server CPUs, DDR4 and DDR5 memory, and multiple storage devices. Across our performance benchmarks, NoCAP incurs 0.48 % overhead in SPEC CPU 2017 Integer Rate and an 0.21 % average overhead with Redis benchmark (maximum 6 %), and a $4.29\times$ slowdown when launching Docker containers. With the Flexible I/O tester (FIO) benchmark, NoCAP incurs 2.05 % average overhead with synchronous reads (maximum 18 % on an HDD machine), and 1.65 % average overhead with AIO and io_uring reads (maximum 46.4 % overhead on an HDD machine). We show that residual flush-event leakage is limited to applications using `POSIX_FADV_DONTNEED`. The remaining eviction-only channel leaks only aggregate page-cache occupancy: in an idealized Evict+Evict experiment over 10 victim working-set sizes, median eviction times differ by only 36.8 ms across classes while the average interquartile range is 165 ms, and a best-case binned classifier reaches only 18.9 % single-sample accuracy versus a 10 % random baseline, corresponding to 0.23 bits per sample or approximately 0.08 bits/s.

I. Introduction

Page-cache attacks have evolved from interface-specific monitoring to a broader primitive-based attack surface. Gruss et al. [15] showed that an attacker can infer victim activity from page-cache residency, using page-cache eviction with `mincore` on Linux and `QueryWorkingSetEx` on Windows; Linux subsequently restricted `mincore` to mitigate cross-process page-cache monitoring [24]. Follow-up work showed that the same leakage extends across containers [3], that `preadv2` with `RWF_NOWAIT` can replace `mincore` as a page-cache monitor [36], and that hardware features of some SSDs can accelerate eviction and reload-based observations [21]. Recently, Neela et al. [27] showed that the attack surface is significantly larger than anticipated, classifying page-cache interactions into four primitives: *monitor*, which directly reports page-cache residency; *reload*, which leaks residency through access time; *flush*, which removes pages and leaks residency through flush time; and *evict*, which removes pages indirectly through memory pressure. Among these primitives, monitor, reload, and flush expose page-cache residency directly or through timing. Linux now limits the `mincore` and `cachestat` system calls as direct monitor oracles [24], [40], while the `preadv2` system call with `RWF_NOWAIT`, an equivalent monitor interface, is still available; reload and flush remain harder to address as they leak information via timing while providing ordinary, userspace-visible, expected behavior.

Straightforward mitigations against these leakage root causes are not practical: making flush privileged would remove the most efficient setup step for Flush+Monitor and Flush+Reload, and eliminate the timing source of Flush+Flush, mirroring common mitigation strategies and discussions for flush-based CPU cache attacks [12], [16]. However, applications legitimately use `posix_fadvise` with `POSIX_FADV_DONTNEED` to drop file data from the page cache, and Linux kernel maintainers repeatedly expressed that at least some of these system calls must remain available even to unprivileged applications [1], leaving substantial attack surface open. Similarly, making ordinary reads constant-time or removing the page cache altogether would change a core performance property of the operating system with highly prohibitive overheads. Consequently, page-cache attacks remain effectively unmitigated in practice with no clear path forward: a deployable defense must significantly reduce leakage without removing the system call and kernel behavior responsible for the root-cause leakage, but one which applications depend on.

In this paper, we present NoCAP, ***N**on-**o**bservable **C**ache **A**ccesses for **P**age Caches* (illustrated in Figure 1), a practical, userspace-preserving defense against side-channel attacks on the Linux page cache. NoCAP is based on a simple obser-

Network and Distributed System Security (NDSS) Symposium 2027
22–26 March 2027, Seoul, Republic of Korea
ISBN 978-1-970672-09-1
https://dx.doi.org/10.14722/ndss.2026.231028
www.ndss-symposium.org

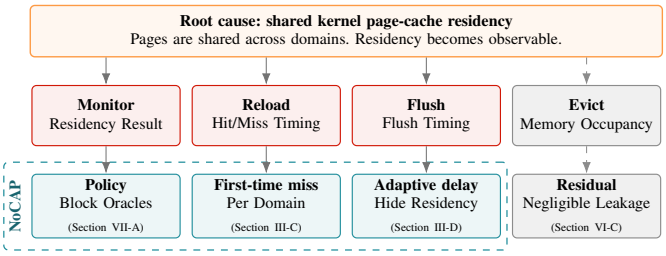


Fig. 1. NoCAP mitigates page-cache attacks at the root cause by removing cross-domain observability of page-cache residency while preserving ordinary page-cache behavior. Section references point to the corresponding design or analysis details.

vation: page-cache operations do not have to be removed to stop page-cache attacks; their cross-domain observability has to be removed. NoCAP therefore mitigates the three root-cause primitives that reveal page-cache residency: For **monitor**, NoCAP treats direct page-residency reports as explicit information flows that must not cross security domains; existing restrictions for `mincore` and `cachestat` are first instances of this policy [24], [40], showing that the Linux kernel security team deemed page cache attacks significant enough to deploy some fixes. For **reload**, NoCAP employs the idea of first-time miss described for CPU caches in TimeCache [28], making the first access to a page indistinguishable from a cache miss for each process, thereby eliminating the cross-domain timing signal used by Flush+Reload and Evict+Reload while preserving read semantics within a domain. For **flush**, NoCAP introduces an adaptive delay that makes flush runtime depend on the request within a domain's state, similar as the reload operation, instead of actual page-cache residency, mitigating Flush+Flush while preserving the existing behavior of `posix_fadvise` with `POSIX_FADV_DONTNEED`.

We implement a NoCAP prototype on Linux and evaluate the security and performance impact with asynchronous readahead and mmap readaround disabled in two steps: **First**, we evaluate NoCAP in CPU and I/O benchmarks to measure the performance cost of eliminating the root-cause leakage while maintaining userspace-visible behavior. Across 8 systems with Intel and AMD CPUs and DDR4 and DDR5 memory, NoCAP introduces a negligible overhead of 0.48 % in SPEC CPU 2017 Integer Rate, an 0.21 % average overhead with Redis benchmark (maximum 6 %), 2.05 % synchronous-read average overhead with the Flexible I/O Tester (FIO) benchmark, 1.65 % AIO- & io_uring-reads average overhead with FIO, and 4.29× slowdown when launching Docker containers. On average, our largest overheads are on a machine with an HDD, which incurs a maximum 18 % overhead with synchronous reads and 46.4 % overhead with AIO and io_uring reads. **Second**, we analyze the remaining page-cache attack surface with NoCAP: Since NoCAP does not disable flushing, an attacker could still determine whether a process flushed a set of pages. However, this residual flush-event signal is limited to applications that use `POSIX_FADV_DONTNEED`; in Debian Code Search, we find 1 235 references to `POSIX_FADV_DONTNEED` across 174 packages, of which 89 packages directly use the flag. We quantify the remaining leakage of the only remaining primitive (evict) in an Evict+Evict channel, where an attacker utilizes the non-deterministic and unpredictable eviction primitive to infer aggregate page-cache occupancy. In an ideal laboratory environment with 10 victim working-set sizes and 100 randomized trials per size, median eviction times differ by only 36.8 ms across classes, while the average interquartile range is 165 ms; a best-case binned classifier reaches 18.9 % single-sample accuracy versus a 10 % random baseline, corresponding to 0.23 bits per sample or approximately 0.08 bits/s, which is 6 to 9 orders of magnitude lower than without NoCAP [15], [27] and weaker than ordinary memory-usage interfaces such as `free` or `top` when those are available. Thus, eviction alone does not provide enough leakage for targeted page-residency attacks; it exposes at most a weak aggregate occupancy signal that many systems already expose more directly. Overall, NoCAP demonstrates that page-cache attacks can be mitigated at their root-cause leakage sources while keeping userspace-visible behavior intact, making it a practical deployable mitigation.

Contributions. In summary, our main contributions are:

1) We introduce NoCAP, a practical, userspace-preserving page-cache defense targeting the root-cause page-cache leakage primitives while preserving functionality.
2) We mitigate root-cause monitor, reload, and flush primitives: direct monitor oracles are treated as explicit cross-domain information flows, reload timing is hidden with a per-process first-time miss strategy, and flush timing is hidden with an adaptive delay, that preserves `posix_fadvise` with `POSIX_FADV_DONTNEED`.
3) We evaluate our NoCAP prototype on 8 Linux systems, measuring 0.48 % SPEC CPU 2017 Integer Rate overhead and 2.05 % average FIO synchronous-read overhead, and 1.65 % average AIO- and io_uring-read overhead.
4) We quantify the residual attack surface, with only 89 packages having remaining flush-event leakage, and eviction-only leakage only yielding aggregate occupancy at 0.23 bits per sample (0.08 bits/s) under ideal settings.

Outline. Section II provides background on side channels, cache defenses, and page-cache attacks. Section III presents the high-level design of NoCAP. Section IV describes implementation details. Section V evaluates the performance impact of NoCAP on 8 systems. Section VI analyzes the remaining attack surface. Section VII discusses practical deployment, limitations, and related work. Section VIII concludes.

II. Background

In this section, we provide background on the Linux page cache, cache side channels and defenses, and page-cache attack techniques relevant to NoCAP.

A. Operating System Page Cache

The page cache stores disk-backed pages, including memory-mapped files, files accessed via file operations, pro-

gram binaries, and shared libraries. It is software-managed and size-limited by available memory. Writing to a shared file-backed page either requires write permissions or follows copy-on-write semantics. Linux tracks page-cache pages with recency information, and evicts cold pages when memory pressure requires it [5], [6], [7], [19].

Linux frequently works with groups of pages consisting of a power-of-two number of contiguous pages, called folios [9]. Folios allow the kernel to support file systems with block sizes larger than the page size, and to optimize IO operations by triggering fewer, larger IO requests. The page cache stores files as one or more folios, depending on their size.

Several kernel interfaces interact with the page cache in ways that are relevant for side channels. Accessing a file brings uncached file data into memory. `posix_fadvise` with `POSIX_FADV_DONTNEED` allows applications to mark pages as unneeded, enabling the kernel to drop them from the page cache. These operations can leak residency only through timing. By contrast, monitor interfaces such as `mincore`, `cachestat`, and `preadv2` with `RWF_NOWAIT` reveal residency directly or through an equivalent explicit status result.

B. Cache Side Channels and Defenses

Cache side channels exploit changes in the cache state to infer victim activity, most commonly through timing [16], [23], [45] or contention [14], [29]. The more prominent cache side channels are Prime+Probe [29], Flush+Reload [45], Flush+Flush [16], and Evict+Reload [17], which exploit either timing differences in memory accesses or flushing, to observe cache set contention. Cache defenses usually follow one of three strategies. First, they remove operations that enable attacks or make them privileged, e.g., by making flush unavailable to unprivileged code or preventing flush instructions from executing [12], [16], [31], or removing timing primitives [16]. While this approach can make attacks more challenging, it often does not eliminate the root cause of the leakage and could therefore be bypassed by replacing the restricted operation with a different functionally equivalent operation, e.g., replacing *clflush* with an eviction set [17]. Second, they eliminate the timing differences an attacker measures, e.g., making operations constant time or close to constant time [18], [41]. Third, they statically or dynamically split the shared cache by security domain, ranging from strict partitioning to randomized or remapped cache placement [11], [13], [26], [30], [35], [43], [44]. First-time miss follows this third strategy: it treats the first shared-memory hit after a domain change always as a miss, thereby splitting timing behavior by domain rather than removing sharing entirely [28], [33]. These ideas were developed for CPU caches, which are fast, set-associative, and, apart from a few instructions, largely transparent to the instruction-set architecture level. In contrast, the page cache is far slower and fully associative. It is also exposed through the ABI via different system calls (syscall). Removing a syscall or changing its behavior can break existing applications, which should be avoided whenever possible. Consequently, the mitigation constraints for the page cache differ from those of hardware caches.

