

Zepade Attacks: Exploiting Zero-Page Deduplication for Side-Channel Attacks

Simon Einzinger, Tristan Hornetz, Michael Schwarz
CISPA Helmholtz Center for Information Security

***Abstract*—Page deduplication is a common optimization in modern operating systems and hypervisors that reduces memory consumption by merging identical pages into a single physical copy. Prior work has shown that deduplication can introduce side channels by exploiting timing differences in Copy-on-Write faults for attacker-crafted page replicas. Consequently, page deduplication is often disabled by default.**

In this paper, we find that even when page deduplication is disabled, zero-filled memory pages remain an always-on, largely overlooked source of shared state. We demonstrate that read-only accesses to zero-pages can be exploited via cache attacks on the shared physical backing page, enabling purely read-based side-channel attacks that preserve sharing. Based on these effects, we introduce Zepade attacks, which exploit zero-page deduplication. We demonstrate the impact of Zepade in three directions. First, we show that zero-page effects break the isolation assumptions of sparse, linear-scan ORAM-style key-value stores, leaking which entries are populated even though the implementation performs a fixed scan over all entries. Second, we show that zero-page access patterns enable application and workload fingerprinting, including inside Intel TDX confidential virtual machines, and demonstrate website fingerprinting with an accuracy above $90\,\%$. Third, we characterize the primitive through covert channels across processes, containers, browsers, and KSM-enabled virtual machines. Finally, we confirm that removing zero-page deduplication in the Linux kernel eliminates the side channel with no measurable performance overhead on SPEC CPU 2017 and minimal memory overhead. Our results indicate that zero-page deduplication is not a benign corner case of deduplication, but an always-on OS-created shared-memory primitive that must be considered in side-channel threat models.

1. Introduction

Operating systems and hypervisors are constantly trying to find a good trade-off between strong isolation guarantees and aggressive performance optimizations. Memory virtualization is a cornerstone of this design space: it provides each process with the illusion of private memory while relying on shared physical resources underneath. To reduce memory pressure and improve cache utilization, modern systems employ content-based memory deduplication, merging pages with identical contents into a single Copy-on-Write (CoW) mapping [12]. While this optimization is highly effective in practice, it introduces implicit shared state between otherwise isolated execution contexts.

A large body of prior work [49, 50, 58, 5, 41, 13, 7, 25, 22, 47] has demonstrated that such shared state can be abused to build side channels. In particular, memory deduplication has been exploited to infer the presence of data [13, 7, 47, 4], identify applications [13, 25, 33], break address space layout randomization [5, 22, 47], and establish covert channels between mutually untrusted processes or virtual machines [57, 41]. Prior work [7] also advocated for removing non-zero page deduplication and only using zero-page deduplication, which they consider secure. As a result, cloud and multi-tenant environments widely disabled or restricted cross-domain deduplication [35].

In this paper, we show that the assumption that zero-page deduplication is secure is false. Zero-pages arise naturally during process creation, memory allocation, and initialization, and operating systems typically optimize their handling by mapping a single read-only zero-page into many address spaces [51]. Even when memory deduplication is disabled, zero-page deduplication remains *always enabled* as a performance optimization. Zero-page deduplication is applied aggressively, often across processes and virtual machines, because zero-pages are ubiquitous and trivially identical. Consequently, accesses to these pages converge on the same physical memory and, thus, the same last-level cache lines.

We introduce Zepade attacks[1], which exploit this shared zero page as a read-only side channel. The attacker obtains its own mapping to a zero-backed page and monitors the corresponding cache lines with standard cache techniques such as Flush+Reload, Flush+Flush, or Evict+Reload. Whenever the victim reads from a zero-backed page at the same page offset, the access changes the shared cache state, making it observable to the attacker. Unlike earlier deduplication attacks that rely on CoW faults triggered by writes [13, 7, 5], our attacks are purely read-based and do not destroy sharing: the attacker only *reads*, and the victim only needs to *read* zero-filled memory. This allows continuous observation of zero-page access patterns.

In contrast to classical cache-based side-channel attacks, zero-pages reveal information about the accessed *location*, and additionally reveal that the accessed *content* is zero. This matters for constant-time software that prevents secret leakage by making its memory-access pattern independent of the secret. We show that there can be implementations that are side-channel-free under the classical definition [18]

1. Zero-page deduplication attacks

yet still leak information via the zero-page side-channel. This observation is one of the central security implications of our work. We demonstrate it with a sparse, linear-scan ORAM-style key-value store [61]. The implementation scans all entries for every lookup and update, and neither static nor dynamic analysis finds secret-dependent memory accesses. Nevertheless, empty entries remain zero-backed, while populated entries are backed by private physical pages. By observing zero-page cache activity during the scan, an attacker infers which entries are empty and which keys exist. This shows that OS-level content-dependent physical aliasing can invalidate the usual virtual-address notion of obliviousness. More broadly, the same issue applies to sparse arrays, hash tables with empty slots, bitmaps, and other data structures in which zero-filled regions encode semantic state.

We further show that zero-page access patterns are sufficiently stable for classification and triggering. During startup and workload initialization, applications access characteristic zero-backed pages. We use these patterns to fingerprint user applications, server applications in containers, websites, and workloads inside Intel TDX confidential virtual machines. An attacker can train a classifier that distinguishes applications and visited websites with high accuracy, resulting in a mean accuracy of up to $90\,\%$ for a 5-fold cross-validation. Crucially, this fingerprinting can be done by unprivileged users, even from restricted environments such as Docker containers or JavaScript runtimes in browsers. We demonstrate that zero pages are also ubiquitous in browsers. Arrays larger than one page are backed by the zero page, starting with the second page in Chrome, giving attackers trivial access to shared memory. We demonstrate that this allows for building covert cross-tab and cross-browser channels with Evict+Reload, which is simpler than Prime+Probe, as used in prior work [31, 48].

Additionally, we show that zero pages help attackers in implementing building blocks for other attacks. We demonstrate a UI-redressing attack that uses zero-page access patterns to detect program startup. We also show that the `umwait`-based architectural side-channel [62] can be used with zero-pages, yielding a highly reliable covert channel that requires no timing and transmits data at $678\,\mathrm{B/s}$. Finally, we show that zero-page deduplication simplifies the setup of ZombieLoad v3 [45] by eliminating the need for explicit shared memory. Thus, similar to the memory-deduplication-based setup [10], zero-page deduplication may enable such attacks across a wider range of scenarios.

Beyond demonstrating the attack surface, we also address defenses. Unlike memory deduplication, zero-page deduplication on Linux cannot be disabled at boot or runtime. Thus, we design, implement, and evaluate a mitigation for the Linux kernel that prevents zero-page-based leakage while preserving the performance benefits of existing memory management mechanisms. Our evaluation shows that the mitigation effectively eliminates the side channel, incurs no performance overhead on SPEC CPU 2017, and adds only a small memory overhead of $1.96\,\%$ across the system. Thus, it is suitable for deployment in both conventional and confidential virtualized environments.

Our findings complement and extend prior work on deduplication-based attacks, which primarily focused on attacker-controlled data and write-induced faults. In contrast, zero-pages are system-managed, always present, and require no interaction with victim-specific content. This makes them an attractive, scalable target, even when classical memory deduplication is disabled and even with hardware-enforced confidential computing.

Contributions. We make the following contributions:

- We identify Linux zero-page deduplication as an always-on, OS-created shared-memory primitive that remains exploitable even when ordinary page deduplication is disabled, leaking memory access patterns in combination with information about the content (zero vs. non-zero).
- We demonstrate Zepade on sparse, linear-scan ORAM-style key-value stores, where we reveal populated entries despite constant memory-access patterns.
- We demonstrate the versatility of Zepade attacks with several case studies, demonstrating fingerprinting of websites, applications, and workloads, even within Intel TDX, as well as UI redressing attacks.
- We design and evaluate a Linux kernel mitigation that closes the zero-page side channel with only a small performance overhead.

Availability. The artifact can be accessed through https://doi.org/10.5281/zenodo.22842539.

Outline. The remainder of the paper is structured as follows: Section 2 provides necessary background. Section 3 contains a systematic overview of zero-page usage across platforms and applications and describes how to monitor access patterns on the zero-page. Section 4 implements covert channels across different isolation boundaries. Section 5 presents several case studies demonstrating the power of Zepade attacks. Section 6 shows mitigations against Zepade attacks. Section 7 provides an overview of related work. Section 8 concludes.

2. Background

In this section, we provide background on memory deduplication, cache attacks, and briefly discuss confidential virtual machines as a modern isolation primitive.

2.1. Memory Deduplication

To reduce memory pressure, especially in virtualized and cloud environments, operating systems and hypervisors employ content-based memory deduplication [50]. Deduplication scans physical memory for pages with identical contents and merges them into a single physical page, which is mapped read-only across multiple address spaces. Write accesses to such a shared page trigger a Copy-on-Write (CoW) page fault, upon which the operating system creates a private copy for the writing process [51].

On Linux, Kernel Samepage Merging (KSM) implements memory deduplication primarily for KVM by periodically scanning candidate pages and merging identical ones [3]. These mechanisms also exist in other operating systems, such as Windows [47] or Android [23]. While cross-domain deduplication has largely been disabled due to security concerns, deduplication within a single security domain and for specific page classes remains in use [47].

Zero Pages. A special case of deduplication concerns zero-filled pages. During process creation and memory allocation, large fractions of memory are initialized to zero. Instead of allocating separate physical pages, operating systems typically map a single, global zero-page into many address spaces as a read-only mapping [51]. Upon a write access, the CoW mechanism transparently allocates a private page.

Zero-page sharing is considered a fundamental optimization and remains enabled even when general-purpose deduplication mechanisms are disabled. Mainline Linux provides no supported runtime knob analogous to KSM to disable the global shared user zero page. Moreover, Linux supports merging zero-pages across virtual machines via KSM, further increasing the scope of sharing [3]. As a result, zero-pages are often widely shared physical memory pages in a system.

