SafeFetch: Practical Double-Fetch Protection with Kernel-Fetch Caching

Victor Duta, Mitchel Josephus Aloserij, and Cristiano Giuffrida,
Vrije Universiteit Amsterdam

https://www.usenix.org/conference/usenixsecurity24/presentation/duta

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

SafeFetch: Practical Double-Fetch Protection with Kernel-Fetch Caching

Victor Duta   Mitchel Josephus Aloserij   Cristiano Giuffrida

Vrije Universiteit Amsterdam

Abstract

Double-fetch bugs (or vulnerabilities) stem from in-kernel system call execution fetching the same user data twice without proper data (re)sanitization, enabling TOCTTOU attacks and posing a major threat to operating systems security. Existing double-fetch protection systems rely on the MMU to trap on writes to syscall-accessed user pages and provide the kernel with a consistent snapshot of user memory. While this strategy can hinder attacks, it also introduces nontrivial runtime performance overhead due to the cost of trapping/remapping and the coarse (*page-granular*) write interposition mechanism.

In this paper, we propose `SafeFetch`, a practical solution to protect the kernel from double-fetch bugs. The key intuition is that most system calls fetch small amounts of user data (if at all), hence *caching* this data in the kernel can be done at a small performance cost. To this end, `SafeFetch` creates per-syscall caches to persist fetched user data and replay them when they are fetched again within the same syscall. This strategy neutralizes all double-fetch bugs, while eliminating trapping/remapping overheads and relying on efficient *byte-granular* interposition. Our Linux prototype evaluation shows `SafeFetch` can provide comprehensive protection with low performance overheads (e.g., 4.4% geomean on LMBench), significantly outperforming state-of-the-art solutions.

1 Introduction

The operating system (OS) kernel is the bedrock of modern systems. To provide service, the kernel includes a syscall interface, an explicit boundary between *untrusted* OS processes and the *trusted* kernel. Hence, it is crucial for the kernel to properly sanitize data that flows through this boundary (e.g., syscall arguments). Failure to do so may lead to (kernel) *double-fetch bugs* [15]. Such bugs occur when the kernel fetches (i.e., reads) the same/overlapping data from user space twice—a common kernel design pattern—without properly (re)sanitizing data on the second fetch. In essence, double-fetch bugs introduce a race condition, which attackers can exploit to mount *time-of-check to time-of-use* (TOCTTOU) attacks—changing user data between the two fetches. This is to bypass sanity checks and typically escalate privileges. Such bugs are both common (as they involve skipping seemingly "redundant" kernel checks [15]) and elusive (as they normally escape testing with production sanitizers [9]).

Prior research [18, 23, 26, 31] has mostly sought to detect and report several double-fetch bugs [2–7]. However, prior detection tools are imprecise and thus unsuitable to mitigate double-fetch bugs in production. More recently, Midas [15] proposed the first mitigation to offer protection (rather than detection) guarantees against double-fetch bugs. Midas relies on the MMU and `copy-on-write` mechanics to expose consistent snapshots of user pages to each syscall. To this end, Midas traps writes to each fetched user page, copies the page, and exposes the new (old) page to the writer (syscall). While this approach structurally prevents double-fetch exploitation, it also incurs nontrivial overhead due to the cost of trapping, remapping, and copying pages as well as operating at the coarse page granularity (causing overtrapping and overcopying due to *false sharing* [15]). Finally, due to the complex and costly operations Midas whitelists a number of syscalls (including the kernel-fetch heavy `execve`), ultimately reducing protection coverage.

In this paper, we propose `SafeFetch`, a practical protection system against kernel double-fetch bugs. The key idea is to move the core instrumentation from writes to kernel fetches, with the kernel maintaining per-syscall *caches* to serve kernel fetches. Indeed, our design has been inspired by Linux kernel developers seeking a practical double-fetch bug mitigation by "*performing some kind of kernel-side caching of user space memory*" [8]. `SafeFetch`'s design prevents concurrent (attacker-controlled) writes from corrupting data exposed to kernel double fetches—which instead hit the previously populated kernel-fetch cache by construction. As we will show, the vast majority of syscalls only copy a few bytes from user space, hence caching kernel-fetch data per-syscall can be done in a *simple* and *inexpensive* way. In contrast to Midas' *write-side* instrumentation strategy, SafeFetch's *fetch-side* strategy eliminates the need for costly MMU-based instrumentation (as writes run uninstrumented), false sharing and overcopying (as the cache operates at the byte rather than page granularity), and syscall-based whitelisting (as individual problematic kernel fetches can be whitelisted as needed). As a result, SafeFetch significantly improves both the performance and the security (protection coverage) of state-of-the-art solutions at a fraction of the complexity.

To support our claims, we implemented SafeFetch on Linux and evaluated our prototype, along with a number of caching-centric optimizations, against a number of standard benchmarks. Our evaluation shows that SafeFetch provides comprehensive protection at a fraction of the overhead incurred by Midas, despite the higher protection coverage (i.e., a single kernel fetch vs. three major syscalls whitelisted). For instance, SafeFetch incurs geomean performance overheads consistently below 5% across standard kernel benchmarks (i.e., LMBench, OSBench, and Phoronix). On the same benchmarks, Midas reports much higher geomean overheads (i.e., as high as ≈15% on OSBench and ≈36% on LMBench). Moreover, Midas incurs a single-benchmark worst-case overhead of 279% (vs. 22% for SafeFetch).

Contributions. We make the following contributions:

- We investigate common kernel fetch patterns during syscall execution and use the resulting insights to design a per-syscall kernel-fetch cache.
- We present SafeFetch, an implementation of our design to structurally mitigate *double-fetch* bugs in the Linux kernel. We show SafeFetch can be seamlessly integrated into existing kernel code paths, resulting in a practical implementation.
- We evaluate SafeFetch on a number of standard benchmarks, confirming that it can comprehensively mitigate double-fetch bugs with low performance overheads (e.g., 4.4% geomean on LMBench).

2 Background

2.1 User/kernel Memory Isolation

Modern operating systems rely on virtual memory support to enforce user/kernel memory isolation, that is preventing user (kernel) execution from accessing kernel (user) memory. This is typically done by using a joint virtual memory address space—where both user and kernel memory mappings coexist during user/kernel execution—and features offered by modern memory management units (MMUs) to enforce isolation. Specifically, on x86 platforms, the kernel can set (unset) the User/Supervisor bit in the Page Table Entries (PTEs) for user (kernel) memory mappings. This prevents user execution from accessing kernel memory. It also prevents the reverse (i.e., kernel execution accessing user memory) assuming Supervisor Mode Access/Execution Prevention features (SMAP and SMEP, respectively) are enabled.

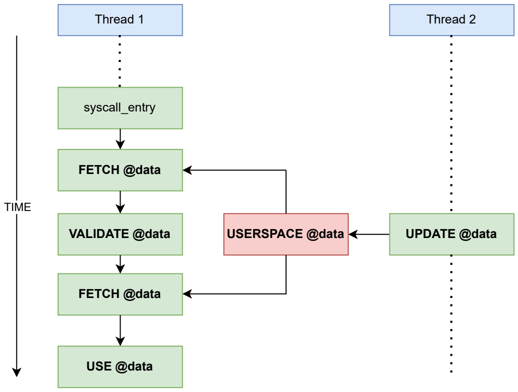


Figure 1: Workflow of a double-fetch exploit.

While important for memory isolation, SMAP complicates the implementation of common operations such as *kernel fetches*, that is user-to-kernel data transfers often issued by the kernel as part of syscall handling (e.g., copying a message from user memory to be sent over the network). To ease their implementation, modern operating systems typically support special kernel *transfer functions* in order to safely transfer data between user and kernel. For example, the Linux kernel offers two user-to-kernel (i.e., copy_from_user and get_user) and two kernel-to-user (i.e., copy_to_user and put_user) transfer functions, which temporarily disable SMAP and copy data from/to user memory (respectively).

2.2 Double-fetch Bugs

A kernel *double fetch* occurs in presence of kernel fetches transferring the same user data twice, that is with multiple user-to-kernel transfer function invocations for the same (or overlapping) user data on Linux. This pattern is normally benign and used to simplify or optimize common types of (e.g., deep or variable-length [26]) user-to-kernel data transfers. However, if the kernel assumes the data to be invariant and only validates data on the first fetch, the second fetch originates a *double-fetch bug*. Such bugs are particularly insidious as they introduce a race condition that may never cause any harm during normal execution. However, an attacker can exploit such

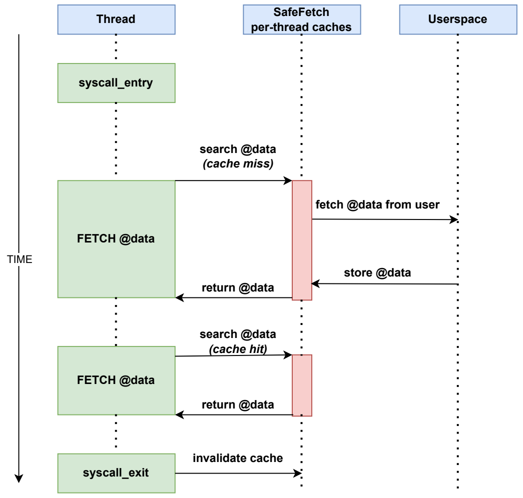


Figure 2: SafeFetch hindering a double-fetch exploit.

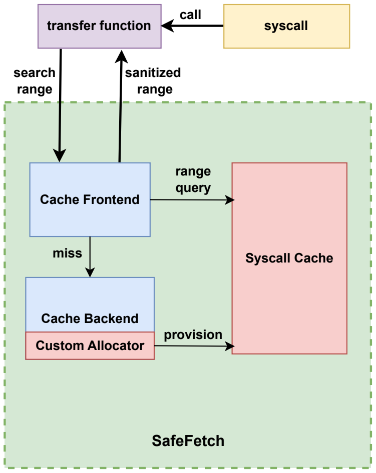


Figure 3: SafeFetch's high-level architecture.

bugs by racing against kernel execution from another user thread and corrupting the (unsanitized) data exposed to the second fetch. Such *time-of-check to time-of-use* (TOCTTOU) attack can bypass sanity checks and often kickstart a privilege escalation exploit [26].