C. Page Cache Attacks

Knowing whether a page is cached can be useful for benign performance-sensitive applications [38]. The same information can also leak victim activity because file-backed pages, especially of executables and shared libraries, are commonly shared across security boundaries. Gruss et al. [15] demonstrated the first page-cache attacks on Linux and Windows. On Linux, their attack used eviction followed by `mincore` to monitor page residency; on Windows, it used `QueryWorkingSetEx`. Later, `mincore` was restricted to mitigate cross-process page-cache monitoring [24]. Boskov et al. [3] demonstrated page-cache attacks across Docker containers.

After the `mincore` mitigation, page-cache attacks remained possible through other mechanisms. Schwarzl et al. [36] showed that `preadv2` with `RWF_NOWAIT` can act as a monitor mechanism, returning early when data is not in memory. Juffinger et al. [21] showed that SSD host memory buffer behavior can accelerate eviction and reload-based observations. Most recently, Neela et al. [27] systematized page-cache attacks into four primitives: monitor, reload, flush, and evict. These primitives can be combined into attack techniques known from hardware cache attacks: Flush+Monitor, Flush+Reload, Flush+Flush, Evict+Monitor, and Evict+Reload.

Reload. The reload primitive measures the time required to access a target page. If the page is cached, the access is fast; otherwise, the kernel has to fetch the page from storage, which takes longer. Thus, reload leaks page-cache residency through timing and is destructive, as the measurement itself brings the page into the page cache. Prior work identified multiple syscall paths that behave as reload mechanisms and traced them to common filemap and readahead paths in the kernel [27].

Flush. The flush primitive removes clean, unmapped pages from the page cache. On Linux, `posix_fadvise` with `POSIX_FADV_DONTNEED` provides a filesystem-agnostic flush mechanism used by legitimate applications to tell the kernel that file data can be dropped. Neela et al. [27] showed that the runtime of flush depends on whether the target page is cached, enabling Flush+Flush-style attacks like in CPU caches [16]. Thus, flush both prepares attacks by removing pages and leaks information directly through timing.

Monitor. Monitor primitives directly report page-cache residency and therefore do not rely on timing. Examples include `mincore`, `cachestat`, and `preadv2` with `RWF_NOWAIT` [15], [27], [36]. Crucially, monitor primitives do not bring the target page into the cache during residency checks.

Evict. Eviction removes pages indirectly by applying memory pressure. Unlike flush, eviction is not deterministic for a single target page; the kernel decides which pages to reclaim based on recency, memory pressure, and page state. Eviction is therefore slower and noisier than flush, but it remains relevant when flush is unavailable or mitigated [15], [21], [27].

Attack Techniques. Page-cache attack techniques combine a primitive that removes a page from the cache with a primi-

tive that observes cache residency. Flush+Reload and Evict+Reload observe the reload timing after flush or eviction. Flush+Monitor and Evict+Monitor observe page residency through a monitor mechanism after flush or eviction. Flush+Flush uses the flush timing itself as the observation primitive. Consequently, the leakage sources relevant for a generic defense are direct monitor oracles, reload timing, and flush timing.

III. NoCAP Design: High-Level Overview

In this section, we provide a high-level design of our defense against side-channel attacks on the page cache, which we call NoCAP: **N**on-**o**bservable **C**ache **A**ccesses for **P**age Caches. We first explain the threat model and attacker capabilities which NoCAP defends. Afterwards, we discuss the principles that guide the design of our defense. Finally, we describe how NoCAP mitigates the underlying primitives that were used in prior page cache attacks: monitor, reload, and flush [27]. While monitor is handled with simple policies, we describe high-level strategies for mitigating reload and flush timing leakage.

A. Threat Model

Following prior work's attacker model [27], we assume an unprivileged attacker who wishes to leak data about another user on the same system. The attacker has local code execution and can execute system calls that leak whether or not pages are cached. The attacker targets globally readable files such as executables, shared libraries, and data files. The presence of such pages in the page cache indicates that another process (possibly cross-user) accessed them. We assume the attacker can use a high-resolution timer, choose page ranges for syscalls, and perform enough repetitions to compute minima, medians, or averages, *i.e.*, NoCAP relies neither on reduced timer precision nor on preventing averaging (see Section IV-E). For our remaining Evict+Evict attack, we finally assume much tighter restrictions, discussed and evaluated in Section VI-C.

We define our security domain as the unit of isolation across which page-cache residency must not be observable. Within a domain, residency and the timing that reveals it may be observed freely, since the executions in a domain can already observe each other's file accesses directly. Across domains, residency must not be inferable from any interface or from the timing of any page-cache operation. In our Linux prototype (detailed further in Sections III-C and IV), a domain is one address space, *i.e.*, one `mm` struct, identified by a freshly allocated `mm` token rather than the raw pointer, so address-space reuse cannot inherit another domain's state. Threads share an `mm` and thus a domain, while forked, exec-ed, and cross-container processes each get a distinct one. We consider attacks from within the same address space to be irrelevant for page cache attacks. Since cross-thread attackers can already manipulate more than just the page cache, e.g., write to `/proc/self/task/<tid>/mem`, they therefore do not have to resort to page cache attacks.

B. Principles

Instead of mitigating specific attack techniques or syscalls, NoCAP mitigates the underlying leakage primitives that enable these techniques. Current attack techniques rely on four primitives: monitor, reload, flush, and evict. Evict removes pages from the cache through memory pressure, which is inherent to caches with limited capacity. Monitor, reload, and flush are the three relevant primitives that leak page-cache residency: monitor leaks directly, whereas reload and flush leak through timing. Figure 1 gives an overview of the root-cause mitigation strategy. As a principled and generic defense, NoCAP should mitigate the leakage at its source, forming the basis of NoCAP:

> **Principle 1: Mitigate the root cause.** NoCAP must address leakage where page-cache state is shared and exposed: inside the kernel.

Although some defensive techniques exist for userspace applications, they have limitations. Userspace programs could defend themselves against the monitor, reload, and flush primitives with private copies of the same file, albeit at the cost of larger disk and memory usage. To additionally prevent eviction, applications would have to lock the pages of *all* critical files (binaries, shared libraries, and data files) in memory, e.g., by using `mlock`. With multiple processes, possibly using different library versions, the memory footprint would increase significantly. While deduplication could help, it would also enable other side-channel attacks [4], [37]. Generally, mitigations in userspace would be implemented in every single application, which contradicts our goal of a generic defense that protects existing code without modifications.

Even when mitigated by the kernel, kernel maintainers and developers are cautious not to break userspace [8], [25]:

> **Principle 2: Preserve page-cache semantics.** Legitimate reads, mappings, and advisory flushing must keep their userspace-visible behavior.

This principle applies to ordinary page-cache functionality, such as reads, memory-mapped access, and advisory flushing. Direct monitor interfaces are different: their purpose or observable status directly exposes page-cache residency across domains, so NoCAP treats them as explicit information flows that must be restricted by policy (see Principle 4).

Prior work [27] suggested making flush (*i.e.*, the `posix_fadvise` syscall with the `POSIX_FADV_DONTNEED` flag) privileged, as it would mitigate three attack techniques: Flush+Monitor, Flush+Reload, and Flush+Flush. However, kernel maintainers are reluctant as this might break legitimate use cases [1]. Indeed, there are well-known applications using `POSIX_FADV_DONTNEED` for precisely removing a file's pages from cache: for example, Redis flushes to drop rarely accessed files from the page cache after accessing them [34]. BorgBackup flushes recently-read pages for file backups to increase performance [42]. For NoCAP to be realistically implemented, it must not change behavior expected by legitimate userspace applications. Since flushing is a syscall, we cannot change the semantics of it to remove its ability to flush.

However, forcibly removing pages from the page cache itself does not leak information by itself. In the case of the

Flush+Flush attack which relies only on flush, the leakage arises from the *time* taken to flush a page from cache, which tells the attacker whether the page was cached or not:

Principle 3: Remove observability, not functionality. NoCAP must hide page-cache residency without prohibiting ordinary cache fills, flushes, or evictions.

Out of the four primitives constituting attack techniques, only monitor, reload, and flush leak page cache residency. Just removing a page from the page cache, via evict or flush, does not leak information, unless an attacker times it in the case of flush. Therefore, NoCAP mitigates the three primitives that leak page-cache residency: monitor, reload timing, and flush timing. The two timing channels require mechanism-specific defenses: reload timing is hidden with per-domain first-time miss state, while flush timing is hidden with an adaptive delay (described later). Monitor mechanisms are simpler: because they explicitly report residency, NoCAP requires that they are not available for untrusted cross-domain observations:

Principle 4: No cross-domain residency oracles. Interfaces that directly report page-cache residency must not be available across security domains.

NoCAP considers mechanisms that directly report the presence of a page in the cache to be security violations. Known monitor mechanisms include `mincore`, `cachestat`, and `preadv2` with `RWF_NOWAIT` [27]. After the disclosure, `mincore` and `cachestat` were restricted to only write-privileged users [24], [40]. Similarly, NoCAP introduces restrictions to `preadv2` with `RWF_NOWAIT`, e.g., by gating the non-blocking status across security domains or by disabling the monitor behavior for untrusted callers. The same rule applies to future interfaces that would otherwise reintroduce a direct page-residency oracle. This monitor component is part of NoCAP's idea; because monitor is an interface-level restriction, the paper focuses its technical detail for the three-primitive defense on making reload and flush timing unexploitable while preserving ordinary page-cache behavior. We discuss monitor mechanisms in Section VII-A.

C. Mitigating Reload: First-Time Miss

To mitigate reload timing, we use a first-time miss strategy like prior CPU cache defenses [33], namely TimeCache [28]. When a process first accesses a cached page, the kernel intentionally delays the access to mimic a disk read, resetting this behavior if the page is evicted. The first-time miss domain is the task's `mm` struct, and thus, threads share an `mm` state, while forked, exec-ed and cross-container processes are independent.

For example, process A's first access to a page is a real disk read, while later accesses come from the page cache directly. If process B then loads the same cached page for the first time, the kernel delays it to simulate a disk fetch, preventing B from discerning cache or disk fetch. B's subsequent accesses are fetched from the page cache without a time delay.

Evicting or flushing a page resets first-time miss state for all processes that touched it, via the existing invalidation path. Eviction, flushing, and truncation clear tracking bits, reads synchronize via folio locking, and inode teardown releases tracking structures. A solution to avoid stale state across address-space reuse is to key first-time miss state by a fresh `mm` token rather than by a raw `mm` pointer.

D. Mitigating Flush: Adaptive Delay

NoCAP's third principle states that it should mitigate the sources of information leakage, namely page cache residency. In the case of flush, the page-cache-state information leakage stems from the *time* it takes to flush a page. Prior work [27] suggested introducing a capability check [22] to mitigate flush. However, this change would require existing programs to adapt, violating NoCAP's second principle: userspace semantics must not be changed. Instead, NoCAP keeps flushing available and hides the timing signal with a delay. The goal is not to make every flush take one constant time since `posix_fadvise` ranges can span one or many pages, and the caller chooses this range. Instead, NoCAP makes flush constant-time for each requested range. For two `posix_fadvise` calls over ranges of the same length, the observable runtime should depend on the request size, but not on whether none, some, or all of the requested pages were resident. The adaptive delay thus targets the worst-case timing envelope for each range, covering fully cached, uncached, and mixed-residency ranges.

IV. Implementation Details

In this section, we describe the technical details of implementing NoCAP. We discuss how we implement first-time miss tracking efficiently, and how we store the tracking data structures in the kernel. NoCAP mitigates monitor, reload, and flush; our prototype implements the reload and flush timing components. The monitor component is an interface-policy component of NoCAP: existing restrictions for `mincore` and `cachestat` satisfy this rule for those interfaces, and NoCAP applies the same rule to `preadv2` with `RWF_NOWAIT` and future equivalent interfaces. Because this component changes oracle availability rather than common page-cache read or flush paths, the implementation and performance evaluation focus on the reload and flush timing components.