2.2. CPU Caches

Modern CPUs employ multi-level cache hierarchies to bridge the latency gap between processor cores and main memory. Caches exploit temporal and spatial locality by storing recently accessed data closer to the cores. Typical systems have private L1 and L2 caches per core and a shared Last-Level Cache (LLC) across cores.

Caches are organized in fixed-size cache lines and arranged in a set-associative structure. On most x86 systems, the LLC is physically indexed and physically tagged, meaning that cache indexing depends on physical addresses. Consequently, different virtual addresses that map to the same physical address also map to the same cache lines. This property is central to many cache-based side-channel attacks, as it enables information leakage across isolation boundaries when physical memory is shared.

2.3. Side-Channel Attacks

Cache-Based Side Channels. Cache-based side-channel attacks exploit timing differences between cache hits and cache misses to infer victim behavior [6, 32, 40, 20, 60, 15, 52, 53]. Attacks such as Flush+Reload [60] and Flush+Flush [15] rely on shared memory between attacker and victim to observe cache activity at cache-line granularity. Other techniques, such as Prime+Probe [32], do not require shared memory but offer lower spatial resolution.

Memory deduplication implicitly introduces shared physical pages between otherwise isolated processes or virtual machines. As a result, it enables cache-based attacks that would otherwise be infeasible, effectively reintroducing shared memory across security boundaries.

Attacks on Memory Deduplication. Memory deduplication attacks exploit timing differences introduced by CoW page faults or related effects [50, 33, 13, 7, 57, 58, 41, 47]. In a typical attack, an attacker places a candidate page into memory and waits for deduplication. A subsequent write reveals whether the page was deduplicated, as CoW faults incur significantly higher latency. Such attacks have been used to infer the presence of data, fingerprint applications, and break address space layout randomization [33, 20, 5, 25, 22, 13, 47]. These attacks generally assume that the attacker can create exact replicas of victim pages and synchronize with the deduplication process.

2.4. Confidential Virtual Machines

Confidential Virtual Machines (CVMs) aim to protect guest memory from a potentially malicious hypervisor using hardware-enforced isolation. Technologies such as AMD SEV [1, 2] and Intel TDX [19] provide memory confidentiality and integrity by encrypting and isolating guest memory at the hardware level. Additionally, as with regular hardware-assisted virtual machines, CVMs rely on a two-stage address translation procedure. First, the CPU uses guest-controlled page tables to translate a virtual address into a guest-physical address (GPA). Second, hypervisor-controlled page tables map the GPA to a DRAM-backed host physical address (HPA). This lets the guest enforce its own memory segmentation while preventing the hypervisor from inspecting its memory layout. However, these mechanisms still rely on shared microarchitectural components, including caches. As a result, attack vectors such as cache-side channels [38, 61] and controlled-channel attacks [59, 55, 54, 28] remain relevant threats, so optimizations such as zero-page sharing must be carefully reconsidered for confidential computing.

3. Shared Zero-Pages as a Microarchitectural Attack Surface

This section describes how shared zero-pages can be exploited to mount side-channel attacks. We provide a high-level overview of the attack primitives in Section 3.1, and a systematic analysis of where and how zero-pages are used in practice in Section 3.2 and Section 3.3. Finally, Section 3.4 describes how we monitor the zero page.

3.1. Overview

Zero-pages introduce implicitly shared physical memory between processes that are otherwise isolated [51]. Furthermore, they play a central role during process initialization [51]. Large parts of a process address space are initially zero-filled, and accesses to these regions are satisfied by the shared zero-page until the first write occurs. As a result, different applications induce characteristic sequences of zero-page accesses during startup. Because modern processors implement physically indexed last-level caches, all accesses

to a shared zero-page converge on the same cache lines, independent of which process or virtual machine issued the access [44], as illustrated in Figure 1. Using a cache attack, such as Flush+Reload [60] or Evict+Reload [26], an attacker can use any virtual mapping to the zero page to infer access patterns on the zero page. These access patterns can be used in various ways, as we show in Section 5. Our two main directions are using side-channel information directly, e.g., to build covert channels, or training a classifier to identify patterns, such as visited websites or applications started.

3.2. Zero-Page Usage Across Platforms

To understand the generality of zero-page-based side channels, we systematically analyze where zero-pages are used across memory regions, execution environments, and system configurations. Table 1 shows an overview of the systems we use for our analysis. Our analysis is based on documented operating system behavior and empirical measurements from a dedicated test application that resolves the physical frame number backing each virtual page. Additionally, when this is not possible, we build a side-channel-based testing tool. This tool creates a Flush+Reload histogram on explicit shared memory, non-shared memory, and a potential deduplicated zero-page. Based on how it compares to the two known scenarios (shared memory vs. non-shared memory), the tool determines whether zero-page deduplication is enabled.

At the level of memory regions, zero-pages are predominantly used to back anonymous memory that is logically initialized to zero. On Linux, newly allocated stack pages are initially backed by the global zero-page. The same holds for uninitialized global and static variables, which language standards and ABIs require to be zero-initialized. Heap growth also heavily relies on zero-pages: memory obtained via common allocation mechanisms such as `malloc`, growing `realloc`, `aligned_alloc`, `valloc`, and `sbrk` is initially mapped to the shared zero-page and only materialized as private physical memory upon the first write. Anonymous memory mappings created with `mmap` and `MAP_PRIVATE`, including read-only mappings and mappings without eager population, similarly rely on zero-pages.

Zero-initialization does not universally imply zero-page backing. The global zero-page does not back thread-local storage pages. Memory returned by `calloc` is eagerly initialized and therefore bypasses the shared zero-page. Anonymous mappings created with `MAP_POPULATE` force immediate physical allocation and do not use zero-pages. Shared anonymous mappings and file-backed mappings, regardless of whether they are private or shared, are backed by distinct physical pages. These observations show that zero-pages primarily back lazily allocated, private, anonymous memory, which constitutes a substantial fraction of memory accessed during process startup and dynamic allocation.

TABLE 1: Systems used to analyze zero-page usage across architectures and operating systems.

ISA	CPU	OS
x86	Intel Core i7-1365U	Ubuntu 24.04
x86	AMD Ryzen 7 7800X3D	Ubuntu 24.04
x86	AMD Ryzen 9 9950X3D	Windows 11, Ubuntu 24.04
ARMv8	ARM Cortex A73	Ubuntu 20.04
ARMv8	ARM Cortex X3	Android 16
RISC-V	T-Head C910	Debian 12

Instruction Set Architectures. All major architectures that support virtual memory, including x86, ARM, and RISC-V, implement equivalent mechanisms. As shown in Table 1, we successfully verify zero-page deduplication on all of these ISAs. On all these architectures, the cache state of shared zero-pages is observable through cache side channels. As a result, the attack surface is not architecture-specific.

Operating Systems. Zero-pages are also a common optimization across operating systems. Linux explicitly maintains a global zero-page and optionally merges zero-pages across virtual machines via Kernel Samepage Merging (KSM). Zero-page deduplication is always enabled and cannot be disabled. We experimentally verify zero-page deduplication on Ubuntu 20.04, 24.04, and Debian 12. Android, derived from Linux, also supports zero-page deduplication. We experimentally verify this on a Pixel 8 running Android 16. Unlike Linux, Windows does not use zero-page deduplication with a global zero page. Using the Windows 11 kernel debugger, we verify this by inspecting the physical addresses of allocated but uninitialized memory. New allocations always yield fresh process-private physical pages, which are initialized to zero and not backed by a shared zero page.

Execution Environments. Finally, zero-pages remain relevant across different execution and isolation environments. We observe zero-page usage in native processes, containers, virtual machines, and browser processes. The same global zero page is used across Docker containers by default. We verify this on a default Docker installation on Ubuntu 24.04, as well as in the cloud on the "Play with Docker" playground[2] supported by Docker. To analyze zero-page deduplication in the browser, we add a helper function to Chrome 145 that returns the virtual address of an `ArrayBuffer`. Chrome uses the zero page starting from the second page of an allocated `ArrayBuffer`. Thus, mapping the zero page is as simple as allocating an 8 kB buffer. Importantly, zero-page sharing also persists inside confidential virtual machines. Although confidential computing technologies protect guest memory contents from a malicious host, they still rely on shared microarchitectural resources, such as caches. As a result, zero-page-based side channels remain exploitable across confidential virtual machines.

3.3. Zero-Page Usage in Applications

To better understand zero-page usage in real-world applications, we analyze their access patterns on Linux. For

2. https://labs.play-with-docker.com/

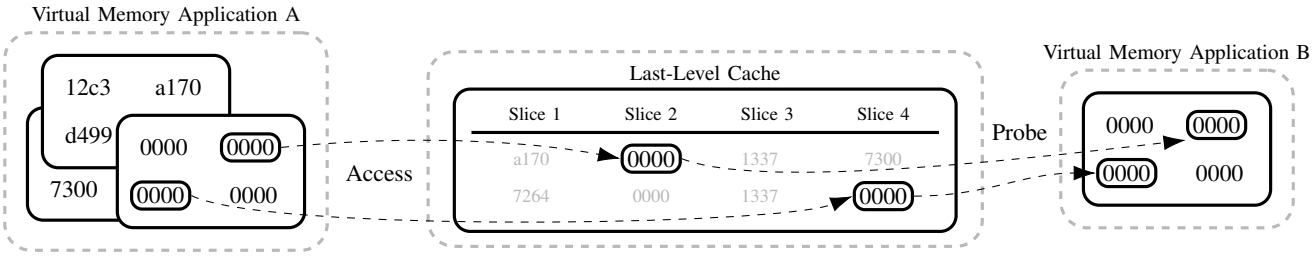

Figure 1: Overview of Zepade attacks exploiting shared zero-pages. Application A primes the last-level cache by accessing the shared zero-page, filling the corresponding cache lines. Application B probes the cache state by measuring access times to the shared zero-page. These timings can reveal whether application A accessed the zero-page and, if so, which offsets it accessed, enabling covert communication or fingerprinting based on temporal and spatial access patterns.