Figure 1 depicts the workflow of a typical double-fetch exploit. In response to *Thread 1* executing a syscall, the kernel first fetches and validates some user **@data**. Shortly after, another attacker-controlled *Thread 2* concurrently modifies the **@data** in user space. In *Thread 1*, the kernel then proceeds to fetch **@data** again without (re)validation. This allows the attacker to bypass validation checks and mount a TOCTTOU attack. Prior work has largely focused on *detecting* such bugs with reasonable accuracy [18, 23, 26, 31]. In this paper, we instead focus on *protecting* the kernel from zero-day double-fetch bugs, while proposing a much simpler and more efficient design than the state-of-the-art protection system [15].

3 Threat Model

We assume a typical local exploitation threat model, with an unprivileged user-space attacker seeking to exploit a kernel double-fetch bug. Other classes of vulnerabilities are out of scope, e.g., addressed by orthogonal mitigations. The attacker ultimately aims to mount a TOCTTOU attack for a variety of different purposes, e.g., privilege escalation, info leak, denial of service, etc.

4 Overview

To hinder exploitation of kernel double-fetch bugs, SafeFetch guarantees that, during the lifetime of each syscall, kernel fetches to the same user data will return the same value. To this end, SafeFetch caches data read by kernel fetches at the per-thread and per-syscall granularity, as illustrated in Figure 2. As shown in the figure, when a syscall fetches some user **@data** for the first time, SafeFetch proxies the fetch to user memory and stores the fetched data in a per-thread in-kernel cache. When the syscall fetches the same **@data** again, SafeFetch retrieves the data from the cache. As such, double fetches are always consistently served with the initial version of the data regardless of any concurrent updates to user memory. At the end of the syscall lifecycle, the cache is invalidated to implement per-syscall caching semantics.

Internally, SafeFetch includes two core components, as shown in Figure 3. The **Cache Frontend** intercepts each kernel fetch, i.e., (user-to-kernel) transfer function invocation, and returns a *sanitized* range (i.e., a contiguous, immutable user memory block) in output. To this end, the frontend queries the current *syscall cache* for the range. In case of a hit, a (sub)range is served directly from the cache. In case of a miss, the frontend notifies the **Cache Backend**. The latter provisions the cache to store the missing (sub)range (fetched from user memory) by means of a *custom allocator*.

Challenges. While SafeFetch's design is conceptually simple, there are several challenges involved in its realization. First, the need for an in-kernel cache may lead to a nontrivial TCB impact, affecting security. We will later show it is feasible to efficiently implement our design with small TCB impact. Second, the need to interpose on all fetch operations may lead to protection coverage (and thus security) issues. We will later show it is feasible to produce a (nearly) full-coverage implementation on modern operating systems such as Linux, faring even better than the state of the art (Midas). Third, our fetch-side instrumentation strategy may end up copying more data than Midas' write-side strategy in case user data is never changed during syscall execution. We will later show such cost is marginal compared to that of MMU-based instrumentation, resulting in consistently better performance. Finally, our design require instrumenting the kernel's fast path (i.e., transfer functions). As such, its instrumentation and data structures need to be carefully designed to efficiently support typical kernel fetch patterns. We will show it is possible to capture a variety of different fetch patterns with relatively simple data structures. In the next sections, we first analyze the patterns relevant to our design. Then, we use the insights gathered from our analysis to detail our design.

5 Profiling Kernel Fetches

While our syscall caches are superficially similar to other kernel caches since they may support similar range queries, our design requirements are fairly unique. For instance, the VMA cache is a classic example of a kernel cache supporting range queries, however, its lookup patterns are wholly different from ours (lookups on locality-friendly memory management operations vs. lookups on kernel fetches) and so are its scope (process vs. syscall) and data storage requirements (fixed- vs. variable-sized data). As such, to make optimal design decisions, we need to learn more about typical patterns for kernel fetches, including their frequency, data transfer size, etc.

To this end, we developed a simple profiler to gather kernel-fetch statistics. Specifically, our profiler interposes on syscall execution and records the following statistics: the total number of ranges a syscall transfers from user space, the average size of ranges transferred by a syscall, and the total amount of data a syscall fetches from user space. Moreover, for each process executed during profiling, we also gather the number of syscalls that transfer data from user space. We use various benchmarks (e.g., LMBench, OSBench) and popular user applications (e.g., Nginx, Apache) to generate a workload to sample syscall execution. In total, our workload generated around 317 million syscall samples, exercising 165 individual syscalls (≈ 52% of all defined syscalls for Linux `x86_64`). Our main profiling results are depicted in Figures 4, 5, 6, 7, 8. We elaborate on the results in the next sections, using the gathered insights to motivate our design.

6 Cache Frontend

To provide the kernel with a consistent view of user memory, the cache frontend interposes on all user-to-kernel *transfer* function invocations requesting a specific user range. In response, the frontend queries the syscall cache for the range by means of the user (virtual) address and the length of the range. After the query completes, the frontend performs a query resolution step, fetching parts of the range from user space into the cache if needed (i.e., if not cached) and then forwarding a sanitized range to the original transfer function.

The frontend considers a user range A *sanitized* if and only if: for any sub-range B of contiguous user addresses, such that $B \subseteq A$, and B was previously fetched during the execution of the syscall (i.e., via a previous transfer function) then B must consist of the same bytes as when it was first fetched. When querying the cache, the frontend uses a predetermined *search policy* to locate the range in the cache. The search policy is subject to the (meta)data structure used to bookkeep the user ranges in the cache.

6.1 Efficient Range Queries

Given a user start and end address (i.e., a range) `SafeFetch` needs to find all cached chunks that overlap with this input range. Since a range may only partially overlap with an existing range (or multiple cached ranges), we are interested in finding the optimal data structure that can service this operation efficiently. For this purpose, other kernel subsystems use either linked lists (for small caches) or red-black trees (for large caches). The Virtual Memory Area (VMA) cache is case in point, generally serving address range queries via a per-process red-black tree. However, the VMA cache's fast path uses a linked list for the few recently used VMAs.

We experimented with both types of data structures in the context of `SafeFetch`. In both cases, a node contains *metadata* recording the start/end address of the range and a reference to the cached *data*. Clearly, in the case of many cached ranges, we expect linked-list-based queries to perform poorly, with a worst-case search complexity of O(n). In the same vein, we expect red-black trees to be more efficient, with a worst-case search search complexity of O(log(n)) due to constant-time rebalancing. Indeed, we experimentally verified that after around 100 cached ranges, the average search time of a linked list is slowed down by a factor of two compared to a red-black tree. However, when the cache contains only a few elements we observed the linked list significantly outperforming a red-black tree. On top of more lightweight search logic, a small linked list has another important performance edge over a small red-black tree on the insertion path (i.e., when adding a new user range into the cache). Indeed, due to rebalancing, insertions in the red-black tree are always around 10 orders of magnitude slower than in a linked list. Since according to our profiling results, double fetches are rare (occurring once every 1,277 sampled syscalls), insertions are frequent and are thus important to consider for performance. For more detailed double fetch statistics from our profiling results,

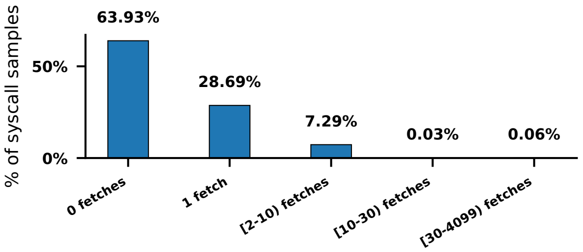


Figure 4: Percentage of syscall samples vs. number of ranges they fetch.

we refer the interested reader to Appendix A.

Selecting the ideal data structure. To select the ideal candidate between our two data structures, we turn to our profiling results. Figure 4 shows the percentage of syscall samples that fetch N ranges from user memory across all the profiled benchmarks. As shown in the figure, most samples ($\approx$ 64%) do not fetch user ranges at all, while the vast majority of those which do, fetch at most one range ($\approx$ 28.7% of samples). Even so, a nontrivial number of samples ($\approx$ 7.32%) fetch at most 30 ranges, while only $\approx$ 0.06% samples fetch over 30 ranges. As our results suggest, a (small) linked list seems the ideal candidate to support range queries for the vast majority of syscall samples. However, we also observed samples fetching as many as $\approx$ 4,000 ranges. In those cases, the linked list has very poor performance and the red-black tree is a vastly better option.

To optimize for all possible syscall scenarios, we ultimately opted for an `adaptive` search policy. In other words, the search policy uses a linked list by default until the number of ranges in the cache reaches a predetermined threshold. When the threshold is exceeded, `SafeFetch` switches to a red-black tree implementation. We detail how we experimentally selected a good threshold in Section 9.3. To convert the linked list into a red-black tree, `SafeFetch` uses an efficient algorithm that constructs a balanced red-black tree from an ordered linked list. This is done by first copying the pointers to ranges in a vector and then iterating over the vector in a binary breath-first search fashion. To keep the algorithm efficient, we need to initiate the conversion when the list contains a number of ranges of the form $2^N - 1$.

6.2 Query Resolution

When processing the result of the query, the **`Cache Frontend`** makes different decissions depending on whether it found a cached user range colliding with the queried range (**cache hit**) or not (**cache miss**).

In case of a miss, no subrange from the queried range was previously fetched from user space. As a resolution, the frontend fetches the range from user space and instructs the backend to allocate storage to insert the range into the cache. It then also indexes the new range by inserting new metadata into the linked list or red-black tree with with O(1) complexity—by piggybacking on the result of the previous query.

