ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Midas: Systematic Kernel TOCTTOU Protection

Atri Bhattacharyya
EPFL

Uros Tesic [*]
Nvidia

Mathias Payer
EPFL

Abstract

Double-fetch bugs are a plague across all major operating system kernels. They occur when data is fetched twice across the user/kernel trust boundary while allowing concurrent modification. Such bugs enable an attacker to illegally access memory, cause denial of service, or to escalate privileges. So far, the only protection against double-fetch bugs is to detect and fix them. However, they remain incredibly hard to find. Similarly, they fundamentally prohibit efficient, kernel-based stateful system call filtering. Thus, we propose *Midas* to mitigate double-fetch bugs. Midas creates on-demand snapshots and copies of accessed data, enforcing our key invariant that throughout a system call's lifetime, every read to a userspace object will return the same value.

Midas shows no noticeable drop in performance when evaluated on compute-bound workloads. On system call heavy workloads, Midas incurs 0.2–14% performance overhead, while protecting the kernel against any TOCTTOU attacks. On average, Midas shows a 3.4% overhead on diverse workloads across two benchmark suites.

1 Introduction

The operating system (OS) kernel provides isolation between processes and is a key trusted computing base. Each *untrusted* userspace process runs under a dedicated user in its own address space and must request resources (such as communication channels or changes to its address space) from the *trusted* kernel. The userspace/kernel interface forms an explicit trust barrier; all data that crosses this boundary in either direction must be carefully checked by the kernel. Userspace processes attack the kernel by issuing system calls (syscalls) that then trigger kernel bugs, elevating the privileges of the process. A common class of kernel bugs are so-called *double-fetch* bugs [35, 38, 40, 44]. They occur when higher-privileged code, such as the kernel, reads the same data from the lower-privileged address space twice. Double-fetch bugs are a *race condition* between threads of different privileges. A *time-of-check to time-of-use (TOCTTOU)* violation occurs when the first read is used to check a condition while the second read is used to modify state. An example of a double fetch bug is when the kernel reads the length of a buffer from userspace, allocates a kernel buffer, then reads the length a second time to finally copy the data from userspace to the kernel. An attacker may concurrently overwrite the length of the buffer (with a larger number) after allocation, causing the memory copy to overflow the kernel buffer. Double-fetch bugs are a frequent problem in kernels and hypervisors [1–10]. Watson [42] blames an unfixable TOCTTOU constellation as a reason for the generic insecurity of *syscall wrappers*. Syscall filtering wrappers require that data read from userspace for the initial check remains the same when the kernel later uses it for computation. Therefore, such filters can currently only check arguments passed by value. Midas enables "deep argument inspection" for SecComp [12, 13] (i.e., checks arguments passed by reference). Without Midas, such inspection is impossible: these checks introduce double fetches, and consequently TOCTTOU bugs.

To mitigate double-fetch bugs in the kernel, a system must prohibit *concurrent changes*[1] to memory accessed by the syscall. Attackers may find crafty ways to trigger such concurrent writes, including: *i*) direct writes from userspace (e.g., from concurrent threads), *ii*) kernel writes from syscalls (e.g., from concurrent syscalls), *iii*) modifying address space mappings, *iv*) concurrent `write` syscalls to a file that alters mapped file pages, and *v*) storing arguments on device-backed pages, leveraging devices to trigger concurrent writes. To prevent attacks, all concurrent writes must be prohibited.

We base our defense on a single key invariant: ***through a syscall's lifetime, every read to a userspace object will return the same value***. From this invariant we derive a *security property* ensuring that every read during the execution of a syscall is tracked. Subsequent reads from the same address will always return the same value. For performance, multiple

[*]This work was done during the author's time at EPFL.

[1]The attacker model includes both concurrent and parallel writes.

versions of an object may exist simultaneously, depending on when the syscall was started and how many concurrent syscalls are in flight. Orthogonally, we derive a *correctness property* that ensures the sharing of the correct version among inflight syscalls. All writes end up on the most recent version of the objects, allowing forward progress. While we implement this invariant in our Midas prototype for the Linux kernel, our defense applies to any modern OS kernel.

Our evaluation of Midas demonstrates low performance overhead. On workloads from the NAS Parallel Benchmarks suite, Midas shows an average performance overhead of 3.7%. Similarly, its performance overhead on more kernel-intensive workloads from the Phoronix Test Suite is 3.4% (with negligible memory overhead). Our security evaluation demonstrates how Midas successfully stops all attacks against vulnerable syscalls. Our contributions are:

- Distillation of TOCTTOU attack vectors into an invariant that protects the kernel against malicious concurrent modifications,
- Midas, a design that prohibits and detects TOCTTOU attacks against modern kernels, prohibiting their exploitation, enabling developers to detect TOCTTOU bugs, and providing the foundation for safe syscall interposition and validation, and
- An efficient implementation of Midas for the Linux kernel that exhibits low (3.4%) performance overhead.

2 Background

Midas orchestrates several mechanisms within the Linux memory subsystem to provide its protection guarantees. Linux uses architecturally defined per-address space page tables to define mappings to pages. Midas protects these pages by temporarily marking them read-only in the page tables. This section provides background information necessary to reason about why and how Midas protects syscalls from concurrent writes.