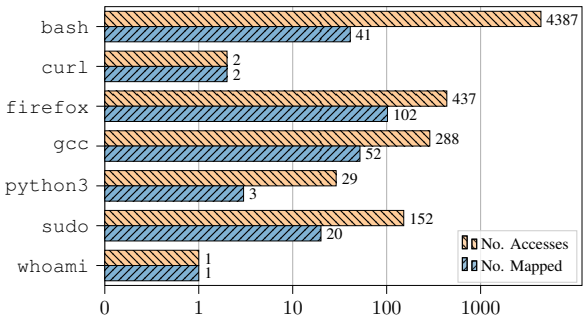

Figure 2: The number of zero-pages mapped and the number of accesses to those zero pages for various applications on Ubuntu 24.04.

this, we extend the Linux kernel with the capability to trace when a process maps or unmaps a zero-page, and when it accesses a zero page. We observe that shared zero pages are near-ubiquitously used across applications, including system utilities and web browsers.

Tracer Implementation. Our tracer consists of three components: a series of patches for Linux v6.8.0 on x86_64, a kernel module, and a shared library. The kernel patches let us register callbacks for events such as mapping and unmapping zero pages. Our kernel module then uses these callbacks to record events, allowing us to track how many zero pages a specific process uses. Optionally, the kernel module can also trace accesses to the zero page. If this is enabled, the module clears the user-accessible bit of the zero page's page table entry, triggering a page fault when the application accesses the page. We then make the page accessible again, allow the application to resume execution for one instruction, and then block access to the page again. This allows us to trace how often the application accesses the zero page, and at which addresses. The final component is a shared ELF library that users can inject into their application using `LD_PRELOAD`. This library registers the application with the kernel module and saves the traces to a file when the application exits.

Results. We run this tracer on various programs from the Ubuntu 24.04 package repositories, recording the total number of times each program maps the shared zero-page and how often these pages are accessed across the entire process tree created by the application. This includes complex applications such as the Firefox web browser, as well as simple utilities such as `whoami` from the GNU coreutils. Additionally, we test GCC and Python 3, which we feed with simple "Hello World" programs (cf. Appendix B for details).

The results of this experiment are shown in Figure 2 and confirm that all test applications use the shared zero page. However, the degree to which they use the zero page varies: For instance, the simple `whoami` utility uses the zero page only once, whereas `bash` accesses it more than 4000 times when starting up. We also find that the number of times an application maps the zero page does not necessarily correlate with the number of accesses. Firefox, for example, maps the zero page more than twice as often as `bash`, but uses it less.

Differences in Access Patterns. Because our tracer records the exact address of each access to the shared zero page, we can build a trace of the cache lines accessed during application startup. This is especially relevant for cache side channels, where differences in these access patterns may reveal information about the application and its inputs to an attacker. When rerunning the previous experiment, we find that, for most applications, the trace remains identical when the application is started the same way. The only exceptions are `bash` and `sudo`, which show minor edit distances of 7 and 11 between their traces, and Firefox, with an edit distance of 156. This is not unexpected, as Firefox spawns multiple processes during startup, so slight differences in concurrency can significantly alter the access pattern.

When changing the input to the test applications, we observe major changes in some access patterns, while others remain largely the same. For instance, when replacing the `.c` file ending of the GCC test program with `.cpp`, we observe an edit distance of 230 between the resulting traces. In contrast, applications like `sudo` change their access patterns only slightly, e.g., based on the specified user, with the highest observed edit distance being 15. We conclude that input, especially in more complex applications, can significantly affect the zero-page's access patterns, allowing an attacker to infer it by observing the zero-page's cache

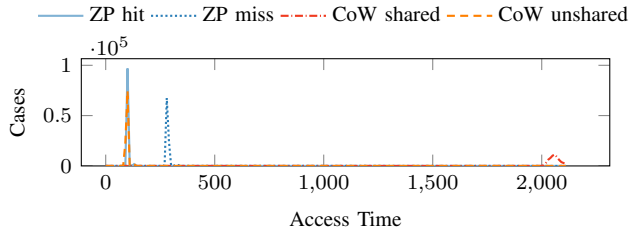

Figure 3: Different access times for cache hit/miss on the zero page and Copy-on-Write (CoW) write page faults vs. normal reads. The timing differences for Zepade attacks are much smaller, but the attack is repeatable.

state. In contrast, the access patterns of some applications provide a reliable fingerprint for identification.

3.4. Monitoring the Zero Page

To observe accesses to the shared zero page, we use cache-based side-channel attacks that rely solely on read behavior. Specifically, we use Flush+Reload [60], Evict+Reload [26], and Flush+Flush [15] to monitor whether a victim accesses a cache line belonging to the zero page. All three techniques exploit timing differences caused by the presence or absence of a cache line in the last-level cache and can be implemented from unprivileged user space.

This monitoring approach is fundamentally different from prior memory-deduplication attacks (cf. Section 7). Previous work mainly relies on *write-based* oracles: an attacker writes to a candidate page and infers whether it was deduplicated by measuring the write latency, which is dominated by the presence or absence of a CoW fault. Such attacks explicitly break sharing as part of the measurement. In contrast, our attacks are *read-based*. We only measure access times to the zero page and never perform writes to it. This distinction is crucial: any write to a zero page would immediately trigger CoW and destroy the shared physical backing, eliminating the channel we aim to exploit. Hence, write-based timing attacks are incompatible with zero-page monitoring.

Read-based Timing vs. Write-based Timing. Figure 3 contrasts the timing behavior of write-based and read-based measurements. Write accesses to a deduplicated page have a high latency of multiple thousand cycles caused by the unsharing of the page, clearly separating shared from unshared pages. However, once a write occurs, the page is no longer shared. In contrast, read-based cache attacks produce a stable timing signal corresponding to cache hits and misses, with a difference of 10 to 100 cycles, while preserving deduplication. Although the timing differences are harder to distinguish, read-based attacks allow continuous observation over long periods, enabling covert channels and fingerprinting attacks that rely on sustained sharing.

Note that a write-based oracle is possible only as a one-shot test per virtual zero-page mapping: the first write triggers CoW and reveals that the page was zero-backed, but it also replaces the shared physical page with a private page. An attacker can create many zero-backed virtual pages and repeat this one-shot test, but this yields a destructive sampling oracle rather than a continuous monitor. For this reason, we focus on read-based cache monitoring, which preserves sharing and supports sustained attacks.

4. Zepade Covert Channels

In this section, we evaluate the Zepade side channel using covert channels. We use covert channels as a measurement tool that provides a direct way to quantify whether zero-page accesses are observable across a given isolation boundary and how reliably they can be distinguished from noise. In Section 4.1, we describe the experimental setup. We evaluate native-code covert channels across different isolation boundaries in Section 4.2, and a browser-based covert channel in Section 4.3. The covert channels are not the paper's main security impact. The case studies in Section 5 use the same primitive for workload inference and sparse-data leakage.

4.1. Experimental Setup

Our evaluation demonstrates that zero-page-based cache side channels are observable and sufficiently stable across different systems. Table 1 provides an overview of the evaluated systems. While we observe minor differences in timing behavior, the behavior is consistent across platforms, and all attacks succeed on the tested systems.

Measurement Methodology. The sender encodes information by conditionally accessing one or more offsets on a zero-backed page. The receiver obtains its own zero-backed page and repeatedly probes the corresponding cache line using a cache-based timing primitive. For native code, we use Flush+Reload or Flush+Flush depending on the experiment. For browsers, where explicit cache flushing is unavailable, we use an Evict+Reload-style strategy. For each configuration, we report both throughput and reliability metrics. In particular, we compute true channel capacity from the raw symbol rate and the observed error rate when transmitting random bits, following the methodology used in prior covert-channel evaluations [27, 15, 37].

4.2. Native Code Channels

We first evaluate native-code covert channels under different isolation scenarios. These experiments distinguish three cases: always-on zero-page sharing within the same Linux kernel instance, container isolation, and virtual-machine isolation when host-side KSM zero-page merging is explicitly enabled.

Same Host. When running sender and receiver on the same host and pinning them to different cores, we observe low error rates below $0.02\,\%$ with a true channel capacity of $137\,\mathrm{kB/s}$ on an Intel Core i7-1365U and $45.69\,\mathrm{kB/s}$ on an AMD Ryzen 7 7800X3D. These results demonstrate that the signal induced by zero-page accesses is strong and reliable.

We also evaluate robustness to system noise by introducing memory pressure during transmission with `stress -m 1`. While noise increases error rates, the channel remains usable. On the Intel system, we achieve a true capacity of $111.6\,\mathrm{kB/s}$ in a noisy environment. The AMD system is less resilient to noise, with increased bit error and miss rates, but still achieves a true capacity of $28.89\,\mathrm{kB/s}$. Since our goal is not to maximize throughput, these experiments primarily show that the channel is exploitable under realistic background activity.

Cross-Container. We next evaluate the covert channel across container boundaries by running the sender and receiver in separate Docker containers on the same host. The channel remains fully functional, achieving a true capacity of $70.58\,\mathrm{kB/s}$ with an error rate of $1.8\,\%$ and $65.3\,\mathrm{kB/s}$ when running with memory pressure on the Intel system. On the AMD system, we achieve a true capacity of $44.75\,\mathrm{kB/s}$ and $33.14\,\mathrm{kB/s}$ under memory pressure. This is expected, as containers share the kernel and its memory management subsystem, including zero-page deduplication. These results show that zero-page-based covert channels cross container isolation boundaries.