Cache hits can either be *perfect* or *partial*. A perfect hit means that the frontend found a cached range which contains the entire range it queried for. In this case, the frontend forwards the sane range from the cache without performing any fetch from user space. A cache hit is partial if the frontend finds a cached range that collides with the range it queried for, but does not contain the entire range. In this case, the queried range may collide with multiple cached ranges previously fetched from user space and the frontend needs to first execute a range defragmentation step to determine the sanitized range. Let B be the queried range and let $M = \{A_i | A_i \in cache \wedge A_i \bigcap B \neq \emptyset\}$ containing all cached ranges that collide with B. Defragmentation involves computing a new range C such that $C = \cup_{i=1}^{n} A_i \cup (B \setminus \cup_{i=1}^{n} A_i)$. In other words, the defragmented range C contains all bytes from the cached ranges colliding with B, while the sub-ranges of bytes that are not cached yet are fetched from user space. The frontend instructs the backend to replace all colliding ranges in the cache with the newly defragmented range C, after which it can service the sanitized range from C. Defragmentation piggybacks on the result of the previous search, because all colliding ranges can be found by using the previous search's iterator. To support this, insertion in the linked list and red-black tree preserves the virtual address ordering of cached ranges.

7 Cache Backend

The **`Cache Backend`** is responsible with managing the backing memory for the syscall cache. Specifically, its main goals are to efficiently manage the lifecycle of ranges in the cache and exploit CPU cache locality as much as possible. The latter can be achieved by stacking together ranges in memory for data locality and thus better CPU cache utilization, which speeds up range queries. The former involves enforcing policies for range allocation/deallocation and storage provisioning/relinquishing techniques to keep cache operations optimal.

To achieve its goals, the backend maintains one cache for each in-transit syscall (at the per-thread granularity) and adheres to a cache organization specifically tailored to maximize data locality when performing range queries through the cache. Additionally, the backend enforces a series of range lifecycle management policies to efficiently oversee cache lifespan. To support this overall strategy, the backend relies on a custom memory allocator.

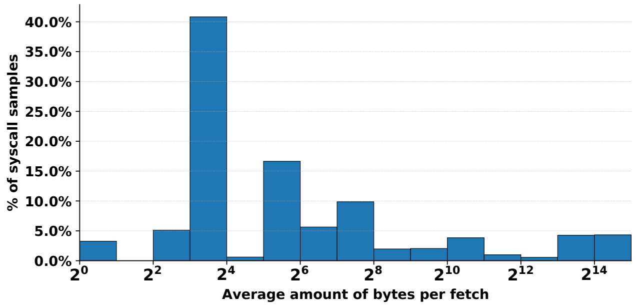


Figure 5: Percentage of syscall samples vs. average size of data they fetch.

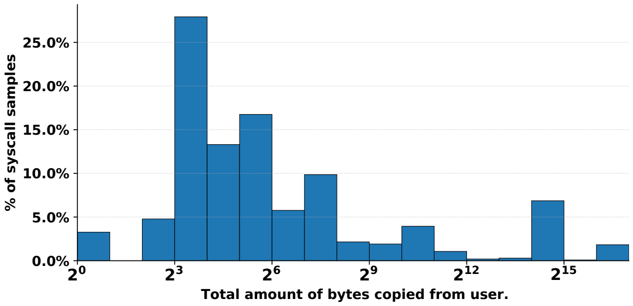


Figure 6: Percentage of syscall samples vs. total amount of data they fetch.

7.1 Custom Allocator

A naive allocation policy would be to create an object in the cache every time a syscall fetches a new range, by using an of-the-shelf kernel allocator (e.g., slab) to accommodate range data and metadata. However, standard kernel allocators do not give control over where objects are placed in (virtual) memory, so we cannot assure locality to improve the performance of range queries. Moreover, for an off-the-shelf allocator the allocation and deallocation logic might cause non-trivial overhead when syscalls transfer many ranges from user space, which happens in practice (see Figure 4).

A better approach is to service kernel memory in larger chunks to fit multiple ranges, i.e., using a *region-based allocation* scheme [13]. In such a scheme, a *region* consists of one or more *buffers* (contiguous memory blocks) with the same data lifetime. In a region, memory objects are allocated one after another in memory (in the current buffer) and get deallocated all at once (by flushing all the buffers), once the lifetime of all the objects in the region ends. As such, region-based allocation allows us to pack ranges together in memory to improve locality. Moreover, it reduces the number of calls to the underlying allocator when allocating ranges. On the fast path, an allocation involves only bumping a pointer to the next slot in the current buffer. Finally, it can efficiently deallocate all the allocated objects in one blow.

SafeFetch uses a custom region-based allocator, which services kernel memory at the granularity of a region, every time the backend requests more storage to hold ranges. To understand how well SafeFetch can benefit from region allocation's locality-friendly design, we turn again to our profiling results. Figure 5 shows the percentage of syscall samples that fetch an average size S of user data across all the profiled benchmarks. As shown in the figure, the majority (i.e., 65%) of syscall samples fetch ranges that are on average less than 64 bytes, confirming a high degree of data locality in a region in the common case. SafeFetch also benefits from the fast deallocation path of region-based allocation, efficiently deallocating all the ranges in the region when the syscall terminates. As an optimization, SafeFetch does not discard the buffers once the region is deallocated, but adds them to a pool for (fast) reuse in new regions created by future syscalls.

The next question is the buffer size one should use. As Figure 5 suggests, small allocation requests are common suggesting small buffers are desirable. Moreover, Figure 6 shows the percentage of syscall samples that fetch a total amount of B bytes of user data across all the profiled benchmarks. As shown in the figure, although a moderate portion of syscalls transfer much data (even above 8 pages), the vast majority transfer far less than a page of data. As a result, SafeFetch uses a single 1-page buffer by default in each new region (syscall) and elastically adds buffers as needed in the edge cases—many fetches per syscall or fetches transferring over 4 KB of data.

7.2 Dual Region Design

A naive approach would be to store the user memory ranges and the metadata necessary for range queries as a standalone object in one single per-syscall region. However, this approach would lead to metadata fragmentation and poor locality because metadata would be intermixed with data bytes. While most fetched ranges are small, we saw that syscalls can transfer larger ranges as well (Figure 5). To maximize data locality for range querying, the Cache Backend partitions the syscall cache in two separate regions: a *data* region stores all byte ranges copied from user space while a *metadata* region stores the bear bone necessities to perform queries over ranges. Consequently, for each user range, the backend maintains two objects: a) a *data* object storing the range of bytes copied from user space and b) a *metadata* object storing the properties of the range used when querying (e.g., user virtual address, length, pointer to the data object, and

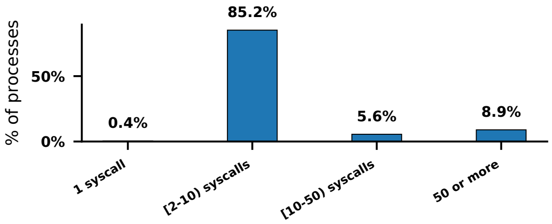


Figure 7: Percentage of processes vs. number of syscalls fetching user data.

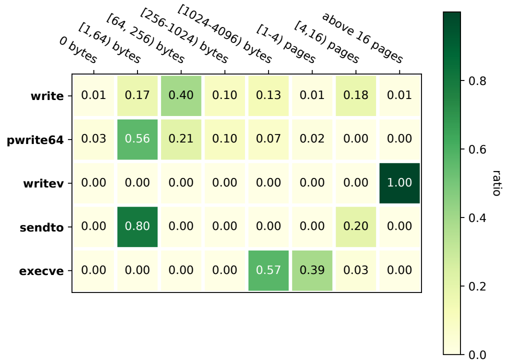


Figure 8: Heatmap showing the total amount of user data fetched by fetch-heavy syscalls.

a field used to link into a red-black tree or linked-list).

The size of *metadata* objects is fixed and small, allowing one to provision metadata regions with buffers smaller than a page. Nonetheless, we chose to serve 1-page buffers to metadata regions as well because this leads to better CPU cache coloring (and utilization) [19]. Despite the same default buffer size, the two regions provision their buffers from separate memory pools (i.e., slab caches) to further improve locality.

7.3 Lifecycle Management

Range allocation policy. A range is allocated in the cache every time the frontend encounters a cache miss while performing a range query. Our profiling data (e.g., Figure 4) suggests that most fetches are not double fetches, hence we expect frequent range allocations in the cache. Our custom allocator helps reduce the pressure on the (less efficient) underlying allocator, because many ranges can be allotted from the same region buffer.

Allocating ranges entails creating metadata and data objects in the appropriate regions. To this end, the backend uses a `region accounting` structure, which keeps track of all buffers allotted to the referent region. To speedup object creation, the `region accounting` structure stores a pointer to a region head buffer and favors servicing allocations from this buffer. As region heads get depleted at some point, new buffers are added to the region when appropriate and region heads get updated.

If an object cannot be created from the region head, the `Cache Backend` goes on the slow path iterating over all buffers allotted to the region until it finds one with enough space to service the request. As shown in Figure 6, some syscalls can transfer large amounts of user data, thus it is possible that a region can hold many depleted buffers. While this is more likely for *data* regions, it can also occur for *metadata* regions when syscalls execute many fetches. To optimize the slow path, the `region accounting` structure keeps a `freelist` containing only the buffers that are not yet depleted and can still service allocations. Lastly, if an object cannot be serviced on the slow path, then the backend leverages the custom allocator to create a new buffer in the referent region.

Cache invalidation policy. When a system call terminates, all ranges currently held in the cache can be deallocated. As a result, one could relinquish all storage held by cached ranges on syscall exit. This strategy reduces the memory footprint and also yields a simple deallocation path flushing all the buffers held by metadata and data regions. Moreover, as discussed, syscalls do not fetch many ranges, thus on average we need not free many buffers. However, as shown in Figure 7, our profiling data shows that 99.6% of the processes that incur fetches do so across at least two syscalls. In other words, if a process issues a syscall that fetches user data, it is likely to issue other syscalls that do the same.