A. Data Structures

NoCAP aims to mitigate the reload primitive in the page cache by introducing a first-time miss strategy. To implement this strategy, we need to track whether a process has previously accessed a page, *i.e.*, whether it has already experienced a miss penalty. Thus, we need to keep track of accessed pages for each process while minimizing memory overhead. We track accessed pages using a bitfield, where each bit corresponds to a folio in the page cache. We store the bitfield in an `xarray`, a dynamic array data structure in the Linux kernel optimized for sparse entries and efficient lookups. Each 64-bit entry in the `xarray` can store the accessed state of 63 folios, ensuring minimal memory overhead. We store the `xarrays` in a maple tree data structure, which we call `i_mm_access_bits`, in the file's inode data structure. As the key, we use a randomly

generated token that uniquely identifies an `mm` instance. Thus, multiple file descriptors and all tasks sharing the same virtual address space, e.g., threads created with `clone(CLONE_VM)`, share the same accessed state. Each process with its own address space (forked, exec'ed, or containerized) gets an independent first-time miss state.

B. Page Access

We now describe the three main scenarios of page accesses in the context of NoCAP: The first access, loading a page for the first time; further accesses, simulating a page cache miss; and eviction, which resets the first-time miss state of all pages.

First access. When a file is accessed for the first time, the kernel reads its data from disk and places it in the page cache, returning only when the data is available. The process triggering the first access experiences a real miss penalty, as the data is not yet cached. The bitfield entries for that process are set to one, indicating that the process has accessed those folios. On subsequent accesses to the same folios, the process receives the data directly from the page cache without a miss penalty. No bits are set for other processes.

Further accesses. When a different process accesses parts of a file that are already in the page cache, the kernel checks `i_mm_access_bits` to determine whether the process has previously accessed those folios. If the corresponding bitfield entry is zero, the kernel adds a synthetic delay to the access, simulating a miss penalty, and then sets the bitfield entry to one. Section IV-C describes two strategies for generating the delays and discusses their trade-offs. On subsequent accesses, when the corresponding bitfield entry is set to one, the process retrieves the data directly from the page cache without a miss penalty. Although accesses via `read` and `mmap` trigger different code paths (`mmap` loads pages via page faults instead of syscalls), we apply first-time miss everywhere, ensuring that the mitigation is consistent across access methods.

Eviction. When evicting a folio from the page cache, the kernel resets bitfield entries for all processes. Thus, the next access incurs a miss penalty, as the kernel treats it as a first-time access. If an attacker evicts a previously accessed folio, they cannot distinguish whether it was reloaded by the victim.

C. Delay Strategy

To simulate a miss penalty, we first consider a simple timing-only baseline and then use a mimic-I/O strategy in the implemented and evaluated NoCAP configuration.

Baseline intuition: naive delay. A straightforward baseline is to introduce a random delay between 25 µs to 100 µs for each synthetic cache miss. We implemented this baseline with an uninterruptible timeout to prevent the process from returning early due to interrupts, but we do not use it in the evaluated NoCAP configuration. The reason is that a fixed delay interval is distinguishable from real disk access times, which vary with the underlying storage device and system state. Furthermore, because the delay does not access the disk, it does not create storage-device contention. Since contention causes a measurable increase in disk access latency [20], an attacker could distinguish a synthetic miss from a real miss by measuring access latency to a different file on the same storage device. Thus, naive delay is useful for understanding the trade-off, but not as the security mechanism we evaluate.

Implemented approach: mimic I/O delay. The evaluated strategy performs an actual read operation from the disk to mimic a real miss penalty, albeit without creating a new page cache entry. We implement this by passing a mimic mapping to the synchronous readahead code, which performs the IO operations as usual, on the correct block device. After the synchronous readahead returns, we discard the mimic mapping, and return the data from the page cache, setting the corresponding bitfield entries to one. This strategy is more secure than the naive delay, as it closely mimics a real miss penalty, including the variability in access times based on the storage device, contention effects, and even other factors such as SMART events. However, it is more expensive than the naive delay, as it slows down parallel disk accesses.

D. Limitations

The limitations of our current Linux implementation are prototype coverage limitations, not fundamental limitations of the NoCAP design. The prototype covers the page-cache fill paths used by our security experiments and the buffered synchronous workloads in the evaluation. A hardened configuration must either route remaining page-cache fill paths through the same first-time miss logic or disable those paths until they are covered. After such coverage, the fundamental residual attack surface is eviction-only behavior: advisory flushing still removes pages, but without the flush timing oracle. Section VI quantifies this residual attack surface.

MMAP Readaround and Asynchronous Readahead. For memory-mapped files, the kernel speculatively loads pages into the page cache using readahead, hoping to reduce future access times. While not effective for processes that sparsely access files, the readahead mechanism can significantly speed up continuous file accesses. Similarly, asynchronous readahead reads not-yet-accessed pages of files into the page cache in the background [27], triggering the next readahead IO operation whenever an asynchronously-read page is accessed. Both of these readahead mechanisms cause pages to be loaded into the page cache without a corresponding access from a process. Thus, NoCAP's first-time miss strategy needs to be applied to these asynchronous page loads as well, to prevent them from being used as a side-channel primitive. Our prototype does not yet cover these speculative fills, as they require integration with additional page-cache paths beyond the synchronous readahead path. In the evaluated configuration, asynchronous readahead and MMAP readaround are disabled in both the baseline and NoCAP kernels, so the comparison is fair, isolating the cost of the implemented mechanisms rather than claiming full-production Linux coverage.

Other Limitations. Transparent huge pages are a kernel feature that allows the kernel to use huge pages for memory management without explicit support from applications. They are not covered by our prototype because they require first-

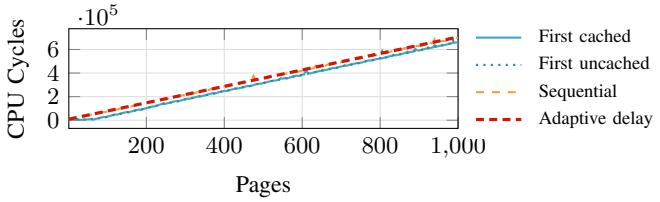


Fig. 2. Adaptive flush timing calibration on $M7$ for requests from 1 to 1000 pages. The x-axis is the number of pages in a flush request and the y-axis is elapsed CPU cycles. The cached and uncached series differ only in whether the first target page is resident; the fitted adaptive-delay line stays above both series, making runtime depend on request size rather than residency.

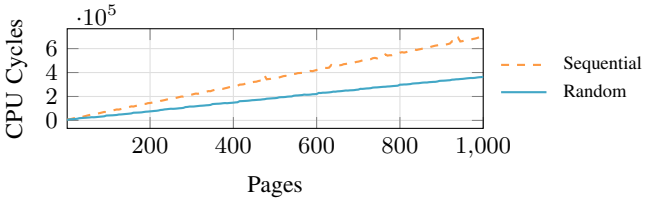


Fig. 3. Flush timing on $M7$ when the number of flushed pages is sampled sequentially or randomly. Both curves report average CPU cycles as a function of pages in the request. Sequentially increasing requests form the higher timing envelope, the conservative case used for selecting adaptive flush delays.

time miss state at huge-page granularity, or a split into ordinary folios at the protection boundary.

E. Adaptive Flush

As discussed in Section III, the timing of a flush depends both on the number of pages involved and on how many of those pages are already present in the cache. In this section, we develop a heuristic approach to determine an adaptive flush delay for a given number of pages. We characterize the fitting procedure on an AMD Ryzen 9 9950X (Zen 5) running Ubuntu 24.04.3 with 16 GB DDR5 RAM and a 2 TB NVMe SSD ($M7$ in Table I). The fitted parameters shown for $M7$ are only used for $M7$. For every other evaluated system, we rerun the same calibration locally and load that system's own slope and intercept before the experiments. Changing the hardware, filesystem, kernel configuration, or memory-pressure regime requires repeating this calibration.

We previously outlined two possible scenarios when flushing multiple pages: either all pages are cached, or only a subset is cached. For the latter case, we assume an attacker is still primarily interested in leaking the cache residency of a single page within the set. Accordingly, we model a simplified setting in which the first page is either cached or uncached, and the attacker's objective is to distinguish between these two states. We evaluate both scenarios 256 times for up to 1 000 pages, and the results are shown in Figure 2.

Across the 201 tested request sizes, the median absolute difference between the cached and uncached averages is 850 cycles, or 0.35 % of the corresponding mean timing, and the 90th percentile is 2 694 cycles, or 2.96 %. Thus, in this calibration experiment, single-page residency is not a strong timing factor. Instead, the order and repetition of flush operations have a stronger impact on flush execution time: sequentially increasing requests are slower than random-size requests for all 201 tested sizes, with a median gap of 166 709 cycles. In particular, repeatedly flushing the same set of pages results in lower reload times on average, *i.e.*, a typical Flush+Reload attack monitoring a set of pages. Choosing a random size of pages to flush every time also results in lower reload times. However, across our experiments, we find that issuing a flush on a sequentially increasing number of pages requires the most time for a given number of pages, shown in Figure 3.

This observation is important for the design of NoCAP's adaptive flush delays. If the delays are determined by repeated flushes or random flushes, NoCAP may not properly account for sequential flushes. In the case where an attacker performs infrequent or random flushes on a set of pages, they could observe lower flush times than expected and infer that the pages were not cached, since truly cached flushes under such conditions would exhibit longer flush times. Therefore, the adaptive delay must account for this scenario to avoid inadvertently introducing a distinguishable timing difference.

We measure flush time for 1 to 1 024 pages, repeating each 256 times, and fit a constrained least-squares line per system, shown in Figure 2 as the "Flush Adaptive Delay". The fit is constrained to stay above the minimum measured flush time at every page, plus a margin equal to the smallest observed gap between any average and minimum measurement, leaving enough room for an attacker to be unsure whether the observed time is a truly cached or deliberately uncached page.

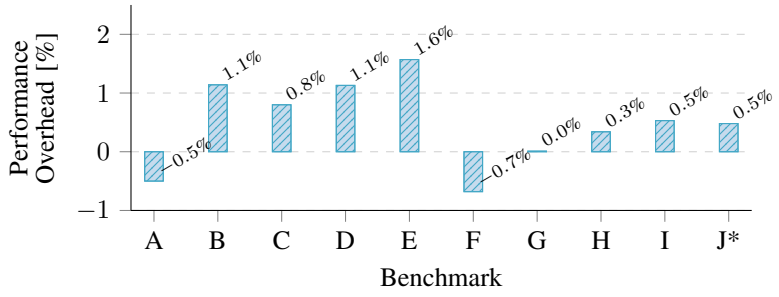
We assume a local attacker with a high-resolution timer, attacker-chosen `posix_fadvise` ranges, and enough repetitions for minima, medians, or averages. NoCAP thus does not rely on reduced timer precision or preventing averaging, but instead, it targets the lower timing envelope left after filtering scheduler and other positive noise. Flushes below the calibrated target are delayed to that envelope, a value set by requested page count instead of the residency. Slow tails from scheduling or I/O contention remain observable but are not stable cached-versus-uncached oracles here.

In our prototype, the final parameters of this constrained linear regression (slope and intercept) are exposed through sysctl and loaded after local calibration, before running untrusted workloads. This sysctl interface is an engineering interface rather than a security policy: production should run calibration as a trusted boot-time or installation-time step and rerun it when the CPU, storage stack, filesystem, kernel configuration, or configured memory-pressure target changes.