Cross-VM with KSM Zero-Page Merging. Finally, we evaluate a virtual-machine configuration in which the sender and receiver run in separate Ubuntu 24.04 KVM guests using QEMU. This experiment requires host-side KSM with zero-page merging enabled. It therefore differs from the previous experiments: the observed cross-VM channel is not caused solely by the always-on guest-local Linux zero page, but by the host merging zero pages across guests. We include this experiment to characterize the risk of enabling KSM zero-page merging in virtualized environments, not to claim that zero-page sharing automatically crosses VM boundaries when KSM is disabled.

Compared to the same-host scenario, we observe higher miss rates, likely due to hypervisor scheduling noise and KSM-induced sharing behavior. Despite this degradation, the channel remains effective. On the Intel system, we achieve a true capacity of $62\,\mathrm{kB/s}$ with an error rate of $4.19\,\%$. On the AMD system, we achieve $20.81\,\mathrm{kB/s}$ with an error rate of $11.28\,\%$. We also evaluate the same configuration with KSM disabled and do not observe cross-VM zero-page sharing, confirming that the cross-VM result depends on explicit host-side deduplication policy.

4.3. Browser-Based Channel

We demonstrate that zero-page deduplication also enables a browser-based covert channel, despite browser sandboxing and site isolation. This experiment shows that JavaScript can obtain zero-backed memory and observe shared zero-page cache effects.

Threat Model. We assume an attacker-controlled receiver running in a Chromium-based browser tab. The sender runs either in a separate Chromium tab (including incognito mode) or in a different browser, such as Firefox. Neither browser has any configuration changes, and the sender and receiver execute on the same physical machine but are otherwise isolated by the browser sandbox.

Design and Implementation. The receiver allocates a large `Uint8Array` object that remains zero-initialized. In Chromium, pages after the first page of such an allocation are backed by the shared zero page.

Each measurement round proceeds as follows. First, the receiver evicts the relevant cache lines by sequentially accessing a large eviction buffer. The receiver measures access latencies to multiple candidate offsets to amplify the timing difference. Our implementation uses a counting thread, as described by Schwarz et al. [46], as the timer primitive; accessing two cache lines suffices to monitor the zero page. The receiver aggregates timing measurements across multiple repetitions to average out noise and outliers caused by the non-scheduled counting thread. The sender encodes information by conditionally accessing a specific zero-page offset. A logical one is transmitted by repeatedly accessing the target offset, while a logical zero is transmitted by remaining idle.

Evaluation. We evaluate the covert channel on Ubuntu 24.04 with Chrome 145 as the receiver. As senders, we use (1) another Chrome tab running in incognito mode and (2) Firefox 147 running concurrently on the same system. In both cases, the channel remains functional, confirming that zero-page deduplication persists across browser instances and different browser engines.

Using one-second symbol intervals, we reliably transmit $4\,\mathrm{bit/s}$. Although the bandwidth is lower than in native code, this result shows that browser code can access the primitive. Improving browser-side signal extraction is purely an engineering task that is outside the scope of this paper.

5. Case Studies

In this section, we present case studies that demonstrate the security implications of zero-page deduplication. We first describe an attack on ORAM schemes (Section 5.1). We then demonstrate that zero-page access patterns are distinct across applications and allow for fingerprinting of applications, their behavior, and websites, even across confidential VMs (Sections 5.2 to 5.5). Finally, we show how zero-page access patterns can trigger UI redressing attacks (Section 5.6), how to build an architectural covert channel based on `umwait` (Section 5.7), and how to use zero-pages as an alternative setup for ZombieLoad (Section 5.8).

5.1. Zero-Page Leakage in Linear-Scan ORAM Key-Value Stores

This case study demonstrates that zero-page-based side channels can break key assumptions of linear-scan ORAM constructions, as used, for example, in the confidential-VM setting by Zhang et al. [61]. We show that sparse data layouts combined with zero-page deduplication introduce a previously unexplored leakage vector that undermines the intended access-pattern obliviousness.

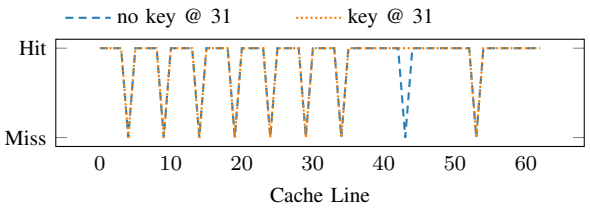

Figure 4: Accessed cache lines on the zero page for the same ORAM, once with key 31 present and once without.

Setup. In our case study, the ORAM store is backed by a large, sparsely populated array. Each entry is a packed struct holding a 2000 B value and an 8-byte length field, i.e., a 2008 B stride, similar to the setup in Zhang et al. [61]. The array is a page-aligned anonymous mapping, so entry i starts at offset $i \cdot 2008\,\mathrm{B}$, spanning one or two of the 4 kB pages. Populating an entry replaces the zero page only for the pages it actually writes. Only indices corresponding to existing keys contain non-zero entries, while the remaining array elements are empty. These empty elements are zero and, in practice, backed by the zero-page. A lookup or update operation performs a linear scan over the entire array, always accessing every element.

From a functional perspective, this scan reveals nothing about the memory content: the same loop executes for every operation. However, at the microarchitectural level, accesses to populated slots differ from those to empty slots. Accesses to empty slots resolve to the shared zero-page, whereas populated slots map to private physical pages. As a result, the zero-page is accessed at array indices corresponding to empty keys, and not accessed where real entries exist. We rely on Flush+Reload to monitor the zero page. By observing which zero-page cache lines are accessed during a linear scan, the attacker can infer which array indices correspond to empty slots. Indices that do not produce zero-page accesses reveal the presence of actual key-value entries.

Results. We find that zero-page access patterns can leak whether a key exists in the ORAM, as shown in Figure 4. Even though the scan order is fixed and independent of the queried key, the pattern of zero-page cache hits reveals which keys are present in the store. The 2008 B stride maps entries to distinct zero-page cache lines unless they are more than 50 indices apart, so collisions do not occur in our store; where slots do share a line, e.g., in bitmaps, a hit only reveals that at least one slot of the group is empty. To verify that the implementation does not leak information in the traditional sense, e.g., due to compiler optimizations [11, 21, 43], we analyze our implementation with DATA [56]. DATA does not show secret-dependent memory accesses, as the implementation is data-oblivious in the traditional sense [18].

The ORAM case study illustrates the main qualitative difference between Zepade and conventional cache attacks. A classical address-trace adversary observes which virtual or physical locations are accessed. In our setting, the access order and the virtual addresses are fixed and independent of the key. Leakage via zero-page deduplication violates a core security goal of linear-scan ORAM designs: memory access patterns should not reveal information about stored data. Importantly, the leakage occurs without observing memory contents or distinguishing read and write accesses. Moreover, there is no secret-dependent (order of) memory accesses on the virtual-address level. The attack solely relies on the implicit sharing introduced by zero-page deduplication and the resulting cache effects. Thus, the side channel reveals properties of memory contents and allocation state, not a secret-dependent branch or address choice in the program. This distinction makes zero-page deduplication relevant for sparse arrays, hash tables with empty slots, bitmaps, compressed data structures, and ORAM-style scans whose security arguments assume that touching every logical entry removes address-pattern leakage. Mitigating this leakage requires ensuring that shared zero-pages do not back empty slots. This can be achieved by explicitly initializing all entries of the backing array.

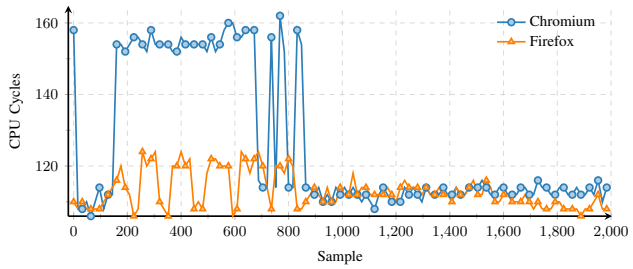

Figure 5: First 2000 Flush+Flush measurements at zero-page offset 0 while starting Chromium and Firefox on an Intel Core i7-1365U.

5.2. Fingerprinting Applications

In this case study, we demonstrate how monitoring the cache lines on the zero-page allows fingerprinting of applications that are started. Different applications have distinct zero-page access patterns during startup. Figure 5 shows such characteristic zero-page accesses during the startup of Chromium and Firefox. We can observe these access patterns via cache-based side channels and train a classifier to identify which application is being started.

Threat Model. To fingerprint different victim applications, we assume that the attacker can execute an unprivileged program on the same machine where the victim's target application runs. Since we want to classify the programs the victim is starting, we assume the attacker's program starts before any of the targeted victims' programs.

Setup. We train a classifier using traces collected by monitoring zero-page accesses during application startup.

To assess the effectiveness of our fingerprinting technique across different settings, we record traces for containerized server applications and user applications running on the host system on an Intel Core i7-1365U and an AMD Ryzen 7 7800X3D CPU. For the containerized server

application, we chose Caddy, Nginx, HAProxy, and Traefik as web servers, and PostgreSQL, MySQL, and MariaDB as database servers. These applications are commonly deployed in containers and used in cloud environments. We chose Chromium, Firefox, Thunderbird, Signal, Spotify, and the LibreOffice suite as user applications. This gives a mix of application types, GUI toolkits, and runtime environments, such as Electron, GTK, and Qt.

Traces are recorded by repeatedly probing the cache state for different offsets within the zero-page using Flush+Flush. Because zero-pages are predominantly used during initialization, we measure 300 000 sweeps over the zero page after the application starts. The data is preprocessed to reduce noise and compress the traces. We replace outliers with the mean value of the trace and compress each trace by accumulating values into buckets of size 300. For each bucket, we compute the mean and z-scaled cumulative sum of its values and use them as features for the classifiers. We use a simple pipeline for classification that employs time-series data, multivariate transformers from the ROCKET family [8], and a Ridge classifier. The classification pipeline consists of a multivariate MiniRocket [9] transformer to extract features from the traces, a standard scaler to normalize the features, and a Ridge regression to perform the classification based on the extracted features. Appendix D contains more details on the classification pipeline and the chosen parameters.