Given our profiling results, it may be tempting to preserve all the allocated region buffers across syscalls. However, while this strategy minimizes the number of calls to the underlying allocator (improving performance), it may also significantly increase the steady-state memory footprint. That is particularly the case for threads issuing syscalls that fetch a lot of user data (even more than 8 pages, as shown in Figure 6). Given these observations, our cache invalidation policy is to preserve the head buffers, for both metadata and data regions, across the syscalls of a given thread but release all the other buffers held by each region, on syscall exit. Additionally, on each syscall exit, we invalidate the bookkeeping structure used by our frontend when performing range queries. Finally, on thread exit we release the residual head buffers.

Cache initialization policy. Given that our two head buffers persist across syscalls of the same thread, the next question is when to allocate such buffers. An option would be to simply allocate the head buffers at thread (i.e., `task_struct`) creation time. However, as shown in Figure 4, syscalls rarely fetch user data, e.g., ≈ 64% of the syscalls do not issue any fetches. Hence, eager head buffer allocation at thread creation time may lead to unnecessary memory overhead. As a result, the backend allocates the two head buffers (and the two regions) *lazily*, on the first user fetch that a thread incurs.

Zero-copy optimization. Given that most syscalls fetch little data, storing data into the cache is generally inexpensive. However, some syscalls are fetch-heavy and copy large amounts of data. In such scenarios, copying data into the cache incurs a nontrivial cost (as shown later in our evaluation), so it is desirable to eliminate as many such extra copies as possible. To investigate optimization avenues, we isolated all syscalls in our profiling results that copy more than a page of data from user space. Figure 8 presents our results in a heatmap.

As shown in the figure, many of these syscalls (with the exception of exec) are I/O bound. To avoid resource contention, I/O bound syscalls rely on the kernel `iov_iterator` functionality to copy dozens of large chunks of user data into kernel staging buffers. Such buffers (and their copied data) persist unchanged until the hardware (e.g., network card, hard disk) becomes available to complete the I/O transfer. To optimize out expensive copies to the cache, we use the kernel staging buffer itself as a data cache for all the `iov_iterator` copies fetching one or more pages. To delay deallocation of such staging buffers until syscall exit, we increase the reference count on the underlying pages. When enabling this *zero-copy* optimization (`SafeFetch` default), our cache invalidation policy also releases the staging buffers on syscall exit.

8 Implementation

We implemented a `SafeFetch` prototype for Linux kernel v5.11 (matching Midas' version for a fair comparison). The **`Cache Frontend`** adds the logic for cache lookups and fetch sanitization by instrumenting all kernel transfer function variants (i.e., `raw_copy_from_user` and all the `get_user` macro variants).

To manage the cache lifecycle for in-transit syscalls, the **`Cache Backend`** stores the accounting structures for (metadata and data) caches as structures in a thread's `task_struct`. Similarly, the bookkeeping structure used (for cache lookups) by the frontend is also referenced by the thread's `task_struct`. The custom region allocator uses two global slab caches (`kmem_caches`) to service region buffers efficiently to in-transit syscalls. We implemented cache invalidation by instrumenting the epilogue of the `do_syscall_64` kernel function and the `do_exit` function. Additionally, we instrumented all `clone` variants to initialize the `SafeFetch` logic in newly created threads. Finally, we implemented the zero-copy optimization by instrumenting the `copyin` kernel function used for `iov_iterator`-based user-to-kernel transfers.

Maintainability. While our design and implementation are optimized based on our syscall profiling data, we efficiently support a variety of syscall patterns. As such, even assuming applications change some of their syscall patterns in the future, we expect only minor (if any) modifications to our implementation. To facilitate the process, SafeFetch already supports runtime changes for all its core parameters, similar to other workload-sensitive kernel features (e.g., KSM, THP, etc.). In other words, with its ability to support many syscall patterns, its small codebase, and compliance to standard kernel design patterns (e.g., use of standard kernel data structures, standard hooking points, etc.), we expect `SafeFetch` to be highly maintainable moving forward.

9 Evaluation

In this section, we experimentally evaluate the security and performance of `SafeFetch`.

9.1 Security

CVE analysis. As `SafeFetch` hinders double-fetch bugs by construction, similar to Midas [15], we verify it can correctly mitigate a known double-fetch vulnerability. `CVE-2016-6516` is a known double-fetch vulnerability introduced in Linux kernel version 4.5. This vulnerability can be triggered via an ioctl syscall in combination with the `FIDEDUPERANGE` flag. This particular syscall attempts to deduplicate the memory pages between the source and its respective destination files. The source and destinations are supplied as parameters in a `file_dedupe_range` structure which the syscall copies from user space. However, before the syscall can fetch the structure it first needs to compute its size by using a count variable located inside the user space structure. After acquiring the count from user space, and performing sanity checks, the syscall fetches the entire structure into kernel memory overwriting the count field in the process, as can be seen in Listing 1. A double fetch vulnerability can emerge if between the first fetch (line **2**) and second fetch (line **17**) the count field is modified in user space. As a result, `vfs_dedupe_file_range` would operate on a malicious count value. This vulnerability was patched within Linux kernel v4.7 by copying back the old value into the data structure (line **23**) after the second fetch.

To evaluate whether `SafeFetch` can defend against this particular double fetch we modified Linux kernel version 5.11 by adding an additional check that verifies if a double fetch has occurred (lines **19-20**) prior to the proposed fix. In order to reproduce the bug

```
// first fetch, fetch count from user space
if (get_user(count, &argp->dest_count)) {
 ret = -EFAULT;
 goto out;
}

// use count to compute size of second fetch
size = offsetof(struct file_dedupe_range
   __user, info[count]);

// sanity check
if (size > PAGE_SIZE) {
 ret = -ENOMEM;
 goto out;
}

// second fetch
same = memdup_user(argp, size);

if (same->dest_count != count)
 printk("Double fetch Occurred\n");

// patch fix, write first fetched count back
same->dest_count = count;

// use second fetch
ret = vfs_dedupe_file_range(file, same);
```

Listing 1: CVE-2016-6516 Double-fetch vulnerability in ioctl_file_dedupe_range

we used the publicly available exploit[1] which runs the FIDEDUPERANGE ioctl in one thread, while constantly modifying the dest_count field in another thread. After running this exploit for 11 iterations on the vulnerable kernel, each iteration exploiting the double-fetch vulnerability one million times, we observed the double-fetch bug at least 10 times per iteration. When we subsequently ran the same experiment on a vulnerable kernel that deployed the SafeFetch defense, we did not observe a single double-fetch bug. This shows that SafeFetch is capable of mitigating kernel double-fetch vulnerabilities.

Security guarantees. SafeFetch's ability to provide the expected security guarantees depends on the correctness of the implementation. To this end, we focus our analysis on TCB impact and protection coverage. As for the former, we note that, despite the carefully optimized implementation, our prototype has a small TCB impact. In total, our solution adds ≈1,750 LOC to the Linux kernel, ≈250 of which are for the region allocator (which the Linux kernel currently does not provide, but is on the list of kernel developers). Moreover, the TCB impact is comparable to that of Midas (1,100 LOC).

As for the latter, SafeFetch relies on hooking standard user-to-kernel transfer functions to interpose on kernel fetches, providing full protection as long as the kernel complies to standard interfaces. We found this to be (nearly) always the case for the core Linux kernel. The nonstandard **unsafe_get_user** wrapper—only used to implement **strnlen** semantics—is the only exception we could find. Nonetheless, in rare cases, some third-party drivers may bypass such interfaces, impacting protection coverage and also breaking standard features such as SMAP in the process [10]. This (minor) limitation is shared by Midas, which also needs to instrument standard user-to-kernel transfer functions. Furthermore, whitelisting some patterns may also reduce protection coverage. For instance, as also observed by Midas, futex requires whitelisting for correct behavior. Indeed, in SafeFetch we had to whitelist a single kernel fetch in futex (and no other syscall), yielding nearly full protection coverage. In contrast, Midas provides lower protection coverage by entirely whitelisting three major syscalls. We also noticed Midas' prototype does not protect fetches issued by interrupt handlers, while SafeFetch does.

9.2 Performance

Evaluation setup. To evaluate the performance of SafeFetch, we tested it on a server comprising of an i7-6700 CPU, 32 GB of RAM and a 120GB SSD, running on a Ubuntu 22.04 operating system. Our comparison baseline is a vanilla kernel version 5.11 on which we build the SafeFetch prototype. We evaluate SafeFetch's default configuration, which uses an adaptive strategy that switches from a linked list to a red-black tree after 63 ranges are cached (see later for details on how we experimentally derived the threshold). The region allocator services 1-page region buffers to per-thread caches (for both metadata and data regions). Furthermore, the **default** SafeFetch config uses the zero-copy optimization for iov_iterator transfers above 4 KB. To get an idea on how SafeFetch directly affects core kernel building blocks, we analyze its performance on two OS benchmarks, namely LMBench [21] and OSBench [11]. Furthermore, we evaluate SafeFetch against popular applications from the Phoronix [12] test suite, to see how the prototype performs under various real-world workloads. For each result presented, we evaluated the corresponding benchmark 11 times and then reported the median. To get an estimate of SafeFetch performance relative to current state-of-the-art defenses, we compare our prototype against Midas—also deployed on Linux kernel version 5.11. As Midas incorporates a syscall whitelist, we made sure to also perform the same experiments on a SafeFetch variant that uses exactly the same kind of whitelisting (**whitelist** configuration).

LMBench. is a popular solution for system benchmarking as it focuses attention on many core kernel subsystems and has widely been used to pinpoint perfor-

[1] https://github.com/wpengfei/CVE-2016-6516-exploit/tree/master/Scott%20Bauer

Table 1: LMBench latency results.