At runtime, NoCAP applies the calibrated line to each `POSIX_FADV_DONTNEED` request. The kernel derives the request size from the `posix_fadvise` length, using the full file size when the length is zero, and computes the target runtime as the calibrated intercept plus the request size multiplied by the calibrated slope. It then performs the actual flush, measures the elapsed cycles, and delays the caller for the remaining time if the flush completed before the target. Thus, for a fixed

TABLE I
LIST OF EXPERIMENTAL MACHINES

Machine	CPU	Distribution	Memory	Disk
$M1$	AMD Ryzen 7 7700X (Zen 4)	Ubuntu 22.04.4	32GB DDR5	1TB NVMe **SSD** - Kingston KC3000
$M2$	AMD Ryzen 7 7700X (Zen 4)	Ubuntu 22.04.4	32GB DDR5	1TB SATA **HDD** - WD Caviar Green - 8MB Cache
$M3$	Intel Core i7-1355U (13th Gen)	Ubuntu 22.04.5	40GB DDR4	512GB NVMe **SSD** - WD SN740 (SDDPMQD)
$M4$	AMD EPYC 8024P (Zen 4)	Ubuntu 22.04.5	32GB DDR5	2TB NVMe **SSD** - Samsung SSD 980 PRO
$M5$	Intel Xeon Silver 4514Y	Ubuntu 24.04.4	128GB DDR5	2TB NVMe **SSD** - Kingston NV2
$M6$	AMD EPYC 9015 (Zen 5)	Ubuntu 24.04.4	32GB DDR5	1TB NVMe **SSD** - Samsung SSD 970 EVO
$M7$	AMD Ryzen 9 9950X (Zen 5)	Ubuntu 24.04.3	16GB DDR5	2TB NVMe **SSD** - Samsung SSD 990 PRO
$M8$	Intel Core i5-11300H (11th Gen)	Fedora 43	16GB DDR4	512GB NVMe **SSD** - Intel 670p



	SPEC Benchmark	Without NoCAP [s]	With NoCAP [s]	Overhead [%]
A	500.perlbench	204.7	203.7	-0.50
B	502.gcc	194.5	196.7	1.14
C	505.mcf	344.8	347.6	0.80
D	523.xalancbmk	168.0	169.9	1.13
E	525.x264	199.8	202.9	1.57
F	531.deepsjeng	199.8	198.4	-0.68
G	541.leela	293.1	293.1	0.01
H	548.exchange2	93.1	93.4	0.34
I	557.xz	318.7	320.4	0.53
J*	Average			0.48

Fig. 4. SPEC CPU 2017 Integer Rate overhead on $M7$, AMD (Zen 5) Ryzen 9 9950X, using six copies. Bars show runtime change with NoCAP relative to the baseline kernel, while the adjacent table lists runtimes. The mean overhead is $0.48\,\%$; the largest slowdown is x264 at $1.57\,\%$, and Deep Sjeng shows a $-0.68\,\%$ change.

request size, the observable flush time is padded to the same target regardless of how many requested pages were actually resident in the page cache.

System or attacker-induced noise during adaptive-flush parameter calibration, *i.e.*, recording the time taken to flush, can be a detriment to the *system performance* after the regressed adaptive-delay line. Such noise can only increase the time it takes to flush during calibration as cache hits take longer and would be considered as cache misses, which overall pushes the regressed line *up*. This higher regressed line makes the system performance worse as now flush generally takes longer to execute, making it difficult for an attacker to discern whether or not a page is cached. System or attacker-induced noise during attacker exploitation serves as a detriment to the attacker, as it pushes the false negative rate up: cache misses are treated as cache hits. Therefore, for an attacker, general or induced noise is a detriment in either case.

V. EVALUATION

In this section, we evaluate the performance of the reload and flush timing components of the three-primitive NoCAP design on 8 machines, listed in Table I. These machines include x86 CPUs from both Intel and AMD, featuring consumer and server CPUs. We apply the patches from Section IV on a Linux kernel v6.18.2, compiled with any drivers that the machines require. Our evaluations are performed twice, once on the kernel without NoCAP, and once on the kernel with NoCAP. We first evaluate NoCAP with SPEC CPU 2017 Integer Rate, following which we evaluate NoCAP's performance with the Redis benchmark. Afterwards, we evaluate NoCAP on the Flexible I/O Tester (FIO), an "I/O workload tester" shipped with the Phoronix Test Suite [32]. We finally compare the performance of reading, flushing, and spawning Docker containers in Section V-D with and without NoCAP.

A. SPEC CPU 2017 Integer Rate

We first evaluate our defense on SPEC CPU 2017 Integer Rate [10]. This benchmark includes ten CPU-bound tests, and some have been patched to avoid file I/O, e.g., perlbench's SpamAssassin [39]. We exclude one of these tests, `520.omnetpp`, as it experienced a runtime error; the test simulates large network events. We present our results in Figure 4, with the evaluation performed on $M7$.

Since NoCAP is implemented at the kernel level and affects the interaction between the page cache, disk, and processes, we do not expect CPU-bound workloads to be majorly impacted. Indeed, we find that NoCAP has a minimal impact on CPU-bound workloads after running the benchmark on both kernels. The most extreme differences are Deep Sjeng experiencing a $-0.68\,\%$ overhead, *i.e.*, the test speeds up, and x264 slowing down by $1.57\,\%$. All other test cases are within these two extremes, with the mean of all test cases showing a negligible $0.48\,\%$ overhead. Our evaluation shows that NoCAP minimally affects CPU-bound tests.

B. Redis

We use the Redis benchmark as instrumented by the Phoronix Test Suite [32]. With 50, 500, and $1\,000$ parallel connections, the Redis benchmark performs 5 operations (GET, SET, LPOP, LPUSH, SADD) and measures the number of requests per second that Redis can handle. Figure 5 shows the

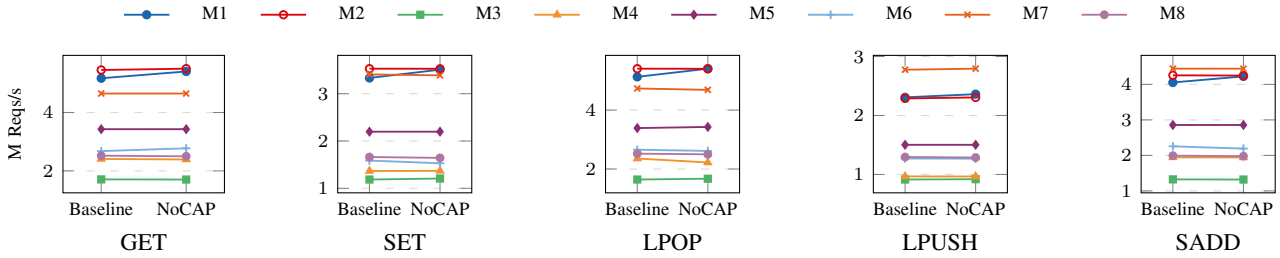


Fig. 5. Redis throughput per operation at 50 parallel connections, before (Baseline) and after (NoCAP), for every machine. Higher is better, so an upward line is an improvement. The machines differ by up to 6× in absolute throughput while the effect is below 6 %.

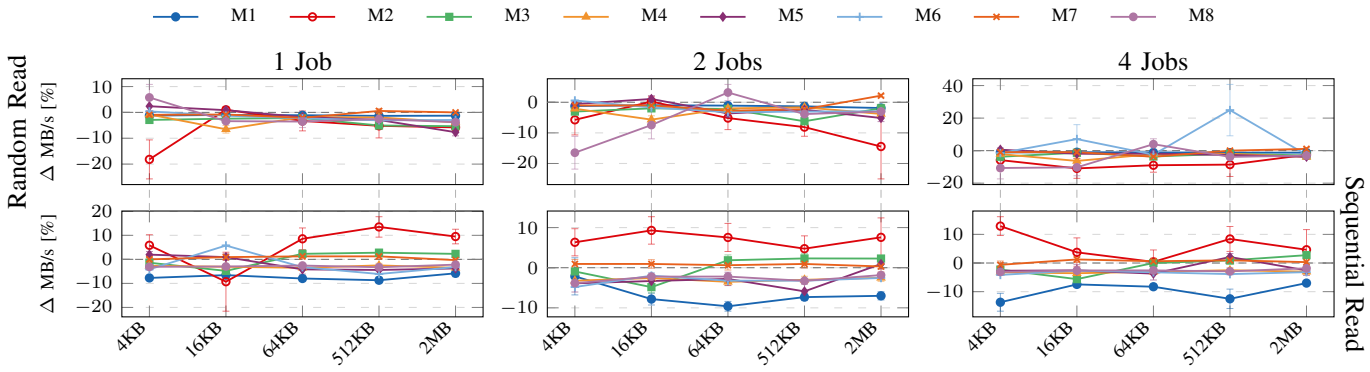


Fig. 6. Aggregate FIO **synchronous** throughput change with 1, 2, and 4 jobs across our systems. Lines show per-machine throughput change with NoCAP relative to the baseline kernel. Top plots are random reads while the bottom plots are sequential reads. Most configurations remain between 0 % to 10 %, with few larger deviations.

results on all of our machines with 50 parallel connections: the average change in overhead is 0.28 % while the maximum effect is −5.7 % on $M4$ (LPOP). In Appendix B, we present the remaining results with 500 and 1 000 parallel connections. With more connections, the numbers remain quite similar: −0.02 % average impact on performance with 500 parallel connections, and 0.37 % average impact with 1 000 parallel connections. The maximum performance impact with NoCAP is below 6 % in all three cases. Our evaluation overall shows that NoCAP minimally degrades Redis performance.

C. Flexible I/O Tester

The Flexible I/O Tester (FIO) simulates different I/O workloads [2], with configurable number of jobs, I/O engines, and block sizes. We run FIO as a test instrumented by Phoronix Test Suite with synchronous reads, AIO, and io_uring as engines. On all systems $M1$ to $M8$, we evaluate FIO's performance on kernels with and without NoCAP, running 1, 2, and 4 jobs, reading variable block sizes ranging from 4 kB to 2 MB in random or sequential order.

We present the results across all 8 systems in Figure 6 which shows the change in FIO's performance with the synchronous read engine. With **random synchronous** reads, we observe that NoCAP performs between 0 % to 10 % worse on most systems at all block sizes. In three exceptional cases, we find that NoCAP performs worse than 10 % with random reads: $M2$ at 4 kB blocks with 1 job (−18.19 %), $M8$ at 4 kB blocks with 2 jobs (−16.48 %), and $M2$ at 2 MB blocks with 2 jobs (−14.46 %). The other notable result with random reads is $M6$ performing 24.92 % better with NoCAP at 512 kB blocks with 4 jobs. Across our systems, NoCAP incurs an average 2.57 % overhead with FIO jobs reading blocks in a random order.

The results look similar with **sequential synchronous** reads too, with most of the overhead on all systems landing between 0 % to 10 %. The major notable exception is the system with the HDD, $M2$, whose sequential reading results are consistently between 0 % to 10 % *improvement* with NoCAP. We attribute this increase in performance to a combination of factors: less continuous pressure on the 8 MB internal HDD cache due to NoCAP stalling first-time misses, disabling the asynchronous readahead, and the HDD controller being able to optimize sequential reads around NoCAP's stalling design. Further affirming these factors is comparing $M2$'s sequential reading performance to its random reading performance, where it consistently performs between 0 % to 10 % worse with NoCAP. Random reads place pressure on the HDD, increasing average seek times. Across our systems, NoCAP incurs an average 1.52 % overhead with FIO jobs reading blocks in a sequential order. Across all job configurations and synchronous reading styles (random and sequential) on our 8 systems, FIO experiences a 2.05 % slowdown with NoCAP.