Results. Performing a 5-fold cross-validation, we evaluate the classification accuracy of different classifiers. For each application, we have recorded 25 traces, each containing 300 000 measurements. We observe that for user applications, all classifiers achieve similar mean accuracies of around 80.3 % with a standard deviation of 5.4 % for traces collected on an AMD Ryzen 7 7800X3D, and 94.9 % (± 2.6 %) for traces collected on an Intel Core i7-1365U. For containerized server applications, we can achieve mean accuracies of around 81.1 % with a standard deviation of 5.0 % on the AMD Ryzen 7 7800X3D and around 90.9 % (± 3.8 %) on the Intel Core i7-1365U. Following the cross-container covert channel (cf. Section 4.2), which shows that zero-page-based side channels are effective across container boundaries, we repeat the classification for the server applications using traces recorded in a Docker container. In this setting, we repeat the same classification procedure and achieve accuracies within 2 % of those in the native setting, confirming that the classification remains effective across container boundaries. Thus, different applications indeed have distinguishable access patterns across the zero-page. Hence, an attacker can identify user and server applications on a target system with high accuracy.

5.3. Fingerprinting Application Behavior

In this case study, we show that zero-page access patterns can not only identify different applications but also leak more specific application behaviors.

Setup. As browsers make heavy use of zero-pages, even after the initial startup, we perform different operations within the Chromium web browser. These actions include opening a new tab, opening the settings page, starting a YouTube video, playing a song on SoundCloud, and clicking a link from a Google search result. Furthermore, we load videos with different codecs (VP9, H.264, and H.265) and a video with no audio. To synchronize measurement recording, the measurement process is triggered via a mouse click by monitoring the evdev interface. However, by monitoring the zero-page, an attacker could identify the start of an action without any external synchronization.

Results. The results show that different actions within Chromium can be identified with high accuracy. Performing a 5-fold cross-validation, we achieve a mean accuracy of around 83 % with a standard deviation of 8.1 % on an Intel Core i7-1365U. This indicates that an attacker can identify which application a victim is using and the actions the victim performs within it.

5.4. Fingerprinting Websites

In this case study, we show that monitoring zero-page accesses enables fingerprinting of websites that a user visits.

Setup. To get a representative sample of popular websites, the evaluation is based on the Tranco list, which provides a reliable ranking of the most popular websites [36]. We select the top 100 unique websites reachable via Chromium and record 50 traces, each with 300 000 measurements, per website. To separate zero-page effects caused by network/TLS processing from those caused by full browser startup and rendering, we additionally include `curl` as a minimal network client. As with Chromium, we let `curl` connect to each website and record 50 traces, each with 300 000 measurements. For requests, we also send the `Cache-Control: no-cache` and `Pragma: no-cache` headers to prevent caching, and a generic user-agent header to avoid serving different content based on the user-agent or on websites blocking requests from `curl`. We run experiments on an Intel Core i7-1365U and an AMD Ryzen 7 7800X3D CPU.

Results. When using Chromium, we achieve mean accuracies of 90.9 % with a standard deviation of 3.8 % on an Intel Core i7-1365U and around 95.7 % (± 0.5 %) on an AMD Ryzen 7 7800X3D for a 5-fold cross-validation.

When using `curl`, we only achieve mean accuracies of around 1.7 % on both systems, as expected due to the small number of zero pages used in `curl` (cf. Figure 2). The low accuracy for `curl` supports our interpretation that the browser experiment is not merely detecting remote endpoints, but richer browser-side behavior. An attacker can use this to identify visited websites with high accuracy.

5.5. Fingerprinting Applications in Confidential Virtual Machines

We evaluate our approach in a confidential virtual machine by targeting the zero page of a VM running under Intel TDX. While TDX prevents a malicious hypervisor

from directly accessing guest memory contents, prior work has shown that cache side channels remain effective despite these restrictions [38, 16]. As a result, we can still mount our zero-page-based fingerprinting attack inside a TDX VM to identify applications during startup.

Threat Model. We assume a privileged attacker who controls the hypervisor on a TDX-enabled host. The attacker additionally has access to the guest kernel binary and can therefore compute the relative offsets of kernel symbols, including the zero page. This assumption is realistic, as the kernel binary is typically stored unencrypted in the VM disk image and can be accessed by a malicious host operator.

Experimental Setup. All experiments are conducted on a single-core TDX VM with $8\,\mathrm{GB}$ of memory, running Ubuntu 24.04 with Linux 6.8.0 on an Intel Xeon 6736P (Granite Rapids) CPU. The host system also runs Ubuntu 24.04 with Linux 6.8.0 and TDX Module version 2.0.

As TDX VMs are deployed in cloud environments, we focus on containerized server applications. Inside the TDX VM, we use Docker to run several web servers, namely Caddy, HAProxy, Nginx, and Traefik, as well as database systems including MariaDB, MySQL, and PostgreSQL.

Locating the Zero Page. TDX prevents the hypervisor from directly inspecting the guest's memory layout. Therefore, before mounting our fingerprinting attack, we must identify the guest kernel's zero page. It is sufficient to locate the zero page in the guest-physical address space, as TDX allows the untrusted hypervisor to resolve guest-physical to host-physical addresses. The kernel base address, and thus the zero page location, is randomized on every VM boot.

To infer the kernel layout, we rely on cache side-channel observations. Specifically, we trigger deterministic kernel activity by repeatedly connecting to the guest's SSH server and disconnecting without sending data. The attacker does not authenticate to the guest, does not require guest credentials, and does not execute code inside the guest. Accepting an incoming connection causes a predictable sequence of kernel memory accesses that can be observed using Flush+Flush. This does not assume SSH login access. The malicious host only initiates unauthenticated TCP connection attempts to an externally reachable service and observes the resulting guest-kernel activity. In our setup, we use SSH because it is commonly enabled in cloud VMs and produces a stable packet-processing path. The same locating strategy can be applied to other network-facing services, provided they induce repeatable guest-kernel accesses.

We generate correctly aligned candidate kernel base addresses in guest-physical memory and add the known offset of `ip_rcv`, which processes incoming packets and is thus reliably accessed. For each candidate, we initiate a TCP connection and probe the corresponding address using Flush+Flush for 32 iterations. Discarding candidates with fewer than two cache hits typically reduces the search space from more than $10\,000$ to fewer than 100 addresses.

Among the remaining candidates, we observe a stable pattern of addresses with constant offsets relative to one another. For a given kernel binary, we determine these offsets by mounting the same attack on a dummy VM. During the actual attack, we add the known offsets to each remaining candidate and count how many of the resulting addresses are cached. In almost all cases, the correct `ip_rcv` address is the candidate that yields the highest number of cache hits. Once we identify the `ip_rcv` address, we can derive the zero-page address by adding a fixed offset.

We repeat this procedure 100 times, rebooting the VM before each attempt to re-randomize the memory layout. Across all runs, we achieve a success rate of $100\,\%$ and an average runtime of $1.99\,\mathrm{s}$, demonstrating that locating the zero page in a TDX VM is both fast and reliable.

Fingerprinting Applications. We preprocess and classify the recorded traces using the same methodology as in our non-TDX experiments. Even under TDX, we successfully fingerprint different containerized server applications based on their zero-page access patterns. The results are consistent with those obtained outside confidential VMs.

Using 5-fold cross-validation, we achieve a mean classification accuracy of approximately $92.9\,\%$, with a standard deviation of $2.0\,\%$, on the Intel Xeon 6736P (Granite Rapids) CPU. These results demonstrate that an attacker controlling the hypervisor of a TDX-enabled host can identify applications running inside a confidential VM by monitoring zero-page activity, despite the isolation guarantees provided by TDX.

5.6. Zero-page Trigger for UI Redressing

In this case study, we show that zero-page access patterns during application startup can serve as a reliable trigger for user interface (UI) redressing attacks [29]. UI-redressing attacks exploit the user's expectations about application behavior and timing. Prior work [14] demonstrated that page-cache attacks can infer application launches and synchronize malicious UI elements accordingly.

Threat Model. We assume code execution on the same host and a restricted placement of the attacker, where unrestricted process enumeration may not be available. These settings include containers, sandboxes, or browser-like environments.

Setup. We analyze the startup behavior of `sudo`, a common application that prompts users for their password. During initialization, `sudo` accesses a sequence of zero-initialized memory locations that results in a distinctive zero-page access pattern. In our experiments, we observe zero-page accesses at fixed offsets within the zero-page, including offsets `0xae9`, `0x60`, `0xa0`, `0x0`, `0x434`, `0x42c`, and `0x448`. These offsets are consistent across executions and form a fingerprint that uniquely identifies the startup of `sudo`. Note that, as these are page offsets, they are also independent of ASLR.

An attacker process continuously monitors these offsets using Flush+Reload. Once the attacker detects cache hits on all, or a sufficient subset to be robust in the presence of noise, of the characteristic offsets within a short time window, the attacker can infer with high confidence that `sudo` has been launched. Upon detection, the attacker immediately presents a spoofed password prompt, e.g., as a graphical dialog window placed in the foreground. The prompt's timing

closely matches that of the expected legitimate prompt. As a result, users might enter their password into the attacker-controlled window, believing it to be authentic.

Results. We successfully evaluated this attack on an Ubuntu 24.04 system running on an Intel Core i7-9700K. In repeated experiments, we reliably detect the zero-page access pattern of `sudo`. With $160\,\mathrm{ms}$, the detection latency is sufficiently small to present the malicious UI prompt before or concurrently with the legitimate password request. These results show that zero-page-based triggers enable highly accurate and timely UI redressing attacks.