Benchmark	Baseline (μseconds)	SafeFetch (%)				Midas (%)	
		default		whitelist			
		ovr.	stddev	ovr.	stddev	ovr.	stddev
Simple syscall	0.32	0.8%	±(1.6%)	0.8%	±(1.3%)	6.8%	±(1.0%)
Simple read	0.43	1.4%	±(0.9%)	1.4%	±(1.2%)	5.4%	±(1.8%)
Simple write	0.4	-0.5%	±(1.8%)	-0.2%	±(1.3%)	4.5%	±(1.6%)
Simple stat	0.68	-1.0%	±(1.5%)	0.8%	±(1.9%)	1.7%	±(1.2%)
Simple fstat	0.49	-1.7%	±(1.7%)	-0.3%	±(1.4%)	4.3%	±(1.4%)
Simple open/close	1.36	3.4%	±(0.6%)	2.0%	±(0.5%)	3.1%	±(0.7%)
Select on 10 fd's	0.46	15.1%	±(2.1%)	15.6%	±(1.0%)	242.2%	±(2.0%)
Select on 100 fd's	1.02	6.4%	±(1.1%)	6.8%	±(1.9%)	114.3%	±(1.4%)
Select on 250 fd's	1.94	4.8%	±(1.1%)	4.4%	±(1.6%)	59.9%	±(1.8%)
Select on 500 fd's	3.5	2.7%	±(1.7%)	2.9%	±(0.6%)	35.4%	±(1.2%)
Select on 10 tcp fd's	0.53	13.4%	±(1.8%)	13.8%	±(1.7%)	211.4%	±(4.6%)
Select on 100 tcp fd's	4.12	5.6%	±(1.2%)	7.7%	±(0.7%)	32.0%	±(2.1%)
Select on 250 tcp fd's	10.11	4.8%	±(1.0%)	6.0%	±(8.1%)	16.3%	±(2.4%)
Select on 500 tcp fd's	20.09	5.4%	±(1.3%)	6.1%	±(0.8%)	11.1%	±(0.9%)
Sig. handler install	0.37	9.8%	±(1.8%)	9.2%	±(1.7%)	278.9%	±(5.3%)
Sig. handler overhead	0.92	22.2%	±(1.9%)	22.9%	±(2.2%)	103.7%	±(29.8%)
Protection fault	0.63	-0.8%	±(5.1%)	-3.1%	±(5.5%)	2.6%	±(44.0%)
Pipe latency	3.45	3.4%	±(1.0%)	1.8%	±(1.1%)	3.3%	±(1.2%)
AF_UNIX latency	5.18	2.4%	±(1.3%)	1.8%	±(1.1%)	1.2%	±(1.2%)
Process fork+exit	72.36	0.4%	±(3.2%)	-0.1%	±(1.8%)	11.1%	±(4.2%)
Process fork+execve	243.43	-1.1%	±(1.7%)	-0.2%	±(2.8%)	4.4%	±(0.9%)
Process fork+/bin/sh	563.78	5.1%	±(2.0%)	0.5%	±(1.8%)	8.8%	±(2.8%)
TCP latency	9.01	1.9%	±(1.2%)	2.0%	±(1.1%)	5.6%	±(1.7%)
UDP latency	6.91	4.4%	±(4.7%)	6.8%	±(3.1%)	47.4%	±(4.7%)
TCP conn. latency	12.96	5.1%	±(14.6%)	-0.5%	±(1.6%)	20.1%	±(5.0%)
Geo mean.	-	4.4%	-	4.2%	-	35.9%	-

Table 2: LMBench bandwidth results.

Benchmark	Baseline (GB/s)	SafeFetch (%)				Midas (%)	
		default		whitelist			
		ovr.	stddev	ovr.	stddev	ovr.	stddev
read	9.59	-4.3%	±(3.0%)	-5.0%	±(3.2%)	4.1%	±(3.0%)
read open2close	9.57	-4.1%	±(2.9%)	0.1%	±(2.7%)	3.1%	±(2.5%)
Mmap read	16.71	1.0%	±(1.1%)	-0.2%	±(1.1%)	0.5%	±(0.8%)
Mmap read open2close	11.35	0.4%	±(0.9%)	-0.3%	±(0.9%)	8.8%	±(0.7%)
libc bcopy unaligned	14.94	0.5%	±(1.0%)	0.3%	±(0.9%)	-0.2%	±(0.8%)
libc bcopy aligned	14.97	0.6%	±(0.8%)	0.7%	±(1.3%)	0.4%	±(1.5%)
Mem. bzero	32.8	-0.1%	±(0.6%)	0.0%	±(0.5%)	-0.1%	±(0.5%)
unrolled bcopy unaligned	8.36	-0.8%	±(1.4%)	-1.8%	±(2.0%)	1.2%	±(1.7%)
unrolled partial bcopy unaligned	9.21	-0.3%	±(0.5%)	-0.3%	±(0.6%)	-0.1%	±(0.6%)
Mem. read	12.67	1.6%	±(9.6%)	-17.6%	±(8.5%)	1.3%	±(9.3%)
Mem. partial read	20.98	-1.9%	±(1.6%)	-2.4%	±(1.7%)	-0.8%	±(2.1%)
Mem. write	12.35	0.1%	±(0.5%)	-0.1%	±(0.4%)	0.1%	±(0.7%)
Mem. partial write	14.78	0.1%	±(0.2%)	-0.0%	±(0.2%)	0.2%	±(0.2%)
Mem. partial read/write	14.32	-0.3%	±(0.8%)	0.0%	±(0.6%)	0.1%	±(1.0%)
Socket bandwidth	8.73	30.3%	±(1.0%)	0.2%	±(0.8%)	1.0%	±(1.2%)
AF_UNIX	15.35	8.4%	±(1.4%)	0.5%	±(1.6%)	3.5%	±(1.7%)
Pipe	6.56	5.1%	±(2.0%)	1.4%	±(1.1%)	3.6%	±(1.1%)
File write	0.28	19.1%	±(4.0%)	0.8%	±(4.7%)	0.5%	±(4.4%)
Geo mean.	-	3.5%	-	-1.2%	-	1.5%	-

mance bottlenecks in OSes [16]. LMBench comprises of a series of latency benchmarks for a series of system calls that are commonly used by user applications and bandwidth benchmarks for relevant I/O related kernel facilities. In Table 1, we report the latency benchmark results of `SafeFetch-default`, `SafeFetch-whitelist`, and Midas relative to the baseline.

From the table we can observe that `SafeFetch-default` is able to provide comprehensive protection with marginal penalty to syscall latency for most syscalls tested by LMBench. The median geometric overhead across all latency benchmarks is around 4.4%. However, there are some syscall outliers that incur a larger penalty due to `SafeFetch`'s protection.

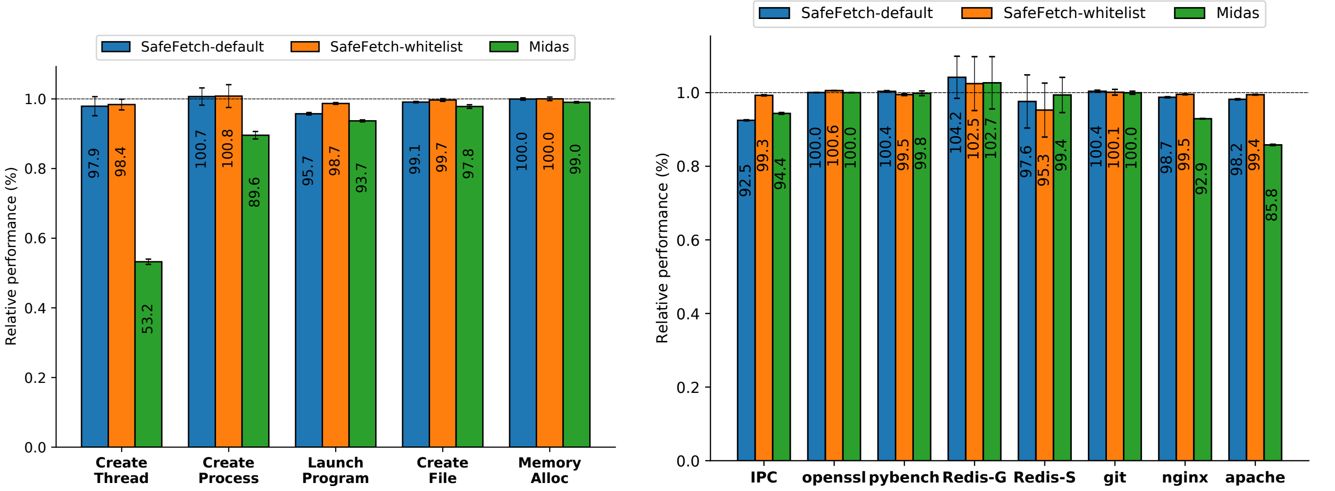
For example, the latency increase, when selecting over a small number of fds, is around 15% and 13%, on file and tcp descriptors, respectively. As the baseline latency is already small (less than half a microsecond), our defense wastes extra clock cycles copying the fd sets and `timespec` structure into `SafeFetch`'s cache from user space. Taking a deep dive in the main source of the overhead, we see that select benchmarks are mainly impacted by the extra downtime when allocating the head region buffers for metadata/data regions, rather than the actual copies into the cache.

Signal handling is also a major overhead outlier when using `SafeFetch-default`. Handling a signal is generally fast, executing with less than one microsecond downtime. However, `SafeFetch` adds extra overhead due to a series of user copies needed to setup and restore a `sig_return` frame, a stack frame used to restore the program context after calling the signal handler. The largest part of the overhead is spent on calls to memcpy when copying the signal frame into the data cache, as the frame can be around 160 bytes in size. At the same time, our results show that Midas reports significant overhead, incurring around 36% geomean overhead across all benchmarks. Signal handling and selecting on file descriptors is particularly slow and in some cases, Midas can triple the latency of these benchmarks.

We attribute our performance edge over Midas due to in-kernel caching being much more efficient than MMU-based instrumentation in the typical case of syscalls copying limited number of data (if any) from user space. Even in the case of the fork+shell microbenchmark, which can copy even a couple of hundred user space ranges, `SafeFetch` manages to outperform Midas in terms of latency, even though Midas whitelists the *execve* syscall.