Figure 7 repeats this comparison for the two asynchronous engines, Linux AIO and the newer ring-buffer-based io_uring. Averaged over both engines, both read patterns, and all 8 sys-

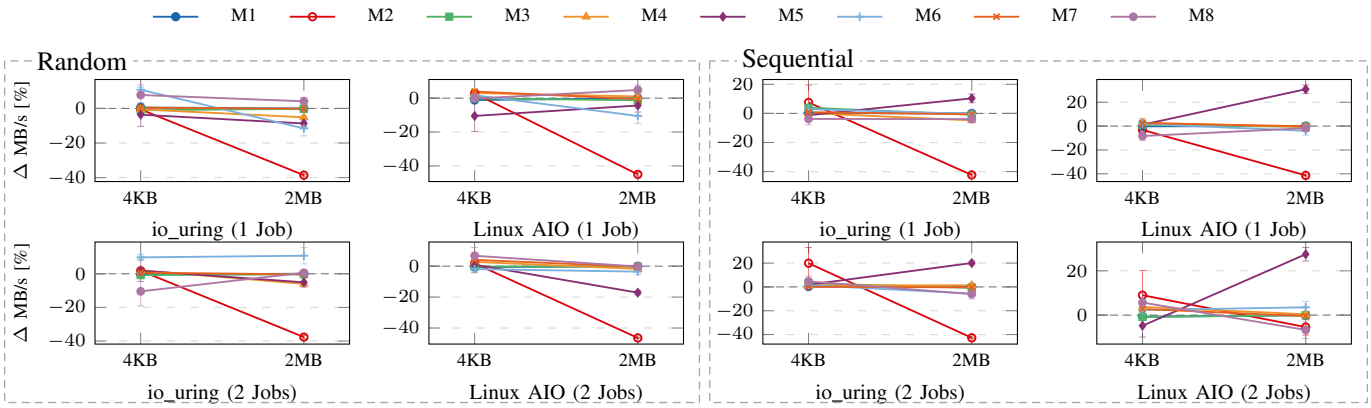


Fig. 7. Aggregate FIO throughput (AIO & io_uring) change with 1 and 2 jobs across our systems. Lines show per-machine throughput change with NoCAP relative to the baseline kernel. Most configurations remain between 0 % to 10 %, with two major deviations: $M2$ at 2 MB sizes, and $M5$ whose performance largely fluctuates.

tems, NoCAP incurs a 1.65 % overhead. Excluding $M2$, using AIO shows an average 0.9 % decrease in performance with random reads, and an average 2 % increase in performance with sequential reads. Using io_uring shows an average 0.18 % decrease in performance with random reads, and an average 0.86 % increase in performance with sequential reads. The notable outlier is $M2$ (HDD) at the 2 MB block size, where it loses between 37.7 % and 46.4 % in every configuration except sequential AIO with two jobs. However, at 4 kB the same machine shows no penalty at all. Since $M2$ is the only machine with an HDD, we attribute the large penalty in request timing to a combination of asynchronous I/O reading and the HDD seek time. In Appendix A, we show the performance of FIO on $M7$ across all tests with and without NoCAP.

We repeatedly tested the outliers mentioned earlier and found the results to consistently either converge to that outlier or be within that range. We attribute such outliers to storage-device-specific interaction with system addresses mapped to internal storage structures, buffers, or perhaps storage caches. Prior work [20] also appears to observe fluctuating timings with their burst sizes on SSDs, similar to our block size.

D. Read, Flush, and Docker Container Performance

Since NoCAP augments reload behavior with a first-time miss and flush behavior with an adaptive delay, we design our own microbenchmarks to measure the worst-case impacts of reloading and flushing under NoCAP. Our microbenchmark, run on all 8 systems, employs a coordinating process that spawns 8 worker processes, all of which interact with a 256 MB file. The coordinating process first flushes or reloads the file. It then instructs each worker process to perform an action on the file (flush or reload). Finally, we average the measured time across all processes. We also measure the impact and slow-down experienced when launching Docker containers with NoCAP enabled.

Testing Reload. The coordinating process first loads the entire file into the page cache, after which each process is instructed to load the file by iterating through pages. In this test, each process encounters NoCAP's first-time miss penalty, and thus we expect the average read time to be comparable to disk reads. We present the results of this test case in Figure 8. As expected, we observe large load times for the entire file, ranging from 1 200 % slower on $M6$ to 6 700 % on $M7$.

Testing Flush. The coordinating process reloads the entire file into the page cache before each process is instructed to flush it. These constitute the cached-file measurements. Afterwards, each process is instructed to flush the file when the file is not in page cache. We expect NoCAP to slow down these uncached flushes to times comparable to those of cached flushes.

Figure 9 shows that this indeed is the case. Five machines, $M1$–$M4$ and $M8$, are within 0.1 % of their flush timings, $M6$ is within 0.98 %, while $M5$ and $M7$ are within 3.4 % of their flush timings. NoCAP enforcing a delay on flush when the file is uncached undoubtedly results in a severe slowdown compared to the baseline kernel. Prior work [27] states that flush is on the order of a few thousand CPU cycles, and slowing it down to approximately 0.04 s in our case (from Figure 9) results in roughly 8 000 to 91 000 times slower flushes to mitigate the flush timing-leakage.

Testing Docker Container Start-Up Times. We test the impact of NoCAP on Docker container start-up times on $M7$ using the Ubuntu 24 Docker image. Without NoCAP, *i.e.*, the baseline kernel, the container requires 0.194 s (n=128, $\sigma_{\bar{x}}$=0.5 ms) on average to start. With NoCAP, the start-up time is on average 0.832 s (n=128, $\sigma_{\bar{x}}$=5 ms), a 4.29× impact, due to the shared libraries and large binaries required to start the Docker container (e.g., docker CLI, `containerd-shim-runc-v2`, `runc`).

VI. Security Analysis

In this section, we analyze the security of NoCAP. We first examine whether reload mechanisms and flushing interact with NoCAP, showing further that the threshold calibration to determine whether pages are cached is less than 0.15 µs. Second, we examine the information that still persists, namely an attacker can determine if a page was removed from cache,

reasoning why this information is not particularly useful to an attacker. Afterwards, we quantify the remaining eviction-only attack surface through an Evict+Evict experiment with 10 victim working-set sizes from 100 to 1 000 pages and 100 randomized trials per size. As prior work [15], [27] did not mention this attack technique, we discuss the Evict+Evict attack in Section VI-C before evaluating whether it can determine the number of pages that were loaded into the cache.

A. Verifying Page Cache Attacks Interact with NoCAP

We verify that page-cache attacks interact with NoCAP by building a tracing technique which prints to the kernel log whenever our mitigated functions are invoked, along with information about which function, process ID, and syscall called them. Flushing in a process reports at `generic_fadvise` function (implementation of adaptive flush), while reloading is reported at `filemap_get_pages` function (starting point of mimic I/O). This way, we verify that flushing and reloading the page cache interacts with our defense.

Furthermore, we perform threshold calibration for flush and reload timings with and without NoCAP on $M7$, using artifacts from prior work [27]. On a baseline system, attackers can derive a timing threshold, while with NoCAP active, timings become indistinguishable. On NoCAP, backwards-reading takes 41.076 µs (n=1000, $\sigma_{\bar{x}}$=33 ns) (uncached) and 40.937 µs (n=1000, $\sigma_{\bar{x}}$=48 ns) (cached), while flushing takes 96.177 µs (n=1000, $\sigma_{\bar{x}}$=24 ns) (uncached) and 96.143 µs (n=1000, $\sigma_{\bar{x}}$=15 ns) (cached). Flushing takes longer than reloading due to margin constraints on the regressed adaptive-flush line. Timings behave similarly across Docker containers (with 20 µs reload overhead). Overall, cached & uncached thresholds for both have margins below 0.15 µs.

We use these thresholds to mount three covert channels with the artifacts and communication strategies outlined by prior work [27]: Flush+Reload, Flush+Flush, and Evict+Reload. By using a file 512 MiB large, the sender transmits 16 KiB of information (as there are 16 384 pages). **Without NoCAP** and the thresholds calibrated for a baseline system, the covert channel receiver is able to accurately receive more than 96 % of bits across all three attack strategies: 99.994 % with Flush+Reload, 97.52 % with Flush+Flush, and 96.21 % with Evict+Reload. **With NoCAP** and the thresholds calibrated for NoCAP, the ability for the receiver to recover bits through the page cache state is largely hindered, and indeed, we see the receiver can recover only around 50 % to 51.5 % of bits — essentially random guessing: 51.33 % with Flush+Reload, 50.14 % with Flush+Flush, and 50.98 % with Evict+Reload.

B. Residual Flush-Event Leakage

The third principle of NoCAP is to mitigate sources of page cache residency leakage rather than mitigating methods to remove a page from cache (see Section III-B). One consequence of this is that although the flush timing-leakage is mitigated, the behavior of flush is not, *i.e.*, flush still reliably removes a page from the page cache. Eviction also removes pages from

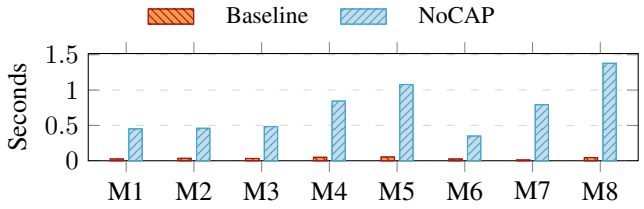


Fig. 8. Average first-time miss penalty when reading a cached file across 8 processes. The first-time miss strategy adds a delay comparable to disk reads.

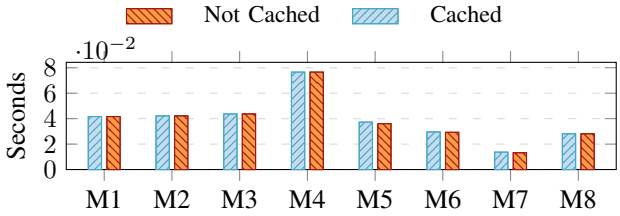


Fig. 9. NoCAP mitigates the flush-timing leakage by adding a delay when flushing a page not present in page cache. Comparing a cached and uncached file flushing, NoCAP mimics flush timing for 5 machines within 0.1 %, 1 within 0.98 %, and 2 within 3.4 % of the original flush timing.

cache; however, the primitive cannot target a page, *i.e.*, an attacker cannot discern whether a specific page is evicted.

Victim-Induced Eviction and Flushing. An attacker can still determine whether a victim evicted or deliberately flushed a page by measuring reload times. When a page is removed from cache, NoCAP applies the first-time miss strategy on the next reload, causing reload times to jump from fast (cached) to slow (disk). This is already observable under the default kernel and is not introduced by NoCAP's design.

This residual channel is bounded by the attacker's polling schedule: monitoring p public pages every Δ seconds yields at most one event bit per page (removed since the last poll or not), for an upper bound of p/Δ event bits/s. For non-cooperative attacks, the effective rate is further bounded by how often the victim actually flushes or evicts those pages. The signal does not reveal who removed the page, the cause (flush or evict), or whether the victim accessed it, but only that a removal occurred on an attacker-chosen public page. It is thus unclear what concrete attacks this primitive enables.

Security of Explicit Flushing Optimizations in Applications. In most cases, when a program optimizes flushing for a file it *writes* to, page cache attacks are not a concern for an attacker for several reasons. First, programs running under a user account typically write to files within the user's home directory, which is not accessible to a cross-user attacker in the threat model of page cache attacks. In general, if a program writes and flushes to a *private* file, inaccessible to the attacker, the attacker cannot observe reload times and, consequently, whether the file's pages were flushed. Second, if the program writes and flushes to a system-wide *public* file, readable by the attacker, the attacker can already monitor the file's contents and its changes, which is likely more sensitive than simply knowing which part of the file was flushed. The

more relevant concern arises when an attacker identifies a program that flushes a *globally-readable* page after reading it. In this case, the key security question is what information an attacker can infer from observing the page has been flushed.

Real-World Prevalence of Explicit Flushing. Our goal is to gain an approximate understanding of how and why varied codebases flush files. Prior work [27] identified a set of syscalls which can reload a page. According to Debian code search, which indexes open-source projects in Debian, there are more than 4 000 000 references to this set. On the other hand, there are only 1 235 references to `POSIX_FADV_DONTNEED` across 174 packages, and many simply define aliases of the flag rather than directly using it.

Since 174 packages are too large to manually analyze, we use Claude (Sonnet 4.6) to summarize flushing usage. We filter out macro/alias uses of `POSIX_FADV_DONTNEED`, leaving 89 packages. We then assess cross-user attack potential by determining: 1) if packages flush globally-readable files, 2) if they would flush such files, 3) if they flush self-created files, 4) if files are opened for writing, and 5) what attackers can infer from flushing in the package's deployment context. We disregard 29 cases where files are write-only, and exclude 14 testing/benchmarking tools (android-platform-system-extras, ioping, stress-ng), resulting in 46 final packages. We **manually** verified all outputs provided by Claude, to the best of our ability. We present our prompts, our validation methodology, and the final 46 packages in Appendix C.