5.7. Architectural Covert Channel via `umwait`

This case study shows that zero-page deduplication enables a purely architectural covert channel based on `umwait` [62]. By combining physically shared zero-pages with architectural monitoring instructions, two isolated processes can communicate without explicit shared memory or reliance on eviction sets.

Background. Zhang et al. [62] demonstrated that the `umwait` instruction can be exploited to build high-resolution architectural side channels and covert channels. The key observation is that `umwait` allows a CPU to enter a low-power wait state until a monitored cache line changes state. The instruction provides architectural feedback via the carry flag, indicating whether a cache-line event terminated the wait or a timeout did. The monitored address is tracked at cache-line granularity and tied to the physical address, not the virtual address. In the original setup, the attacker and victim must share memory to operate on the same physical cache line. This typically restricts attacks to native code with access to explicit shared memory mappings.

Setup. In our setup, the sender and receiver are isolated processes that both map a zero-page at some virtual address. The receiver monitors a cache line within the zero-page using `umwait`. The sender encodes information by conditionally issuing a `prefetchw` instruction to the same zero-page cache line. As shown by Zhang et al. [62], `prefetchw` has effects similar to a write on cache coherence, even though it does not perform an architecturally visible store. Importantly, because `prefetchw` is not a real write, it does not trigger a CoW fault. The zero-page remains shared, and the physical backing remains unchanged. Nevertheless, the cache line state transition caused by `prefetchw` is sufficient to wake the receiver from `umwait`. By observing the architectural carry flag set by `umwait`, the receiver can determine whether the sender accessed the cache line.

Results. On an Intel i9-12900H, a non-optimized version of our covert channel transmits at $678\,\mathrm{B/s}$ with no transmission errors. This is roughly a factor of 8 faster than the one initially reported by Zhang et al. [62].

5.8. Alternative ZombieLoad v3 Setup

In this case study, we show that zero-page deduplication enables an alternative and more flexible setup for ZombieLoad v3-style attacks [45]. By leveraging implicitly shared zero-pages, we can satisfy ZombieLoad v3's shared-memory requirement without relying on explicit shared memory, broadening the attack surface.

Background. ZombieLoad v3 is a microarchitectural data sampling attack that leaks data transiently forwarded from internal CPU buffers across execution contexts. A key requirement of the ZombieLoad v3 setup is the use of shared memory, which lets an attacker access the same physical page via two different virtual addresses [45]. In the original setup [17], this shared memory is established via explicit shared memory, readily available in native code but difficult or impossible to use in restricted environments such as browsers [10]. Zero-page deduplication fundamentally changes this assumption. Because all zero-page mappings resolve to the same physical page, they naturally satisfy the shared-memory requirement of ZombieLoad v3. Compared to prior approaches, this setup is both simpler and more robust. The implications are particularly relevant for browser-based and sandboxed attack scenarios. Prior work by Easdon et al. [10] used memory deduplication to recreate shared memory within the browser. However, this approach relies on content-based deduplication, which is slow, probabilistic, and often disabled due to security concerns. Zero-page deduplication, in contrast, is always active, deterministic, and faster, as it does not require scanning memory contents.

Results. We implemented the ZombieLoad v3 variant from Easdon et al. [10] using zero-page instead of memory deduplication. On a ZombieLoad-affected Intel Xeon E3-1505M v5 (microcode 0xd6), running Ubuntu 20.04.5, we successfully reproduce leakage of around $1\,\mathrm{B/s}$, in line with original results [10].

6. Mitigations

In this section, we implement and evaluate mitigations preventing Zepade attacks. In Section 6.1, we focus on operating-system-level defenses, as zero-page sharing is a kernel-managed optimization. Additionally, Section 6.2 presents user-space mitigations as a short-term mitigation. Our goal is to eliminate unintended sharing that enables cache-based leakage while preserving system performance.

6.1. Operating-system Mitigations

An intuitive mitigation is to disable memory deduplication entirely. In virtualized environments, disabling KSM in the host configuration also disables zero-page deduplication. While effective for virtual machines, this approach does not address zero-page sharing on non-virtualized systems. On such systems, zero-page deduplication always remains active even when general-purpose deduplication is disabled. Unfortunately, Linux does not provide a runtime, boot-time, or compile-time flag to disable zero-page sharing.

Thus, to address zero-page-based leakage, we implement a kernel-level mitigation that prevents zero-page sharing. Our mitigation is based on a small patch to the Linux kernel that disables the reuse of the global zero-page for user-space mappings. Our patch is based on Linux kernel 6.8.0,

the default kernel on Ubuntu 24.04. Instead of mapping a single, read-only zero-page into multiple address spaces, the kernel allocates a private zero-filled physical page when a zero-initialized mapping is first accessed. This preserves the semantics of zero-initialized memory while removing the shared physical backing, preventing Zepade attacks.

Kernel Patch Design. The patch modifies the Linux memory management subsystem that handles anonymous page faults. In the unmodified kernel, a read fault on a zero-initialized anonymous page maps the global zero-page. It marks the page table entry read-only, plus a software-usable bit indicating the zero page. Our patch intercepts this path and replaces the zero-page mapping with a freshly allocated, explicitly zeroed private page. Subsequent accesses operate on a process-private physical page, eliminating cache-line sharing with other processes or virtual machines.

This design follows existing kernel conventions for page fault handling and CoW. It reuses the same allocation and zeroing mechanisms already employed when handling write faults on zero-pages. As a result, the patch is minimal, self-contained, and introduces no new abstractions or complex bookkeeping. Importantly, it does not affect file-backed mappings, shared memory, or other mechanisms that rely on explicit sharing. Future work could investigate a per-process mitigation, similar to the PreFence mitigation [42].

Security Argument. By ensuring that a globally shared physical page never backs zero-initialized anonymous memory, the mitigation removes the root cause of the presented attacks. Since our attacks rely on shared physical addresses mapping to shared cache lines, eliminating this sharing directly prevents all attacks based on zero-page accesses. Thus, our mitigation provides a strong security guarantee.

Performance Evaluation. We evaluate the performance impact of our mitigation using two complementary benchmarks on an AMD Ryzen 9 9950X3D with 64 GB RAM, running Ubuntu 24.04 with kernel 6.8.0.

First, we run the SPEC CPU 2017 benchmark suite to measure the performance of the compute-intensive workload. Across all benchmarks, we observe no measurable runtime overhead compared to an unmodified kernel. This result is expected, as SPEC CPU workloads rarely trigger first-touch faults on large numbers of zero-initialized pages during steady-state execution. Thus, the mitigation has a negligible impact on CPU-bound applications.

Second, we evaluate memory consumption using a benchmark adapted from Oliverio et al. [30]. We sequentially start four virtual machines, each running an Apache web server, with a 30 s delay between launches. This workload stresses memory allocation during process and VM initialization, highlighting the memory-saving benefits of zero-page sharing.

Results. The results show that disabling zero-page sharing slightly increases memory consumption (cf. Figure 6). Compared to the baseline system with zero-page deduplication, our mitigation fully prevents Zepade attacks. In our benchmark, we measure an average memory overhead of 1.96 %. After the system has stabilized, this memory overhead drops to 0.78 %. This makes it suitable for deployment on both native systems and confidential virtual machines.

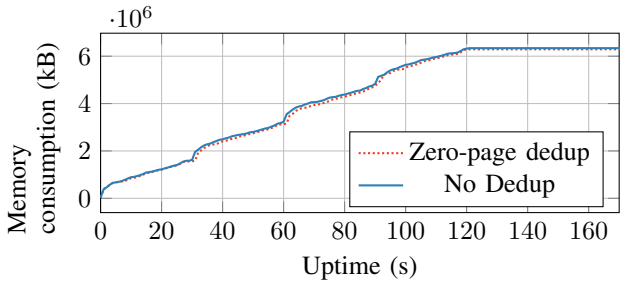

Figure 6: Memory consumption over time for sequential VM startup with different deduplication strategies.

The measured overhead represents the benefit the system loses from no longer mapping all zero-filled anonymous pages to a single global physical page. Thus, our evaluation quantifies the cost of removing the optimization that creates the side channel. In our workloads, this lost optimization benefit is small.

6.2. User-space Mitigation

While kernel-level mitigations provide security guarantees by eliminating the root cause, they require changes to the operating system and are therefore not immediately deployable on existing systems. As a short-term mitigation, we propose a user-space mitigation that prevents zero-page sharing without requiring kernel modifications.

The core idea of the user-space mitigation is to proactively eliminate zero-page mappings by ensuring that private physical pages are allocated before an attacker can observe them. This is achieved by iterating over a process's virtual address space and explicitly writing to each writable page, thereby triggering allocation of a private physical page. Concretely, the mitigation walks all relevant memory regions obtained from `/proc/self/maps` and performs a write access to one byte per page. To avoid overwriting any data, it first reads the byte at the target address. As a result, the operating system resolves the access by allocating a private, zero-filled physical page instead of mapping the global zero-page. After this initialization phase, no zero-page-backed mappings remain in the process address space.

The implementation follows the same principles as the kernel-space mitigation but shifts responsibility to potential targets in user space. It relies solely on standard interfaces to inspect a process's memory layout and requires no special privileges. The mitigation can be applied selectively, for example, during process startup or before executing security-sensitive code. Importantly, it preserves the semantics of zero-initialized memory and does not affect correctness, as all pages remain logically zero-filled after initialization. Additionally, we hook the functions that lead to new zero-page allocations (cf. Table 2 in the appendix) to cause a page fault after allocation.

Security Argument. By eagerly mapping all zero-initialized pages, the user-space mitigation removes the shared physical backing that enables zero-page-based side channels. Since each page is backed by a private physical frame after initialization, cache-based attacks relying on shared zero-pages are no longer possible. The security guarantees therefore match those of the kernel-level mitigation but are limited to the protected process itself and do not extend system-wide. Thus, covert channels remain possible, as processes can choose not to apply the mitigation.