The standard deviation relative to the baseline (the **`stddev`** column), suggests that our results are stable across measurements. One exception for `SafeFetch` is the TCP connection benchmark, which has moderate standard deviation due to one single (high) outlier. Lastly, we also noticed that applying whitelisting does not typically provide much performance gain for `SafeFetch`. However, for fetch-heavy benchmarks, such as fork+shell, whitelisting does completely eliminate the incurred overhead. This trend is even more noticeable in our LMBench bandwidth benchmark results, which we report in Table 2. As shown in the table, `SafeFetch-default`'s overhead on those benchmarks is again generally low (≈ 3.5% geometric mean overhead) and whitelisting completely eliminates the overhead for the outliers for `SafeFetch`—and to a lesser extent for Midas.

`OSBench.` We also evaluate `SafeFetch` on OSBench benchmarks. OSBench uses libc wrappers to evaluate basic kernel functions such as: process creation (e.g., `fork`, `waitpid`), thread creation (e.g., `pthread_create`, `pthread_join`), launching programs (e.g., `exec`), file creation (e.g., `fopen`, `fwrite`), and memory allocation (e.g., `malloc`, `free`). The results are shown in Figure 9, which reports the performance relative to the baseline



(a) OSBench

(b) Phoronix

Figure 9: SafeFetch performance on OSBench and Phoronix benchmarks relative to the baseline kernel. Standard deviations are reported relative to baseline performance.

kernel for SafeFetch-default, SafeFetch-whitelist, and Midas. Performance is expressed as the inverse of the time spent executing each benchmark.

These results show that creating threads and processes for the default SafeFetch configuration is marginal, i.e., ≈ 2% for thread creation and ≈ -1% for process creation. Interestingly, Midas does incur a significant performance loss on these benchmarks and some results significantly deviate from those reported in the original Midas paper (e.g., ≈ 88% vs. ≈ 5% slowdown for the "Create Thread" benchmark). We verified that the standard deviation is small for all the configurations (lower than 3% relative to the mean for SafeFetch and at most 2.7% for Midas), which confirms Midas' overheads are consistent in our environment. We attribute the deviations to Midas being particularly environment-sensitive. For instance, a trivial change in the allocator behavior may introduce/eliminate false sharing and significantly change Midas' performance characteristics. As detailed later, false sharing as well as other sources of overhead such as TLB shootdown are more likely to occur in a multi-threaded scenario. This is consistent with Midas' overhead being higher when creating threads.

Creating files is typically an I/O bound task and we expected SafeFetch-default to incur more overhead from the extra writes to the caches. However, the file creation benchmark creates small 32-byte files to prevent saturation of the underlying storage, thus imposing fairly low pressure on the caches. Across all benchmarks from OSBench, SafeFetch offers competitive performance with a negligible 1.3% geomean performance penalty, especially when compared to Midas's geomean performance impact of ≈ 15.4%. The overhead outlier for SafeFetch is the program launching benchmark (≈ 4.3%), as it relies on the fetch-heavy exec system call. Midas's performance penalty on OSBench is ≈ 15%, mostly because of thread creation. Looking at SafeFetch-whitelist performance, we observe that the geomean overhead on OSBench approaches 0%, as whitelisting completely eliminates the overhead on exec syscalls.

Phoronix Test Suite. The Phoronix Test Suite comprises of a wide range (> 600) of open-source benchmarks to evaluate application or OS performance under various workloads. We selected a range of benchmarks that evaluate popular user applications, each exhibiting various levels of kernel activity. From the chosen benchmarks, OpenSSL, is mainly compute bound and thereby its performance is mostly OS-agnostic. Therefore, we chose OpenSSL as a representative example for the best case scenario, assessing whether SafeFetch's OS instrumentation perturbs user space performance. The other benchmarks vary from single-threaded (e.g., pybench) to multi-threaded (apache) and multi-process (nginx) workloads. We also chose the IPC benchmark to highlight the worst-case scenario, as it spends most of the time in I/O syscalls (writev) to transfer multiple 128 byte chunks between processes, via TCP.

For benchmarks that report throughput, we compute the ratio relative to the baseline and, for benchmarks that report execution time (e.g., pybench, git), we compute the inverse ratio relative to the baseline. The speedup of each benchmark relative to the baseline for SafeFetch

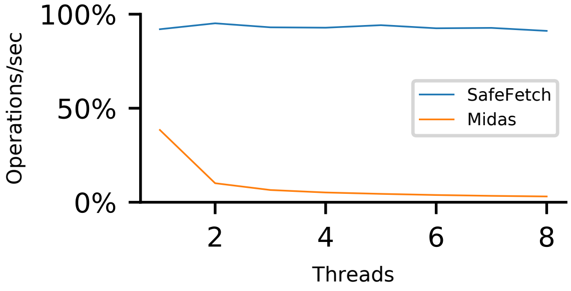


Figure 10: Relative throughput to the baseline when fetching from concurrent threads

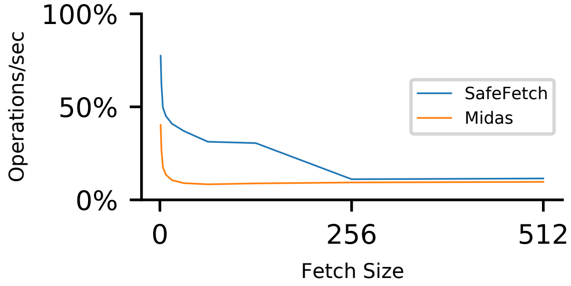


Figure 11: Relative throughput to the baseline when fetching user buffers of increasing sizes

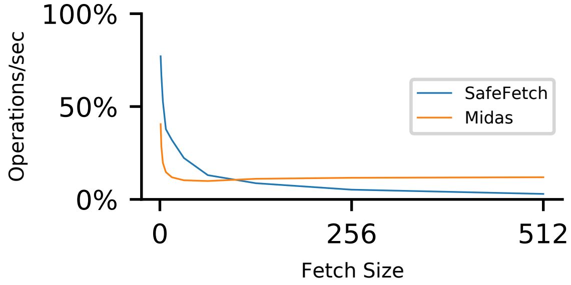


Figure 12: Relative throughput to the baseline when fetching user buffers one 1-page fetch at the time

(default and whitelist configs) and Midas is shown in Figure 9. On benchmarks that do not rely much on syscalls (e.g., pybench, openssl, git), `SafeFetch-default`'s performance is nearly equivalent to that of the baseline. Interestingly enough, benchmarking git revealed that some git commands can copy a couple of 1-page buffers from user space (during `write` syscalls). Even so, the performance of our default configuration is slightly faster than that of the baseline. This shows that caching is not necessarily a bottleneck even when storing large user ranges, provided that this does not happen frequently. Redis uses `recvfrom` and `sendto` syscalls when processing key store requests and responses. As Redis relies on pipelining to batch multiple client requests in a single response, the `sendto` syscall can cache large user space ranges (e.g., 8 pages). This would explain why `SafeFetch-default` scores 2.4% throughput degradation on *Set* benchmarks. Midas performs slightly better on this benchmark, having around ≈1% throughput degradation.

In contrast to the other benchmarks, the Apache and Nginx web servers heavily rely on I/O syscalls to respond to requests over the network and perform logging on the file system. Yet, `SafeFetch-default` scores only 1.3% and 1.8% throughput degradation on Nginx and Apache (respectively), while Midas scores 7.1% on Nginx and 14.2% overhead on Apache. As we will see in the next section, `SafeFetch`'s zero-copy optimization provides a large benefit on these benchmarks reducing the throughput degradation by nearly a factor of 5 compared to caching large `iov_iterator` copies. On the IPC benchmark, our default `SafeFetch` configuration scores 7.5% throughput degradation. Although the IPC benchmark exercises syscalls that can leverage the zero-copy optimization, these syscalls fetch buffers smaller than a page and thus never trigger the optimization. Whitelisting does not provide much benefit on Phoronix with the exception of the IPC benchmark, where whitelisting writev syscalls reduces the throughput degradation to 1%.

Overall, across all Phoronix benchmarks the default `SafeFetch` configuration has around 1.2% overhead and, when applying the same whitelisting as Midas, `SafeFetch` overhead again approaches the 0%. Conversely, Midas scores around 3.2% overhead (more than **2x** overhead vs. `SafeFetch`) even though it whitelists three major system calls. Across all the Phoronix benchmarks, the standard deviation is small, with the exception of Redis benchmarks which reported higher values (around 7% for both Midas and `SafeFetch`). The higher standard deviation matches that reported by others for multi-core systems, where Redis performance relies on the NUMA configuration of the system [1].

`Microbenchmarks.` To better understand the performance characteristics of `SafeFetch` and Midas, we devised microbenchmarks that repeatedly cause the kernel to fetch data from a user process. The goal is to evaluate throughput (number of fetches per second) as (i) the number of threads that fetch the same data increases or (ii) the size of the fetched data increases.

For the first scenario, our microbenchmark repeatedly issues a syscall to fetch a single 32-byte user buffer (the median fetch size in our profile) from multiple concurrent threads. We ran our microbenchmark for different number of threads (1 to 8, matching our core count).

Figure 10 reports the average per-thread throughput we measured relative to the baseline. As shown in the figure, SafeFetch's throughput is stable as we increase the thread count, with a 7.1% degradation on average. This is because kernel-side caching scales efficiently with no interferences across threads. In contrast, Midas' throughput progressively degrades as we increase the thread count, with a degradation as high as ≈97% with 8 concurrent threads. A number of factors contribute to the heavy performance penalty.