Remaining Attack Surface. To the best of our knowledge, no attacks exploit flush events regardless of granularity (page or cache line). While some packages may be interesting to investigate (the "Human Expert Recommendation" column in Appendix C), we find no valuable leakage. The potential concerning cases we identify are possibly due to misconfiguration or are not practically significant. For instance, media software (kodi, vdr) might flush public files, revealing what a user streams. However, the attacker could already read those files. Similarly, tools like dd or cp copying public files to private directories leak information only in specific use cases. Until future work demonstrates critical leakage, the performance overhead of flush mitigation, e.g., private page cache copies, does not justify the security gained. Simply knowing a file was flushed rarely leaks sensitive information, and we leave further attack exploration to future work.

C. Evict+Evict

Attack Technique. Evict+Evict is a technique where the attacker applies eviction pressure on the page cache and measures how long it takes for one of its own pages to be evicted. In a set-associative cache this is Prime+Probe— prime a set, then probe whether the victim evicted the attacker's page. The page cache is fully associative, so there is no set to target, and thus, eviction reduces to an occupancy attack on the whole cache. This is the same residual shape as in fully-associative secure-cache designs like MIRAGE [35]: conflict-based targeting is removed, but aggregate occupancy can still, in principle, leak how much cache state another domain added.

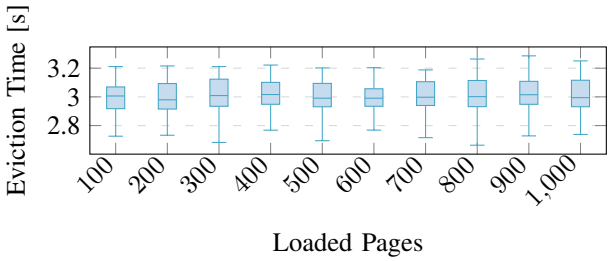


Fig. 10. Residual Evict+Evict leakage. Boxplots show eviction time after the victim loads 100 to 1000 pages. Medians differ by at most 36.8 ms, average IQR is 165 ms, and the best-case 10-class classifier reaches 18.9 % accuracy.

Monitor, reload, and flush can target specific pages via syscall arguments, but eviction cannot. Once NoCAP mitigates the first three, an attacker relying only on eviction can no longer ask whether a particular victim page is resident, but only measure whether global occupancy changes. We assume the attacker lacks direct memory-usage interfaces (e.g., `free`, `top`). This is conservative, since such interfaces would give the same information more directly. Instead, the attacker applies memory pressure until one of its own pages is evicted, using the time to eviction as an indirect occupancy measurement. If successful, this could reveal the approximate size of a globally-readable, victim-loaded file, but not access patterns.

Experimental Setup. We use the state-of-the-art eviction technique from Neela et al. [27], run on an AMD Ryzen 7 PRO 5850U (Ubuntu 22.04, Linux 6.8.0, 32 GB DDR4) under the residual-attack model, where monitor, reload, and flush are unavailable as victim-page oracles. For a best-case attacker, we remove background load, disable swapping, and restrict the victim to one of 10 working-set sizes (100–1 000 pages, step 100). Before each trial, the attacker establishes 24 GB of baseline pressure, then applies up to 8 GB more as the probe phase. The attacker first loads one page, the victim loads a randomly selected working set, and the attacker repeatedly adds pressure while reloading its own page. The stop condition is the first reload exceeding the cached-access threshold, indicating eviction, and the measured feature is elapsed time to that point. We repeat each working-set size 100 times in randomized order and evaluate a best-case binned single-sample classifier over the 10 classes.

Evaluation. We filter out outliers that lie outside 1.5 times the interquartile range, and present the boxplots in Figure 10. Across the 10 victim working-set sizes, median eviction time ranges from 2.980 s to 3.016 s, a spread of only 36.8 ms. By comparison, the average interquartile range is 165 ms, and the average whisker span after 1.5-IQR filtering is 504 ms. Using the binned distributions in Figure 10, an optimal single-sample classifier over the 10 classes reaches 18.9 % accuracy, compared to a 10 % random-guessing baseline. The mutual information between the victim working-set size and the measured eviction-time bin is 0.23 bits per sample; with a median sample time of approximately 3.0 s, this corresponds to roughly 0.08 bits/s in this idealized setup. Thus, Evict+

Evict exposes a weak occupancy signal, but not a reliable page-residency oracle comparable to monitor, reload, or flush. Although intuition suggests that a larger number of pages might require less time to be evicted, we find that this number largely fluctuates and is not consistent across runs. Furthermore, although our case study is constructed with just ten working-set sizes, real systems hold files with a variable number of pages, many of which are close in size. In those cases, identifying the exact file that is accessed is far noisier.

VII. Discussion & Related Work

In this section, we discuss what NoCAP does not cover, prior mitigations, and what matters for a practical deployment.

A. Monitor Mechanisms

The clearest and most direct leakage comes from interfaces that directly reveal page cache residency: monitor mechanisms. NoCAP intentionally does not mask such explicit oracles with noise as the defense's fourth principle assumes that monitor mechanisms are fixed when reported. If an interface reports residency, the robust fix is at that interface, e.g., by restricting the information returned, adding permission or capability checks, or changing cross-domain behavior. Kernel maintainers indicated that this is viable for direct monitor interfaces because they are explicit information flows that can be fixed when discovered without breaking userspace semantics [24]. Reload and flush are different: they leak through timing along common kernel paths while preserving ordinary syscall results. Those paths are the focus of NoCAP.

B. Remaining Attack Surface

NoCAP removes neither memory pressure nor `posix_fadvise` with `POSIX_FADV_DONTNEED`, *i.e.*, flushing. Both can still change cache state by removing pages, and an attacker can still use them to disturb or slow down system performance. This is outside the scope of a side-channel defense: the leak is not that pages can be removed (see Section VI-B), but that the attacker can reliably observe if a victim brought them back.

Consequently, denial-of-service and cache-disruption attacks remain possible. What NoCAP removes is the precise signal needed to turn such disruption into a practical page-residency channel. Eviction-only attacks are therefore the main residual side-channel concern, but they are slower, noisier, and less targeted than attacks based on the three mitigated primitives: monitor, reload, and flush [15], [21], [27]. Moreover, the residual eviction signal is only an aggregate occupancy signal; if an attacker can read ordinary memory-usage interfaces such as `free` or `top`, those interfaces provide the same information more directly, reliably, and most importantly, stealthily.

As NoCAP performs the mimic-I/O strategy to simulate miss penalties (see Section IV-C), it is possible that I/O device contention side channels [20] may be amplified. As all our benchmarks including FIO with concurrent jobs (see Figures 6 and 7) were performed on daily-used systems with background processes and daemons, all user processes were interacting with NoCAP. Since there is no visible performance drop from concurrent jobs, *i.e.*, no visible contention from up to 4 jobs, significantly more jobs may be required to induce visible and measurable contention. Hence, we see no indication that our defense enables a storage contention side-channel attack on a single victim job. Furthermore, the SSD contention side-channel attack [20] does not have a spatial dimension unlike page cache attacks, which can target specific $4\,\mathrm{kB}$ pages.

C. Relation to Prior Mitigations

Prior page-cache mitigations mostly patched individual monitor interfaces [24], [40]. NoCAP incorporates this lesson into its mitigation of the three relevant primitives: direct monitor oracles are not exposed across security domains, reload timing is hidden by first-time miss, and flush timing is hidden by adaptive delay. The restrictions for `mincore` and later `cachestat` satisfy this rule for two concrete interfaces, but timing-based attacks and future equivalent syscalls remain recurring problems without the rest of the NoCAP design [24], [27]. Making `posix_fadvise` with `POSIX_FADV_DONTNEED` privileged would mitigate all flush-based attacks, but it changes a long-standing interface that real applications use to manage the page cache. NoCAP therefore mitigates monitor, reload, and flush by combining the monitor rule with timing defenses for reload and flush, avoiding the compatibility cost of disabling ordinary page-cache operations.

NoCAP also differs from CPU-cache defenses in where the trade-off is paid. Making flush privileged, preventing flush execution, or making flush closer to constant time has been discussed for such attacks [12], [16], [31], [41]. First-time miss instead splits shared-cache timing by domain [33]. We adapt these two ideas to the software-managed page cache: first-time miss for reload and adaptive delay for flush. Unlike full partitioning or randomization, this does not split the entire page cache and therefore preserves most of the memory-saving benefit of shared file-backed pages.

D. Performance and Deployment

In Section IV-D, we disabled mmap readaround and asynchronous readahead since our prototype does not cover those paths. To gauge realistic performance, we ran FIO on $M1$, $M2$, and $M7$. With readaround and readahead enabled, sequential reads are $6.7\times$ faster on $M1$ and $9.4\times$ on $M7$, and random reads $6.1\times$ and $4.7\times$ faster, respectively. On $M2$ (HDD), the gains are much smaller, at $1.3\times$ for random and $1.2\times$ for sequential reads. If our prototype covered these paths, *i.e.*, a 'secure' asynchronous readahead that is itself first-time-miss aware, we believe the relative overhead may be impacted differently depending on the reading style. The impact would be at the *start* of file reads, as the first reads are not covered by asynchronous readahead in either case (the first read would either come from disk or page cache in the baseline). Subsequent *consecutive* reads from the process would benefit from the speed-up, as the secure asynchronous readahead brings the pages into the page cache ahead of the access while incurring the first-time miss delay, prior to the process accessing those pages. Therefore, the overhead would

be based on how fast the process reads the file and how much NoCAP impacts the secure asynchronous readahead. Random reads would not see such a speed-up, but they do not benefit from asynchronous readahead in the baseline either. The security implications and engineering challenges of such a design should be carefully considered.

Our adaptive-flush sysctl interface is unprivileged for testing, as practical deployment would use the privileged sysfs interface (`/sys/kernel/debug`). Parameters may require recalibration for new hardware, but we assume it is infrequent.

The main deployment question for NoCAP is the security-performance trade-off in the timing defenses. Our reload mitigation adds work on first cross-domain cache hits, while our flush mitigation delays selected flush operations to reduce timing leakage. Both mechanisms are designed to preserve syscall results and page-cache semantics, but their overhead depends on workload behavior. Workloads that repeatedly share and reload file-backed pages across domains, or workloads that heavily use `posix_fadvise` with `POSIX_FADV_DONTNEED`, are therefore the most relevant performance use cases to consider for a trade-off. Still, as shown in Section V, the overheads can be minimal for single domain I/O, e.g., single, simple processes performing repeated reads on the same file, whereas they are heavyweight when multiple shared libraries or binaries are shared across domains, e.g., Docker container launches experience a $4.29\times$ slowdown (see Section V-D). One aspect a real-world implementation of NoCAP would have to consider still is the readahead behavior of the kernel, which can introduce overheads when fetching a larger amount of unrelated data into the page cache. Still, our work shows that NoCAP is suitable as a kernel-level hardening mechanism, offering system administrators the choice to mitigate page-cache attacks without changing userspace semantics.