Performance Considerations. The main drawback of the user-space mitigation is its overhead. Pre-faulting all pages introduces additional memory writes and page faults during initialization, increasing startup time and memory pressure. Unlike the kernel-level mitigation, which still relies on demand paging, this approach requires more memory. It incurs additional overhead from hooking functions, parsing address-space information, and writing to memory.

In practice, this consideration is acceptable for short-lived or security-critical processes where preventing information leakage outweighs startup costs. However, for large address spaces or memory-intensive applications, the eager allocation can significantly increase memory consumption. For this reason, we view the user-space mitigation as a temporary or situational defense, suitable until kernel-level support is available.

6.3. Application-level Mitigation

While operating-system and user-space mitigations address zero-page sharing at a global or process level, application developers can also reduce the attack surface by avoiding characteristic zero-page access patterns. As shown in Section 3.2, access patterns can leak side-channel information. Additionally, Section 5.6 shows that characteristic access sequences during application startup can serve as reliable triggers for UI redressing. Eliminating such patterns at the application level can therefore mitigate these attacks, even on systems where zero-page sharing remains enabled. To help developers identify these issues, we provide a tracing tool that records accesses to zero-pages at cache-line granularity (see Section 3.3). By running an application with this tracer, developers can obtain a detailed profile of which zero-page offsets are accessed during execution, including the order and the instruction that accesses them. This allows pinpointing code paths that produce distinctive or repetitive zero-page access patterns. Once identified, such patterns can often be eliminated with minimal changes. Typical strategies include explicitly initializing memory before reading it and restructuring data layouts to avoid sparse or padded accesses. Notably, application-level mitigations provide best-effort protection and do not eliminate the underlying side channel. They are therefore complementary to, but not a replacement for, operating-system-level defenses. However, for security-sensitive applications such as authentication helpers or privilege escalation tools, reducing or removing distinctive zero-page access patterns can significantly raise the bar for practical exploitation and mitigate UI redressing attacks in the absence of system-wide fixes.

7. Related Work

In this section, we summarize the most relevant lines of work and position Zepade accordingly.

Page-Deduplication Attacks. Suzaki et al. [49, 50] first showed that write latency can reveal whether a page has been deduplicated: if the page is shared and write-protected, a write triggers a CoW fault and becomes measurably slower. Owens et al. [33] applied this observation to detect co-resident virtual machines and infer the guest operating system by probing for pages that are characteristic of specific images. Xiao et al. used deduplication as a signal to detect virtualization [58]. Subsequent work improved channel designs and explored robustness in realistic settings, including cross-VM scenarios [41, 20]. Gruss et al. [13] demonstrated that deduplication attacks can be mounted even from JavaScript, enabling the detection of applications and websites. Schwarzl et al. [47] showed that deduplication-induced effects can be observed without local code execution, turning deduplication into a remote oracle. In contrast to previous work, Zepade can mount similar attacks without requiring memory deduplication to be enabled. Moreover, Zepade does not require any writes to trigger page unmerging.

Deduplication oracles were also used to undermine address randomization. Barresi et al. [5] showed how to break ASLR by testing candidate pages for deduplication. Kim et al. [22] refined these ideas to break KASLR more efficiently by exploiting kernel page properties and deduplication behavior.

Page Deduplication for Rowhammer. Bosman et al. [7] investigated more powerful primitives, including inferring contents under certain conditions and composing deduplication side channels with Rowhammer-style fault attacks to obtain end-to-end compromises in browsers. Razavi et al. [39] combined deduplication (to establish shared physical backing) with Rowhammer to corrupt victim memory and escalate privileges. Both are orthogonal to Zepade, as they use deduplication as a building block for Rowhammer.

CoW in Other Layers. Palfinger et al. [34] showed that CoW semantics in storage layers (e.g., CoW filesystems) can create cross-VM side channels that reveal information about file contents, independent of the hypervisor.

Secure Deduplication. Previous attacks on deduplication resulted in mitigations, such as disabling or restricting cross-domain deduplication in multi-tenant settings [47]. Researchers have also proposed mechanisms that retain some memory benefits while limiting information leakage. Oliverio et al. [30] studied secure variants of deduplication and quantified the memory and performance trade-offs under realistic consolidation workloads. Lindemann et al. [24] proposed a modification to KSM that introduces fake deduplication to make CoW-based oracles unreliable, while still providing memory savings. However, defenses primarily target content-based scanning and CoW-write oracles, not special cases such as zero-page sharing.

8. Conclusion

We showed that zero-page deduplication is an always-on source of shared physical state that persists even when conventional page deduplication is disabled. By exploiting cache effects induced by read-only accesses to the globally shared zero page, Zepade attacks enable non-destructive observation of zero-backed memory activity, creating practical covert channels across processes, containers, browsers, and virtual machines, including confidential VMs. Our case studies demonstrate the security impact, ranging from application and website fingerprinting to UI redressing triggers, simplified setups for transient-execution attacks, and leakage that bypasses traditional constant-time notions in ORAM-style designs. Finally, we show that the root cause can be addressed: eliminating global zero-page reuse in Linux removes the side channel with negligible performance impact and only small memory overhead, making it a practical mitigation.

9. Acknowledgements

We want to thank our anonymous reviewers for their comments and suggestions. This work was supported in part by Google through a Sponsored Research Agreement. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Protecting VM Register State With SEV-ES, 2023.

[2] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020.

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

[4] Seungyeon Bae, Taehun Kim, Woomin Lee, and Youngjoo Shin. Exploiting memory page management in ksm for remote memory deduplication attack. In *International Conference on Information Security Applications*. Springer, 2023.

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

[6] Daniel J. Bernstein. Cache-Timing Attacks on AES, 2005.

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

[8] Angus Dempster, François Petitjean, and Geoffrey I Webb. Rocket: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*, 34(5), 2020.

[9] Angus Dempster, Daniel F Schmidt, and Geoffrey I Webb. Minirocket: A very fast (almost) deterministic transform for time series classification. In *ACM SIGKDD conference on knowledge discovery & data mining*, 2021.

[10] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. Rapid Prototyping for Microarchitectural Attacks. In *USENIX Security*, 2022.

[11] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. Do compilers break constant-time guarantees? In *FC*, 2025.

[12] Mel Gorman. *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[13] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security*. Springer, 2015.

[14] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page Cache Attacks. In *CCS*, 2019.

[15] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[16] Tristan Hornetz, Hosein Yavarzadeh, Albert Cheu, Adria Gascon, Lukas Gerlach, Daniel Moghimi, Phillipp Schoppmann, Michael Schwarz, and Ruiyi Zhang. TDXRay: Microarchitectural Side-Channel Analysis of Intel TDX for Real-World Workloads. In *IEEE S&P*, 2026.

[17] IAIK. ZombieLoad PoC (GitHub), 2019.

[18] Intel Corporation. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations, 2020.

[19] Intel Corporation. Intel Trust Domain Extensions (Intel TDX) Module Base Architecture Specification, 2021.

[20] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *RAID'14*, 2014.

[21] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with MSVC . In *CANS*, 2016.

[22] Taehun Kim, Taehyun Kim, and Youngjoo Shin. Breaking kaslr using memory deduplication in virtualized environments. *Electronics*, 10(17), 2021.

[23] Byeoksan Lee, Seong Min Kim, Eru Park, and Dongsu Han. Memscope: analyzing memory duplication on android systems. In *Asia-Pacific Workshop on Systems*, 2015.

[24] Jens Lindemann. Faking deduplication to prevent timing side-channel attacks on memory deduplication. In *ACSAC*. IEEE, 2024.

[25] Jens Lindemann and Mathias Fischer. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines. In *ACM Symposium on Applied Computing*, 2018.

[26] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security*, 2016.

[27] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the other side: SSH over robust cache covert channels in the cloud. In *NDSS*, 2017.

[28] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. Severed: Subverting amd's virtual machine encryption. In *EuroSec*, 2018.

[29] Marcus Niemietz and Jörg Schwenk. UI Redressing Attacks on Android Devices. *Black Hat Abu Dhabi*, 2012.

[30] Marco Oliverio, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Secure page fusion with vusion. In *SOSP*, 2017.

[31] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, 2015.

[32] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

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

[34] Gerald Palfinger, Bernd Prünster, and Dominik Ziegler. Prying cow: Inferring secrets across virtual machine boundaries. In *ICETE*, 2019.

[35] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 2016.

[36] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *NDSS*, 2018.

[37] Fabian Rauscher, Carina Fiedler, Andreas Kogler, and Daniel Gruss. A systematic evaluation of novel and existing cache side channels. In *NDSS*, 2025.

[38] Fabian Rauscher, Luca Wilke, Hannes Weissteiner, Thomas Eisenbarth, and Daniel Gruss. Tdxploit: Novel techniques for single-stepping and cache attacks on intel tdx. In *USENIX Security Symposium*. 2025.

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

[40] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS*, 2009.

[41] Hong Rong, Huimei Wang, Jian Liu, Xiaochen Zhang, and Ming Xian. Windtalker: an efficient and robust protocol of cloud covert channel based on memory deduplication. In *International Conference on Big Data and Cloud Computing*, 2015.

[42] Till Schlüter and Nils Ole Tippenhauer. Prefence: A fine-grained and scheduling-aware defense against prefetching-based attacks. In *EuroS&P*, 2025.

[43] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking bad: How compilers break constant-time implementations. In *ACM Asia Conference on Computer and Communications Security*, New York, NY, USA, 2025.

[44] Michael Schwarz. Software-based Side-Channel Attacks and Defenses in Restricted Environments. In *PhD Thesis*, 2019.

[45] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[46] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*, 2017.

[47] Martin Schwarzl, Erik Kraft, Moritz Lipp, and Daniel Gruss. Remote memory-deduplication attacks. In *Network and Distributed System Security Symposium*, 2022.

[48] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses. In *USENIX Security Symposium*, 2021.