First, Midas' MMU-based instrumentation introduces nontrivial costs on the fast path: (i) software page table walks to retrieve and mark PTEs in protected state and (ii) TLB misses after transitioning pages from/to read-only state. Such costs are noticeable even in a single-threaded scenario, where Midas already reported a significant 62% throughput degradation. Second, transitioning pages to/from protected state adds extra (e.g., page table) locking to the fast path, leading to higher lock contention and lower performance as we increase the number of threads. Third, changing page protections in a multi-core setting introduces increasingly expensive TLB shootdowns as we increase the number of threads. Last, false sharing effects are generally more prominent as we increase thread count. To synthetically simulate these effects, we configured each thread in our microbenchmark to interleave fetches with writes to the same page as the fetched data. When rerunning our microbenchmark, we observed no noticeable throughput differences for SafeFetch. In contrast, Midas' throughput further dropped by ≈3% on average, due to false sharing inducing additional page fault and (over)copying overhead.

For the second scenario, our microbenchmark uses a single thread (best-case scenario for Midas) repeatedly issuing a syscall to fetch user data of a given size. We ran our microbenchmark for different sizes (1 to 512 pages) of a user buffer transferred by a single fetch. Figure 11 reports the average throughput we measured relative to the baseline. As expected, both SafeFetch and Midas experience significant throughput degradation as the user buffer grows larger and larger. However, SafeFetch consistently outperforms Midas and even by a wide margin up to 256-page buffers. This shows that the costs associated to page protection (Midas) are consistently higher than those of copying data (SafeFetch), especially when the buffer is in the CPU caches. To further stress our implementation, we also repeated our experiment when distributing the user buffer transfer over multiple fetches per syscall (1 page fetched at the time). Figure 12 presents our results. As shown in the figure, only the heavier pressure on SafeFetch's cache lookups at a large number of fetches (128 pages and beyond) gives Midas a performance edge. Nonetheless, it is rare for syscalls to copy such large amounts of data over many fetches. For instance, we only recorded 2,350 samples in our profile (out of ≈317 million syscall samples) that fit this pattern.

Table 3: SafeFetch-induced memory utilization.

Benchmark	SafeFetch raw memory utilization					
	cache provisioning			zero-copy pins		
	avg.	99^{th}	peak	avg.	99^{th}	peak
LMBench	**8.02 KB**	**8 KB**	**236 KB**	**0.65 KB**	**0 KB**	**9.6 MB**
OSBench	**8.04 KB**	**12 KB**	**12 KB**	**0 KB**	**0 KB**	**0 KB**
Phoronix	**8.04 KB**	**8 KB**	**44 KB**	**5.82 KB**	**16 KB**	**128 KB**

Memory utilization. We show per-syscall raw memory utilization in times of storage provisioned to caches and pages pinned by the zero-copy optimization across all benchmarks in Table 3. The table depicts average, peak, and 99^{th} percentile utilization, only for syscalls that use the SafeFetch defense at all. On average, syscalls use at most 3 pages for the cache storage and pin at most 2 pages of memory (e.g., Phoronix). While peak utilization is higher on LMBench and Phoronix the 99^{th} percentile results suggest that this trend is isolated to only a couple of executed system calls, for both storage provisioning and zero-copy pinning. On Phoronix, we observed that the calling thread's resident memory size is increased on average by ≈0.12% due to provisioning in-transit syscall caches. Overall, our results show that, even at (short-lived) peak utilization, SafeFetch only marginally impacts the system memory footprint, demonstrating realistic overheads for a practical solution.

9.3 Design Breakdown

In this section, we discuss the optimal configuration for SafeFetch's adaptive cache lookup algorithm as well as the performance benefits of SafeFetch's core design decisions (i.e., using a region allocator, preserving head region buffers across syscalls, and the zero-copy optimization for iov_iterator copies).

Adaptive search algorithm. SafeFetch uses an adaptive search algorithm which switches from a linked list to a red-black tree after a threshold of cached ranges is reached. We have determined the optimal conversion threshold experimentally. To this end, we performed repeated experiments (101 times), with each experiment performing 100 1-byte fetches from a set of random virtual addresses while using a given data structure (linked list or red-black tree). We ensured the addresses were nonoverlapping (no double fetches), thereby simulating the most common scenario we uncovered during profiling.

Figure 13 reports the average time (search+insert time) of each fetch as the number of cached ranges grows, for the linked list and red-black tree configurations. The

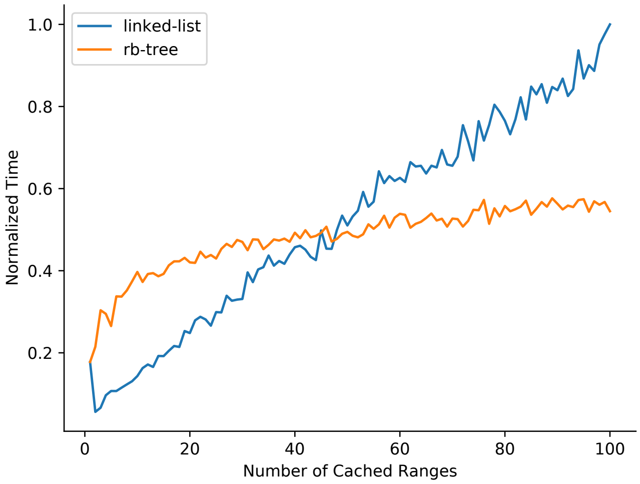


Figure 13: Average normalized time for each fetch vs. number of cached ranges.

average times are normalized with respect to the largest observed value. As shown in the figure, after ≈ 50 cached ranges the red-black tree is more efficient than the linked list. To establish the most efficient conversion threshold, we also need to consider constraints of the conversion algorithm. As mentioned earlier, our algorithm is most efficient if a dataset contains $2^N - 1$ entries, which, according to our results, suggests a threshold of 31 or 63.

To determine the optimal one, we compared average cumulative search+insert times for the first 64 fetches in our experiments with all possible conversion thresholds smaller than 63. Figure 14 reports our results, highlighting the cost of conversion with yellow bars. While the 31 and 63 configurations offer similar performance, SafeFetch uses the latter by default. This is to delay conversion as much as possible, which is beneficial for (more common) syscalls with a lower number of fetches.

Benefits of region allocation. We used LMBench to understand how region-based allocation speeds up syscall performance for SafeFetch. As a baseline, we use our default SafeFetch prototype, equipped with a region allocator which relies on 4 KB buffers for data/metadata provisioning. The baseline also preserves region heads across syscalls (executed from the same thread). We compare the baseline against a SafeFetch version invoking a standard kernel allocator (i.e., kmalloc) when a syscall needs to add ranges into the cache and freeing all the objects on syscall exit (the **noregion** configuration). Moreover, to determine the benefits of preserving buffers across syscalls, we also compare the baseline against a region-enabled SafeFetch version not preserving region heads across syscalls (the **noheads** configuration).

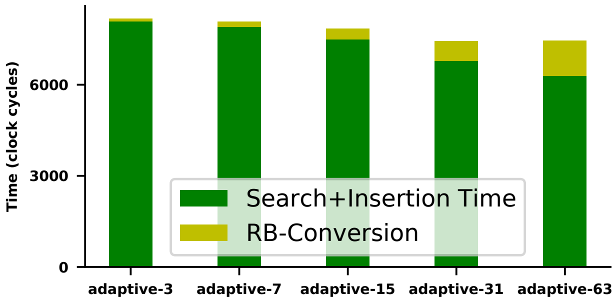


Figure 14: Cumulative times of the first 64 fetches for different adaptive configurations of SafeFetch. The conversion cost is marked on top of each bar.

For brevity, we focus on LMBench results. Across all the LMBench benchmarks, the **noheads** configuration adds 3.1% geomean overhead compared to the baseline, with the select benchmarks being the worst offender. Select on 500 file descriptors is 18% slower on the **noheads** configuration compared to the baseline. This is somewhat expected as the select syscall can be interrupted (e.g., by a signal), which may prompt the application logic to restart the syscall. The **noregion** configuration, in turn, adds a 5% geomean overhead compared to the baseline, showing that region allocation significantly speeds up syscall performance for SafeFetch. The delta is even more evident if we look closely at the worst-case overheads. Major overhead offenders for the **noregion** configuration are select on file descriptors (as high as 30% slowdown compared to the baseline), signal handling (≈ 21% slowdown), and exec calls (≈ 4.3% overhead when spawning a shell).

Zero-copy optimization. To understand the benefits of the zero-copy optimization, we evaluate two SafeFetch configurations: the default SafeFetch configuration (/w zero-copy) and the same configuration with the zero-copy optimization disabled (/wo zero-copy). We compare our two configurations against the baseline Linux kernel on a series of I/O benchmarks. To showcase the worst-case scenario, we used the AF_UNIX socket stream bandwidth and the pipe bandwidth benchmarks part of the LMBench suite. To show the gains of zero-copying on real-world applications, we used the Apache and Nginx web servers which are known to exercise I/O syscalls. Table 4 presents our results, with the SafeFetch-induced throughput degradation compared to the baseline.

As shown in the table, without the zero-copy optimization, the throughput degradation is significant. For instance, AF_UNIX throughput is reduced by ≈ 39% while pipe throughput is reduced by ≈ 16%. This is because,

Table 4: SafeFetch-induced throughput degradation.

Benchmark	SafeFetch **/wo zero-copy (%)**	 **/w zero-copy (%)**
AF_UNIX	**39.1%**	**8.4%**
Pipe	**16.1%**	**5.1%**
Nginx	**5.9%**	**1.3%**
Apache	**9.3%**	**1.8%**

without zero-copy optimizations, SafeFetch copies large chunks of user data to the data cache, incurring significant overhead. For instance, avoiding unnecessary copying reduces the throughput degradation to 8.4% and 5.1% on the AF_UNIX and pipe benchmarks, respectively. Copying large chunks of user data impacts performance of real-world applications as well: Nginx and Apache have their throughput reduced by 5.9% and 9.3% compared to the baseline (respectively). Enabling the zero-copy optimization reduces throughput degradation for both web servers to below 2%. Our results confirms the zero-copy optimization is key to good I/O benchmark performance.