VIII. Conclusion

Page-cache side channels remained effectively unmitigated because the straightforward mitigations are prohibitive: they break backwards compatibility, and making the relevant operations constant-time introduces high overheads. We close this gap by showing that page-cache attacks can be mitigated at their root-cause leakage sources while preserving userspace-visible behavior. We presented NoCAP, a practical Linux defense that makes direct monitor primitives unavailable across security domains, hides reload timing with per-process first-time miss, and hides flush timing with an adaptive delay that preserves `posix_fadvise` with `POSIX_FADV_DONTNEED`. We implemented the reload and flush timing components of NoCAP and evaluated them on 8 systems, measuring only $0.48\,\%$ overhead in SPEC CPU 2017 Integer Rate, an $0.21\,\%$ average overhead with Redis benchmark, and a $4.29\times$ slowdown when launching Docker containers. With the Flexible I/O tester (FIO) benchmark, NoCAP incurred $2.05\,\%$ average overhead with synchronous reads, and $1.65\,\%$ average overhead with AIO and io_uring reads. With monitor, reload, and flush mitigated, the remaining page-cache side-channel surface is reduced to aggregate occupancy leakage: in our Evict+Evict experiment, a best-case binned classifier reaches only $18.9\,\%$ single-sample accuracy across 10 victim working-set sizes, corresponding to 0.23 bits per sample or approximately 0.08 bits/s, and this signal is weaker than ordinary memory-usage interfaces when they are available. NoCAP therefore demonstrates a deployable path to mitigate page-cache attacks without breaking the page-cache semantics that make those systems practical.

Ethics Considerations

This work is defensive and discloses no new vulnerability in a live third-party system. We ran every experiment on laboratory machines we own and control. The only external interaction is the Debian Code Search analysis in Section VI-B, which we queried through its public API with our API key.

Acknowledgment & Generative AI Usage

We thank our anonymous reviewers, and particularly thank our shepherd for their invaluable feedback and guidance. We further thank Smriti Suresh, Lukas Giner, Roland Czerny, and Simone Franza for their inputs at various stages of this work. This research is supported in part by the European Research Council (ERC project FSSec 101076409) and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding was provided by generous gifts from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

We used ChatGPT, Claude, Copilot, and Grammarly. No text was generated by these tools, though some sentences were retouched for clarity. Figure 1 was drafted with ChatGPT. No other figure content was tool-generated, but Claude wrote the pgfplots code that reads and plots our raw data, and the Python programs that post-process and print the statistics. Claude also summarized Debian Code Search (DCS) results for `POSIX_FADV_DONTNEED` usage and produced the initial package categorization in Section VI-B, which we verified against the source-code context returned by DCS before using the counts and security interpretation. We detail our prompt in Appendix C. All outputs were verified by the authors.

References

[1] D. Alden, "More timing side-channels for the page cache," 2026. [Online]. Available: https://lwn.net/Articles/1061743/

[2] J. Axboe, "fio - Flexible I/O tester," 2026. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html

[3] 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.

[4] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, 2016.

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

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

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

[8] ——, "Reconsidering BPF ABI stability," 2023. [Online]. Available: https://lwn.net/Articles/921088/

[9] ——, "On pages and folios," 2026. [Online]. Available: https://lwn.net/Articles/1064861/
[10] S. P. E. Corporation, "SPEC CPU 2017," 2017. [Online]. Available: https://www.spec.org/cpu2017/
[11] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, 2012.
[12] S. Enomoto and H. Kuzuno, "FlushBlocker: Lightweight mitigating mechanism for CPU cache flush instruction based attacks," in *EuroS&P Workshops*, 2021.
[13] 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 *S&P*, 2023.
[14] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, 2018.
[15] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, "Page Cache Attacks," in *CCS*, 2019.
[16] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.
[17] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security*, 2015.
[18] Intel, "Security Best Practices for Side Channel Resistance," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/security-best-practices-side-channel-resistance.html
[19] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement." in *USENIX ATC*, 2005.
[20] J. Juffinger, F. Rauscher, G. La Manna, and D. Gruss, "Secret Spilling Drive: Leaking User Behavior through SSD Contention," in *NDSS*, 2025.
[21] J. Juffinger, H. Weissteiner, T. Steinbauer, and D. Gruss, "The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer," in *DIMVA*, 2025.
[22] M. Kerrisk, *capabilities(7) — Linux manual page*, 2026, https://man7.org/linux/man-pages/man7/capabilities.7.html.
[23] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology (CRYPTO)*, 1996.
[24] J. Kosina, "make mincore() more conservative," 2019. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=134fca9063ad4851de767d1768180e5dede9a881
[25] Linux, "Getting the code right," 2026. [Online]. Available: https://docs.kernel.org/process/4.Coding.html
[26] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing," in *HPCA*, 2016.
[27] S. R. Neela, J. Juffinger, L. Maar, and D. Gruss, "Eviction Notice: Reviving and Advancing Page Cache Attacks," in *NDSS*, 2026.
[28] D. Ojha and S. Dwarkadas, "TimeCache: Using Time to Eliminate Cache Side Channels when Sharing Software," in *ISCA*, 2021.
[29] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *RSA Conference (CT-RSA)*, 2006.
[30] D. Page, "Partitioned Cache Architecture as a Side-Channel Defence Mechanism," *Cryptology ePrint Archive, Report 2005/280*, 2005.
[31] M. Peters, F. Stolz, J. P. Thoma, T. Güneysu, and Y. Yarom, "Revisiting Prime+Prune+Probe: Pitfalls and Remedies," in *ACSAC*, 2025.
[32] Phoronix Media, "Phoronix Test Suite," 2026. [Online]. Available: https://www.phoronix-test-suite.com/
[33] K. Ramkrishnan, S. McCamant, P. C. Yew, and A. Zhai, "First Time Miss: Low Overhead Mitigation for Shared Memory Cache Side Channels," in *Conference on Parallel Processing*, 2020.
[34] Redis, "Reclaim page cache of RDB file #11248," 2022. [Online]. Available: https://github.com/redis/redis/pull/11248
[35] G. Saileshwar and M. K. Qureshi, "MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design," in *USENIX Security*, 2021.
[36] M. Schwarzl, E. Kraft, and D. Gruss, "Layered Binary Templating," in *ACNS*, 2023.
[37] M. Schwarzl, E. Kraft, M. Lipp, and D. Gruss, "Remote Memory-Deduplication Attacks," in *NDSS*, 2022.
[38] 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/
[39] SPEC, "500.perlbench_r SPEC CPU 2017 Benchmark Description," 2020. [Online]. Available: https://www.spec.org/cpu2017/Docs/benchmarks/500.perlbench_r.html
[40] L. Torvalds, "cachestat: fix page cache statistics permission checking," 2025. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5f537664e705b0bf8b7e329861f20128534f6a83
[41] T. Vasilas and R. Brad, "A Decade in Software-Based Side and Covert Channel Attacks and Countermeasures: A Survey," *IEEE Access*, vol. 13, pp. 56 587–56 606, 2025.
[42] T. Waldmann, "use posix_fadvise to avoid spoiling the OS cache," 2015. [Online]. Available: https://github.com/borgbackup/borg/commit/c7d232c4cec6cf14424e8761135e627518b75ac5
[43] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 494, 2007.
[44] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security*, 2019.
[45] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security*, 2014.

Appendix A
Flexible I/O Tester (FIO) on $M7$

Figure 11 presents results of the Flexible I/O Tester on $M7$: the left half uses the synchronous read engine with 1, 2, and 4 jobs, the right half uses AIO and io_uring with 1 and 2 jobs.

Appendix B
Redis Performance: 500 & 1 000 Connections

In Figure 12, we present Redis' throughput with 500 and 1 000 parallel connections before and after NoCAP.

Appendix C
Flush Analysis With Claude

To analyze how packages flush files, we query Debian Code Search for occurrences of `POSIX_FADV_DONTNEED` and ask Anthropic's Claude (Sonnet 4.6) to process the response:

> Please access the fadv.json from debian code search for genuine use cases of `POSIX_FADV_DONTNEED`. We define genuine to be (a) not macros, aliases, or redefinitions; (b) has logical code right before and after; Place all your findings in a local findings.md file which contain the package name, a short description of the package, and whether or not the package has a genuine use case of `POSIX_FADV_DONTNEED`. Does the package flush files the program itself created? Does it flush files that are opened for writing? Furthermore, for the findings.md, please add per genuine-usage-package: what can a cross-user side-channel attacker infer based on knowing the package flushes a globally-readable file, keeping the package's typical deployment context in mind (how it is typically deployed, whether it runs with user-level permissions). We would also like to know whether the package would flush globally-readable files in another column. If the package uses `POSIX_FADV_DONTNEED` for benchmarks or internal tests, please put them in a separate category called tests and do not count these towards the genuine uses. Then, please sum up the total number of packages found and number of packages remaining with genuine use cases of `POSIX_FADV_DONTNEED`.

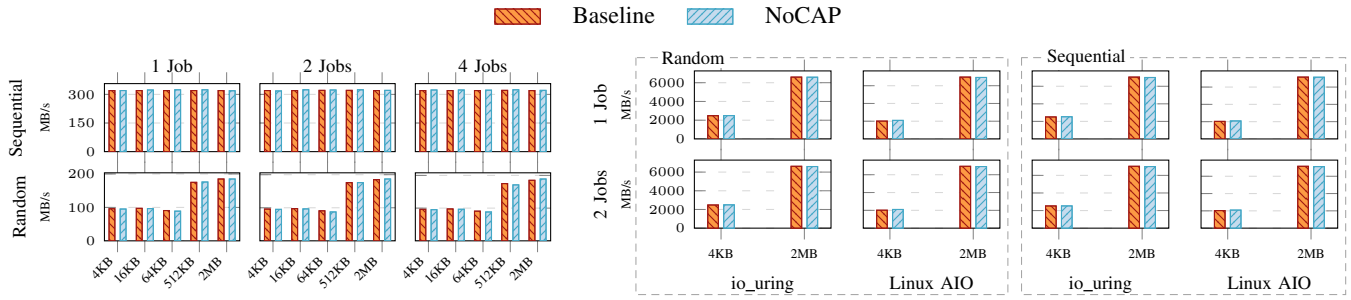


Fig. 11. FIO throughput on $M7$, AMD (Zen 5) Ryzen 9 9950X, for every read engine we test. **Left:** the synchronous engine at 1, 2, and 4 jobs; rows separate read patterns and columns separate job counts. **Right:** the io_uring and Linux AIO engines at 1 and 2 jobs; rows separate job counts and columns separate read pattern and engine. In both halves the x-axis is the FIO block size and the bars compare baseline and NoCAP throughput in MB/s.

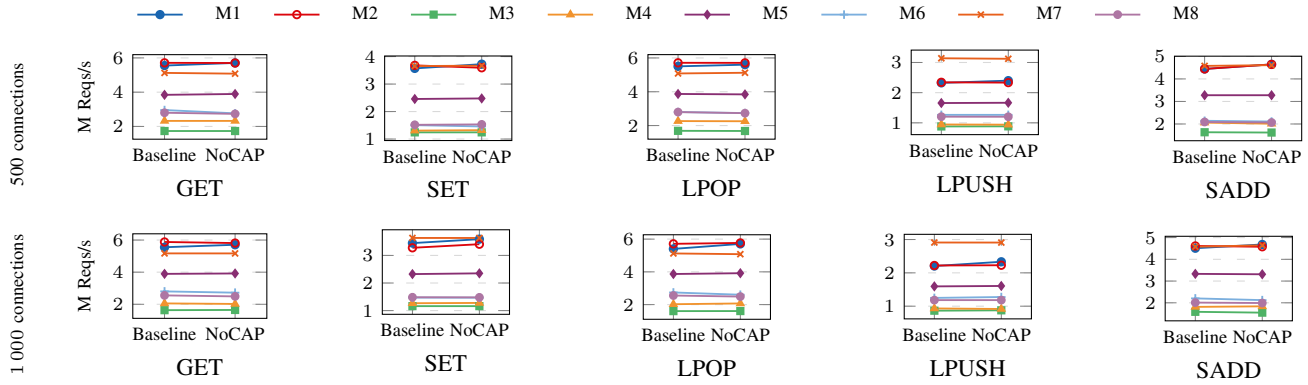


Fig. 12. Redis throughput per operation, before (Baseline) and after NoCAP, for every machine. The **top** row is 500 parallel connections and the **bottom** row is 1 000, as marked on the left of each row; columns separate the five Redis operations. Higher is more throughput.