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

[50] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Implementation of a memory disclosure attack on memory deduplication of virtual machines. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 96(1), 2013.

[51] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. 2015.

[52] Fabian Thomas and Michael Schwarz. RISCy Cache Coherence: Timer-Free Architectural Cache Attacks via Instruction/Data Cache Incoherence. In *IEEE S&P*, 2026.

[53] Fabian Thomas, Michael Torres, Daniel Moghimi, and Michael Schwarz. ExfilState: Automated Discovery of Timer-Free Cache Side Channels on ARM CPUs. In *CCS*, 2025.

[54] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[55] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark

Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.
[56] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *USENIX Security*, 2018.
[57] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. A covert channel construction in a virtualized environment. In *ACM conference on Computer and communications security*, 2012.
[58] Jidong Xiao, Zhang Xu, Hai Huang, and Haining Wang. Security implications of memory deduplication in a virtualized environment. In *International Conference on Dependable Systems and Networks (DSN)*, 2013.
[59] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.
[60] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.
[61] Ruiyi Zhang, Albert Cheu, Adria Gascon, Daniel Moghimi, Phillipp Schoppmann, Michael Schwarz, and Octavian Suciu. SNPeek: Side-Channel Analysis for Privacy Applications on Confidential VMs. In *NDSS*, 2026.
[62] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security*, 2023.

Appendix A.
Zero-page Usage in Allocation Mechanisms

In Section 3.3, we discussed how applications make use of zero-pages. We extend this overview with Table 2, which summarizes which common memory regions and allocation mechanisms use zero-page backing on Linux.

Appendix B.
Testing Zero-page Usage in Applications

Section 3.3 describes an experiment in which we count the number of zero pages used when starting different applications. These applications were invoked as the root user on an x86_64 Ubuntu 24.04 installation with the commands shown in Table 3. The `hello_world.c` file contains the following code:

TABLE 2: Zero-page (ZP) usage for common memory regions and allocation mechanisms on Linux (kernel 6.8.0, Ubuntu 24.04), based on empirical measurements.

Memory region / allocation	ZP backed
Stack pages	✓
Uninitialized global/static variables	✓
Thread-local storage	✗
`malloc`	✓
`calloc`	✗
`realloc` (growing)	✓
`aligned_alloc`, `valloc`	✓
`sbrk` (new pages)	✓
`mmap` anonymous, private	✓
`mmap` anonymous, private, `MAP_POPULATE`	✗
`mmap` anonymous, shared	✗
`mmap` file-backed (private/shared)	✗
`mmap` anonymous, `PROT_READ`	✓

TABLE 3: Commands used for tracing zero-page accesses.

Command	Version
`bash`*	5.2.21
`curl https://www.wikipedia.org/`	8.5.0
`firefox-bin https://www.wikipedia.org/`	147.0.2
`gcc hello_world.c -o hello_world`	13.3.0
`python3 -c 'print("Hello World")'`	3.12.3
`sudo -i`*	1.9.15p5
`whoami`	9.4

*Interactive session was immediately terminated with `exit`.

```c
#include <stdio.h>
int main() {
  printf("Hello World!\n");
  return 0;
}
```

Appendix C.
System Configuration

In Section 4.2 we implemented a covert channel for cross-VM communication. Appendix C shows the full KSM configuration used for our experiments, and the following listing shows the modifications we made to the `QEMU` configuration.

```
<cpu mode="host-passthrough" check="none"
     migratable="off"/>
<clock offset="utc">
  <timer name="kvmclock" present="yes"/>
</clock>
```

TABLE 4: Configuration values for KSM written to the respective files in `/sys/kernel/mm/ksm/`. Files marked with a * indicate that we have changed the default value.

Configuration file (Modified)	Configured Value
advisor_max_cpu	70
advisor_max_pages_to_scan	30000
advisor_min_pages_to_scan	500
advisor_mode	[none] scan-time
advisor_target_scan_time	200
max_page_sharing	256
merge_across_nodes	1
run*	1
sleep_millisecs	20
smart_scan	1
use_zero_pages*	1

Appendix D. Fingerprinting Classification

Here, we evaluate the classification accuracy of our pipeline across different bin sizes and feature modes. We use the same classification pipeline across the fingerprinting case studies (cf. Section 5.2, Section 5.3, Section 5.4, and Section 5.5). Specifically, we compact samples into fixed-size bins, extract features using the MiniRocketMV transformer, and classify with a Ridge regression classifier. Figure 7 shows the classification accuracy of both transformers across different bin sizes and feature modes on an Intel Core i7-1365U for the user application fingerprinting case study (cf. Section 5.2). Bins are constructed by grouping consecutive samples, for which we compute the mean value across the samples in the bin (referred to as "raw" in Figure 7), and the cumulative sum of the samples in the bin normalized by their mean (referred to as "cumsum" in Figure 7). We chose 300 samples per bin as a trade-off between accuracy and memory consumption, providing high accuracy while keeping the feature space manageable. We also evaluate the classifiers for each of the three feature modes: raw access counts only, cumulative sum only, and both channels concatenated. While the classification pipeline uses the MiniRocketMV transformer, we also evaluate the more powerful MultiRocketMV transformer. Each transformer is configured with 10 000 kernels. The MultiRocketMV transformer does not significantly improve accuracy over MiniRocketMV, but it does increase the required memory and computational resources. We conclude that MiniRocketMV performs best in concatenated-feature mode, which we use for all fingerprinting-based case studies in Section 5.

Appendix E. Ethics Considerations

Affected Stakeholders. Our research affects operating system developers, browser vendors, CPU vendors, cloud and virtualization providers, application developers, security researchers, and end users.

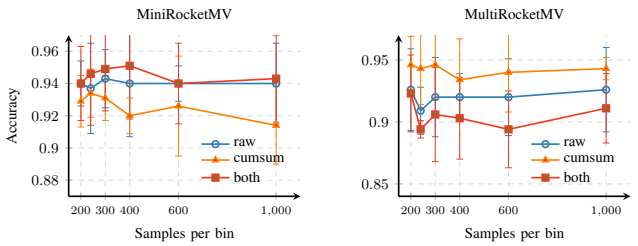

Figure 7: Classification accuracy of MiniRocketMV and MultiRocketMV across bin sizes and feature modes (raw access counts, cumulative sum, or both channels concatenated) on Intel Core i7-1365U. Both classifiers exceed 90 % across the full range of temporal resolutions, with error bars showing ±1 standard deviation over cross-validation folds.

Principles. All experiments were conducted in controlled environments on machines owned by the authors. The work focuses on observation and measurement of existing system behavior rather than exploitation of deployed systems, and it is accompanied by concrete mitigations.

Mitigations. We designed, implemented, and evaluated a kernel-level mitigation that eliminates zero-page-based side channels while preserving the semantics of zero-initialized memory and incurring minimal overhead. This mitigation directly addresses the root cause of the attacks rather than relying on partial or application-specific defenses. In addition, we discuss user-space and application-level mitigations to provide short-term and defense-in-depth options.

Decision to Publish and Proceed. We weighed the potential risk that our work could aid malicious actors against the benefits of exposing a previously underexplored attack surface. Zero-page deduplication is always enabled, widely deployed, and largely undocumented in threat models. Keeping the risk undisclosed would disproportionately benefit attackers who can independently discover and exploit it, while limiting the ability of defenders, system designers, and researchers to respond.

Appendix F. Meta-Review

The following meta-review was prepared by the program committee for the 2027 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

F.1. Summary

This paper introduces Zepade attacks, which exploit zero-page deduplication, to create side channels. Exploiting this side channel, an attacker can leak which entries are populated in linear-scan ORAM-style key-value stores, fingerprint applications, workloads, and websites, and establish covert channels across processes, containers, browsers, and virtual machines.

F.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field

F.3. Reasons for Acceptance

1) This paper identifies an impactful vulnerability. The paper reveals new attacks by exploiting zero-page deduplication, which has not been discussed in prior research.
2) This paper provides a valuable step forward in an established field. It is known that page deduplication can be exploited and lead to attacks; however, these attacks rely on non-zero pages, and exploits based on zero pages have not been studied.

F.4. Noteworthy Concerns

1) The real-world impact of the attack is unclear due to the lack of a strong end-to-end attack.
2) Fingerprinting case studies (Section 5.2 - Section 5.5) are conducted under a closed-world setting. It's unclear how the attack performs in an open-world setting.
3) Without evaluation and discussion of cache-line collisions, the broader claim that zero-page deduplication is relevant to hashtables and bitmaps is not well supported.
4) Missing important details including the threadmodel of UI redressing.

Appendix G. Response to the Meta-Review

This paper shows that zero-page deduplication creates a shared physical state that persists even when conventional page deduplication is disabled. We demonstrate several primitives that exploit this shared state, including ORAM leakage, application and website fingerprinting, and covert channels. Building an end-to-end attack is outside the scope of this paper and orthogonal to its contributions.

We performed an ad hoc experiment with an open-world scenario for our website fingerprinting case study (cf. Section 5.4). Here, we additionally record a single trace for each of 5000 websites in an unmonitored set. The classification accuracy on an Intel Core i7-1365U is $73.6\,\%$ with a standard deviation of $0.5\,\%$ and $86.8\,\%$ ($\pm$ $0.4\,\%$) on AMD Ryzen 7 7800X3D with the same experimental setup as in Section 5.4. Notably, in this scenario, $50\,\%$ of the recorded traces are classified as unmonitored. Nonetheless, the classification accuracy remains significantly above random guessing.

Our ORAM case study (cf. Section 5.1) shows that zero-page sharing can leak information in sparse data structures, e.g., ORAM-style key-value stores. Cache-line collisions, e.g., in bitmaps or hash maps, may leak coarser information, i.e., that an entry in the cache line was accessed. Still, in certain scenarios, this primitive enables exploitable leakage, as our ORAM case study demonstrates. We leave an evaluation of the impact of cache-line collisions to future work.