10 Related Work

While double-fetch or TOCTTOU (Time-of-Check-to-Time-of-Use) vulnerabilities affect different interfaces and components (e.g., enclaves [17, 25], sandboxes [22], compilers [29], and compartments [14]), we focus here on closely related research on operating system kernels. Serna [24] was the first who coined the name "*double-fetch vulnerability*" to describe an instance in the Windows kernel. Since then, research has focused on finding double-fetch bugs (through static and dynamic program analysis) or even mitigating these issues.

Static analysis. On the static analysis front, Wang et al. [26] leverages pattern matching analysis on source code to find double-fetch bugs in the Linux kernel. DFTinker [20] extends such pattern-based approach to increase double-fetch coverage and reduce false positives. While pattern-matching techniques proved effective in uncovering new double-fetch bugs, they still produce a high rate of false positives and negatives and are fundamentally limited to detecting only specific bug patterns.

DEADLINE [31] and DFTracker [27] improve static detection of double fetches by means of compiler-level symbolic execution. While these approaches can be applied more broadly (e.g., to detect compiler-induced double fetches [28,30]) and are generally better suited for vetting false positives, symbolic execution introduces other limitations (e.g., path explosion) and does not completely remove false reporting (e.g., due to imprecise memory modeling or incomplete code coverage). DEADLINE, for example, does not detect double fetches in inline assembly, which is widely used in the kernel.

Dynamic analysis. On the dynamic analysis front, Jurczyk and Coldwind [18] propose Bochspwn, which instruments memory access callbacks in an x86 emulator to trace double-fetch patterns. Coupled with fuzzing, their technique found a series of exploitable double fetches in the Windows kernel. However, Bochspwn incurs high overheads because its tracing instrumentation affects the emulator's fast path. Wilhelm [28] uses a similar approach to Bochspwn and found double-fetch vulnerabilities in the Xen hypervisor, one of which was introduced through compiler optimizations. Schwarz et al. [23] use a fuzzer in conjunction with concurrent user memory accesses to detect double fetches through a cache covert channel. However, such a solution is reliant on hardware features (e.g., caches), which vary across microarchitectures and are subject to noise. While dynamic techniques are often more precise than static approaches, they are fundamentally subject to code coverage and limited to finding double-fetch bugs only on executed code paths.

Mitigations. Midas [15] is the first proposed mitigation to protect the operating system kernel against double-fetch vulnerabilities. Midas relies on MMU-enabled Copy-on-Write semantics to create snapshots of user pages accessed by the kernel during syscall execution. As a result, Midas incurs nontrivial runtime performance overhead due to the cost of trapping/remapping and the coarse (page-granular) write interposition mechanism. Additionally, Midas whitelists some syscalls, most notably the fetch-heavy *exec* syscall. In contrast, SafeFetch eliminates the need for MMU-based instrumentation and offers comprehensive protection at a fraction of Midas' cost. Moreover, our solution is simple and seamlessly integrates with existing kernel code paths. Indeed, SafeFetch's high-level design is inspired by the "*kernel-side caching of user space memory*" on the wishlist of Linux kernel developers to address double-fetch vulnerabilities [8].

11 Conclusion

We presented SafeFetch, a practical double-fetch bug protection system which caches kernel fetches at the syscall granularity and serves subsequent fetches of the same data from the cache, thus ensuring that user data never changes across fetches. We showed that SafeFetch offers comprehensive protection at a fraction of the cost of state-of-the-art solutions such as Midas with marginal memory overheads and geometric performance overheads consistently below 5% across various OS benchmarks (e.g., 4.4% on LMBench and 1.3% on OSBench) and real-world workloads (e.g., 1.2% on Phoronix).

Availability

To encourage adoption, we have open sourced SafeFetch at https://github.com/vusec/safefetch. We are also actively engaging Linux kernel developers to seek mainline inclusion.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported by Intel Corporation through the "Allocamelus" project, by NWO through project "INTERSECT" and "Theseus", and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] Best practices for adapting phoronix test suite to benchmark linux performance. https://blogs.oracle.com/linux/post/best-practices-for-adapting-phoronix-test-suite-to-benchmark-linux-performance.

[2] Cve-2013-1332. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1332.

[3] Cve-2015-8550. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8550.

[4] Cve-2016-10433. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10433.

[5] Cve-2016-10435. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10435.

[6] Cve-2016-10439. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10439.

[7] Cve-2016-8438. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8438.

[8] Detect and avoid ToCToU double-fetch / double-read from userspace. https://github.com/KSPP/linux/issues/95.

[9] KASAN. https://github.com/google/kasan/wiki.

[10] Ni linux device drivers fails to install daqmx. https://knowledge.ni.com/KnowledgeArticleDetails?id=kA03q000000wzMJCAY.

[11] OSBench Authors. OSBench. https://https://github.com/mbitsnbites/osbench.

[12] Phoronix Authors. Phoronix. https://www.phoronix-test-suite.com.

[13] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Reconsidering custom memory allocation. In *OOPSLA*, 2002.

[14] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andres Sanchez, Babak Falsafi, and Mathias Payer. Securecells: A secure compartmentalized architecture. In *IEEE S&P*, 2023.

[15] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic kernel TOCTTOU protection. In *USENIX Security*, 2022.

[16] Aaron B Brown and Margo I Seltzer. Operating system benchmarking in the wake of lmbench: A case study of the performance of netbsd on the intel x86 architecture. In *SIGMETRICS*, 1997.

[17] Felix Dreissig, Jonas Röckl, and Tilo Müller. Compiler-aided development of trusted enclaves with rust. In *ARES*, 2022.

[18] Mateusz Jurczyk and Gynvael Coldwind. Identifying and exploiting windows kernel race conditions via memory access patterns. 2013.

[19] Hsien-Hsin S Lee and Gary S Tyson. Region-based caching: an energy-delay efficient memory architecture for embedded processors. In *CASES*, 2000.

[20] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In *WASA*, 2018.

[21] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.

[22] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *USENIX Security*, 2020.

[23] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *AsiaCCS*, 2018.

[24] Fermin J. Serna. Ms08-061 : The case of the kernel mode double-fetch. https://msrc.microsoft.com/blog/2008/10/ms08-061-the-case-of-the-kernel-mode-double-fetch/, 2008.

[25] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *CCS*, 2019.

[26] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *USENIX Security*, 2017.

[27] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science*, 13, 2019.

[28] Felix Wilhelm. Xenpwn: Breaking paravirtualized devices. *Black Hat USA*, 2016.

[29] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: Bypassing cfi through compiler-introduced double-fetches. In *IEEE S&P*, 2023.

[30] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *USENIX Security*, 2023.

[31] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *IEEE S&P*, 2018.

A Double Fetch Statistics

In this section, we present detailed statistics related to fetch and double fetch occurrences across our benchmarks. Table 5 presents our results. For both generic fetches and double fetches (i.e., fetches that transfer data overlapping with a previous fetch within the same syscall) we detail: the rate of (double) fetches relative to the total number of syscall samples collected for the benchmark (the **Rate** column), the total number of distinct syscalls that executed a fetch during the benchmark (the **Syscalls** column). Additionally, we report the minimum, average, and maximum number of (double) fetches executed by one single syscall sample. The last row in the table refers to syscalls executed by background applications. As shown in the table, each benchmark has a nontrivial (e.g., up to 21 for Phoronix) number of syscall types with double fetches. Moreover, the number of double fetches performed by double-fetch syscalls greatly varies (e.g., ranging from 1 to 218 for Phoronix).

Table 5: Statistics for (double) fetch rates.

Benchmark	Statistic	Fetches	Double Fetches
LMBench	**Rate**	**1/2**	**1/273457**
	Syscalls	**38**	**7**
	Min/Avg/Max	**1/1/459**	**1/50/67**
OSBench	**Rate**	**1/3**	**1/80**
	Syscalls	**17**	**5**
	Min/Avg/Max	**1/6/134**	**1/42/43**
Phoronix	**Rate**	**1/4**	**1/5993**
	Syscalls	**47**	**21**
	Min/Avg/Max	**1/1/661**	**1/1/218**
Background	**Rate**	**1/2**	**1/644**
	Syscalls	**93**	**20**
	Min/Avg/Max	**1/2/4099**	**1/130/467**

Across all benchmarks, syscalls are likely to execute fetches (around 1 in 3 syscalls fetch user buffers), but more often they will fetch only a small number of user buffers (e.g., on average syscalls fetch 6 user buffers on OSBench). This prompted us to use a linked list as the default data structure for efficient caching. However, across all benchmarks, there are occurrences of fetch-heavy syscall executions (e.g., with up to 661 fetches on Phoronix), motivating the need for an adaptive strategy that resorts to a red-black tree to handle fetch-heavy scenarios. Moreover, the striking difference between fetch rates and double fetch rates across all benchmarks, with double fetches being far less frequent, suggests that cache insertions happen often and thus it is crucial to factor in insertion time when determining the optimal threshold to convert to a red-black tree.

Looking more closely at the variability of the number of fetches a syscall makes—across the syscalls that fetch data in our profile—we observed that ≈55% have a stable fetch rate (i.e., they fetch the same number of user buffers every time). Most syscalls with stable fetch rates perform either 1, 2 or 3 fetches, with the exception of sendmmsg which performs 6 fetches all the time. From the syscalls that have variability in the number of executed fetches, 45 system calls execute between 1 and at most 8 fetches while only 5 system calls execute more than 8 fetches. Special cases are 3 system calls, i.e., pwrite64, execve, and write that have high variability and execute 1-255, 1-1411 and 1-4,099 fetches, respectively. Again, this variability motivates the need for an adaptive strategy for cache lookups. Looking instead more closely at the variability of the number of double fetches—across the syscalls that fetch data twice in our profile—we observed that ≈60% execute only one double fetch, while 33% execute either 1 or 2 double fetches and the sendmsg syscall may execute between 3 and 6 double fetches. The execve syscall is again an exception and can execute between 1 and as many as 467 double fetches.