We summarize Claude's findings in Table II and validate its interpretation for each of these 46 packages, to the best of our ability. Each package is complex, and understanding code context in files of tens of thousands of lines was itself aided by LLM tools. For validation, we searched through the codebases of the functions where flushing was used, attempting to determine where it ultimately gets called from. With some packages, e.g., coreutils' dd or cp, snapraid, vmtouch, or nocache, the input file is chosen by the user. With other packages, e.g., bacula, borgbackup, or bup, the files being backed up depend on whether the user chooses a public file. Packages with the "Maybe" *can* flush globally readable files, but it depends on a configuration in which that program can flush global files. For example, gnome-multi-writer flushes an ISO file after flashing it; the information leakage is relevant if the ISO is only accessible in a publicly-readable directory.

Appendix D
Artifact Appendix

Our paper presents a defense against page cache attacks in the Linux kernel. While page cache attacks were thought to be mostly mitigated in 2019, recent work has demonstrated that such attacks are still possible. With the classification of primitives which interact with the page cache — flush (measure time to remove page from cache), reload (measure time when bringing page into cache), evict (memory pressure to remove pages from cache), and monitor (kernel mechanisms reporting page presence in cache) —, and combinations of primitives into attack strategies, page cache attacks have been shown to be more severe than previously thought.

Our kernel-level defense mitigates page cache attacks with four principles. Important for this artifact are Principles 2 and 3: preserve page cache semantics and remove observability, not functionality, *i.e.*, do not break userspace while mitigating page cache attacks. The flush primitive has been mitigated using an adaptive delay, where the time taken to flush pages no longer leaks whether or not a page is in cache. The reload primitive has been mitigated using a first time miss principle, where every process encountering a page for the first time experiences as if the page comes from disk.

A. Description & Requirements

1) How to access: Our source code and virtual machine is available at Zenodo: https://zenodo.org/records/21864019, with only the source code made conveniently available at https://github.com/isec-tugraz/NoCAP.

2) Hardware dependencies: We tested our virtual machine on x86 desktop and notebook CPUs with at least 16 GB of memory. We recommend that the evaluation system have at least 4 GB of memory free while running the virtual machine. We strongly recommend using a machine with an SSD; machines with HDDs still work, but will increase review time.

TABLE II
GENUINE PRODUCTION USES OF `POSIX_FADV_DONTNEED` IN READ MODE

Package	Can Flush Globally Readable Files	Cross-user inference	Human Expert Recommendation
bacula	Maybe	Which files are being backed up	✓
borgbackup	Maybe	File access order during backup	✓
btrfs-progs	Maybe	Which files are being defragmented	
bup	Maybe	Which files/sizes are being chunked during backup	✓
cloop	Maybe	When decompression completed	
coreutils	Yes	Copy timing (cp, dd);	✓
dpkg	Yes	Which package was installed/upgraded and when (`/usr/` files are world-readable)	
e2fsprogs	Maybe	Which files are being defragmented	
erlang	Maybe	Redefinition	
fio	Maybe	Which file and at which offsets fio is benchmarking	
freefilesync	Maybe	Which files were synced	✓
gfs2-utils	Maybe	When GFS2 mkfs operation completed	
gnome-multi-writer	Maybe	Which ISO is being burned	✓
guestfs-tools	Maybe	When guestfs operation completed on which VM image (Unclear whether called)	
hashrat	Maybe	Flushes hashrat's log files	
insserv	Yes	Which init scripts were processed	
kodi	Maybe	Which media file is playing; flush offset reveals playback position	✓
lftp	Maybe	Which torrent files were recently seeded or completed	
libosmium	Maybe	Which OSM data compressed files were flushed	
localsearch	Maybe	After extracting metadata of a file, localsearch flushes the file	
mono	Maybe	Redefinition: False positive	×
nbdkit	Maybe	Plugin offers a method to flush	
ncdc	Maybe	Which files are downloaded or served via DC++	
nocache	Yes	Which file is flushed (explicit user request)	✓
nodejs	Maybe	Which WebAssembly WASI files are flushed	
ocamlnet	Maybe	Redefinition: False Positive	×
ocfs2-tools	Maybe	OCFS2 device file was detected	
plocate	Maybe	Flushes whole database to test cold-cache behavior	
proftpd-dfsg	Maybe	Redefinition: False Positive	×
qemu	Maybe	Flush VM during live-migration	✓
recoll	Maybe	fstree walk processed file	
rust-coreutils	Yes	Copied file (same as GNU dd)	
samhain	Yes	File was scanned	
scrub	Maybe	Whether it flushed its *own* file	
snapraid	Maybe	User flushed out file via commandline	✓
startpar	Yes	Which services start at boot	
ucommon	Maybe	Redefinition: False Positive	×
util-linux	Maybe	Generic utilities, flush used internally	
vdr	Maybe	Which recording is playing	✓
vdr-plugin-streamdev	Maybe	Which recordings are being streamed	✓
vdr-plugin-vnsiserver	Maybe	Which recordings are accessed remotely	✓
vmtouch	Yes	Which files are explicitly evicted	✓
w-scan-cpp	Maybe	Contains vdr code	
xapian-omega	Maybe	Which files are crawled/indexed	
xfsprogs	Yes	Which file is marked unneeded	✓
yadifa	Maybe	Which DNS zones were loaded and zone reload timing	

Symbols: (✓) recommended by a human expert for future investigation; (×) concluded to be a false positive.

3) Software dependencies: The host machine (evaluation computer) must be running Linux, preferably Debian-based like Ubuntu (above 22.04). The host machine should have QEMU emulator version 7.0.0 or later installed (verifiable with `qemu-system-x86_64 --version`), have KVM acceleration, and have the evaluating user in the `kvm` group.

To easily view and inspect the patches in an interactive GUI manner, we recommend KDE's Kompare (may require additional KDE libraries). Otherwise, command-line tools or even VS Code are sufficient to inspect the kernel patches.

4) Benchmarks: None

B. Artifact Installation & Configuration

Download the QEMU `nocap-vm.qcow2` disk image and the `run-vm.sh` shell script. Verify that the VM boots with the baseline kernel (stock Linux 6.18.2) with `./run-vm.sh baseline`. Input to the VM can be sent through either the QEMU graphical window or the terminal console.

There are two users in this VM:

1) **root** whose password is **root**, and
2) **user** whose password is **user**.

The system-level configuration of the VM is performed as **root**, while NoCAP is exercised as **user**.

C. Experiment Workflow

Detailed instructions are provided in the artifact's README. This section provides the high-level workflow.

First, check whether the VM launches:

1) Download the QEMU disk image.
2) Download the run-vm.sh script.
3) Test whether the VM launches with the baseline kernel.
4) Verify the kernel using `uname -r`.

Second, inspect the kernel patches.

Third, launch baseline and calibrate the flush threshold:

1) Launch the VM with baseline and log in as **root**
2) Navigate to the calibration directory
3) Run the flush calibration (10-30 min).
4) Disable the asynchronous read-ahead kernel mechanism
5) Log out as **root**

Fourth, verify that page cache attacks are possible:

1) In the VM with baseline, log in as **user**
2) Create a testing file using the script
3) In `primitives/`, compile the testing programs
4) Test flush: observe timing differences when cached and not
5) Test reload: observe timing differences cached and not
6) Log out as **user**, log in as **root**, and `poweroff` the VM.

Finally, verify that the defense mitigates the timing leakage introduced with flush and reload:

1) Launch the VM with NoCAP and log in as **root**
2) Apply the flush calibration parameters, disable the asynchronous read-ahead mechanism
3) Log out as **root** and log in as **user**
4) Test flush: no timing differences when cached and not
5) Test reload: no timing differences when cached and not

D. Major Claims

- (C1): NoCAP mitigates the timing difference in flush (`posix_fadvise + POSIX_FADV_DONTNEED`), used by an attacker to determine whether or not a page is in cache. Flushing a non-encountered page should take the same time regardless of presence in page cache (Figure 2, Section IV-E). This is proven by experiment (E2).
- (C2): NoCAP mitigates the reload timing difference (`read`) with a mimicking-I/O strategy (Section IV-C). Reading a non-encountered page should take the same time regardless of whether it is cached. This is proven by experiment (E3).
- (C3): NoCAP does not change userspace semantics. Flushing still removes a page; only the *timing* is affected (Section III-D). This is proven by experiment (E2).

E. Evaluation

The artifact's README provides detailed instructions and commands to demonstrate the functioning of our defense.

1) Experiment (E1): [Flush Timing Calibration] [10 human- + 30 computer-minutes]: Calibrating the flush time

[Preparation] Launch the VM with the baseline kernel (`./run-vm.sh baseline`) and log in as **root**

[Execution] Run the `flush-calibration.sh` script in `/root/flush-calibration/parameter-determination/`, which may take between 10-30 min depending on hardware.

[Results] The calibration parameters are saved in `data/`.

2) Experiment (E2): [Flush Timing] [10 human-minutes]: Determining the flush timing is mitigated

[Preparation] Launch the VM with the baseline kernel (`./run-vm.sh baseline`), afterwards with the NoCAP kernel (`./run-vm.sh nocap`)

[Execution] First in the baseline kernel as **root**, disable the asynchronous readahead (`./disable-readahead.sh`). Log out as **root**, and log in as **user**. In the `primitives/` directory, compile (`make`) and first bring the testing file into cache using `./build/reload`. Use `./build/monitor-cachestat` against the testing file to ensure the pages are cached. Run `./build/flush` against the testing file and note the timings. Use `./build/monitor-cachestat` against the testing file to ensure the pages are no longer cached. Run `./build/flush` against the testing file and note the timing difference.

Perform the same set of steps but in the NoCAP kernel. First as **root**, run `./apply-sysctl-params.sh` and `./disable-readahead.sh`. Then as **user**, repeat the same set of steps to test flush timing: reload the file, check presence using monitor, flush the file (note the timings), check presence using monitor, flush the file again (note the timings). For stable values, repeat the set of steps multiple times.

[Results] In the baseline kernel, flushing pages in cache should take **longer** than flushing pages not in the cache. In the NoCAP kernel, they should both be around the same time. Note that it may be possible that in the NoCAP kernel, the time taken to flush pages may be longer than the baseline kernel. This behavior is expected, as the adaptive flush line is a linear regression with some margin added above it.

3) Experiment (E3): [Reload Timing] [10 human-minutes]: Determining the reload timing is mitigated

[Preparation] Launch the VM first with the baseline kernel, following which launch it with the NoCAP kernel

[Execution] First in the baseline kernel as **root**, disable the asynchronous readahead (`./disable-readahead.sh`). Log out as **root**, and log in as **user**. In the `primitives/` directory, compile (`make`) and first flush the testing file out of cache using `./build/flush`. Use `./build/monitor-cachestat` against the testing file to ensure the pages are not in cache. Run `./build/reload` against the testing file — note the timings. Use `./build/monitor-cachestat` against the testing file to ensure the pages are resident in cache. Run `./build/reload` against the testing file — note the timing difference.

Perform the same set of steps but in the NoCAP kernel. First as **root**, run `./apply-sysctl-params.sh` and `./disable-readahead.sh`. Then as **user**, repeat the same set of steps to test reload timing: flush the file, check presence using monitor, reload the file (note the timings), check presence using monitor, reload the file again (note the timings). For stable values, repeat the set of steps multiple times.

[Results] In the baseline kernel, reloading pages in cache should take **less time** than reloading pages not in the cache. With NoCAP, they should both be around the same time.

F. Notes

Our experiments and benchmarks in the paper were run natively on the host kernel, with the mitigated kernel active directly on hardware, not inside a VM. The experimental machines (Table I) were daily-used systems with multiple physical cores, running normal background processes and systemd services. All results in the paper were collected on Ubuntu systems, with one exception on Fedora.

The VM supplied with this artifact only demonstrates that our prototype functions correctly; it was not used to produce the paper's results. We chose Void Linux over a full Ubuntu `qcow2` image for its minimal footprint, and so reviewers may see differences from the reported numbers due to the fewer vCPUs, the virtualized environment, and runit instead of systemd.