2.1 Page Tables and Memory Protection

Virtually all modern architectures (e.g., x86, ARM, SPARC, and RISC-V) implement separate virtual and physical address spaces (AS) based on fixed-size regions called pages. Programs execute in their virtual address space while caches and main memory are accessed using physical addresses. Architectures rely on page tables orchestrated by the operating system to translate between these address spaces and to protect such accesses. Page tables are arranged as radix trees, where different bits of the virtual address are used as indices into successive levels of the page table. At the leaf page table, a unique pagetable entry (PTE) stores the translation and protection information for a page.

A PTE in x86-64 is a 64-bit value holding, among others, the following metadata: a *Present bit (P)* to mark the PTE's validity; *Protection bits (NX, R/W, U/S)* to restrict the type of access and the privilege level of the accessing code; *Software-usable bits (SW1-SW4)* that are ignored by the MMU and used by the operating system to store metadata; and a *Page Frame Number (PFN)* to identify the page's physical address.

An access using a virtual address first reads the corresponding PTE's present bit to check its validity. Then, the access checks whether the access is allowed from the executing code's privilege level by checking the U/S bit and whether the read/write access is allowed by checking the R/W bit. When all checks pass, the processor uses the PFN to find the data in the caches or in memory. When a check fails, the processor raises a protection fault/exception and moves control to an OS-specified exception handler.

Reading PTEs from a multi-level page table is an expensive operation, and modern processors cache PTEs in caches known as Translation Lookaside Buffers (TLBs) to reduce the cost of subsequent accesses. On most architectures, the OS is responsible for keeping TLBs coherent with the page table, necessitating entries to be flushed from TLBs when the corresponding PTE is updated.

2.2 Linux Memory Subsystem

Linux implements various abstractions—including processes, files, and shared memory—using the architecture's page tables. All threads within a Linux process share a single address space, and consequently use the same page table for translation and protection. Each page within the process' virtual address space may be mapped or unmapped. Mapped pages have separate read/write/execute permissions. Programs typically have write-execute exclusion, meaning code pages cannot be written to and data pages cannot be executed. These permissions map directly to page-table bits. Pages in Linux may also be copy-on-write (COW) pages, which are mapped read-only in multiple address spaces, but duplicated when any process writes to it, resulting in a separate copy.

Linux maintains userspace and kernel mappings to memory in distinct parts of the virtual address space. The PTE entries for kernel mappings, located in the top half of the address space, have the U/S bit set. These kernel mappings are identical for all address spaces, and are kept consistent across the corresponding page tables. In contrast, the PTE entries for userpace mappings, located in the bottom half of the address space, have the U/S bit reset. A userspace page has at least one userspace mapping and at least one kernel mapping. Shared userspace memory is implemented by mapping a page in more than one address space.

Files in Linux occupy a separate namespace (rooted at /). However, when files are read or written, parts of the file are cached in the kernel's page cache (which consists of pages mapped in the kernel's address space). Moreover, programs can explicitly map pages from a file, in which case the corresponding pages from the page cache are also mapped at userspace addresses in the process' page table. Mapped file pages can therefore be accessed by the file-system driver using kernel addresses, and userspace programs using userspace addresses. Userspace pages not backed by a file are called *anonymous pages*.

2.3 Supervisor Memory Protection

Kernel accesses to userspace memory use userspace mappings, introducing the risk of the kernel confusing userspace data structures for kernel data structures. An attacker can exploit this behavior via bugs in the kernel. Essentially, the attacker needs to set up either data structures or code within its accessible memory, then exploit a kernel bug to make the kernel use these data structures, or execute this code.

Architectures and OSs have mitigated these vulnerabilities by introducing *supervisor memory protection*. Under supervisor memory protection, kernel read/write/execute access to userspace memory raises a fault (depending on the state of a per-core system register). On x86-64, these features are known as Supervisor Memory Access Protection (SMAP, for data accesses) and Supervisor Memory Execution Protection (SMEP, for code accesses). Bits in the CR4 register track whether these protections are active, and the privileged `stac`/`clac` instructions are used to quickly enable and disable SMAP. In the OS, all accesses to userspace memory are made explicit, using *transfer functions* to read from and write to userspace memory. Any unintended access outside of these functions causes a hardware fault, indicating a kernel bug or an attack. Linux implements the `copy_{from/to}_user` functions, which use the access control instructions to disable SMAP before accessing userspace data, and then re-enable SMAP afterwards. Transfer functions make kernel accesses to userspace data explicit, allowing Midas to reliably track and protect all kernel fetches from userspace memory.

2.4 Double-Fetch Bugs

Double-fetch bugs occur when a privileged environment (such as the kernel) reads untrusted memory multiple times, returning different values each time. Such a situation is depicted in Figure 1, where the value of `X` in memory is changed by an attacker between two reads by the target thread. Exploiting such a bug requires a race condition, i.e., accesses to memory in a particular order across threads. A specific variety is the *time-of-check to time-of-use* (TOCTTOU) bug which occurs when the first fetch validates an object's value and the second fetch uses the same object's value. TOCTTOU bugs are widely studied in file systems, where the API makes it possible to swap the file after validating the access rights [20, 32, 33, 37, 43]. TOCTTOU bugs affect both kernel [23, 40] and dynamically-loaded driver code [7, 14]. Wang et al. [40] showed that double fetches appear not only in kernels, but wherever there is a trust boundary to cross (e.g., kernel—hypervisor [44] and hardware—kernel boundaries [28]).

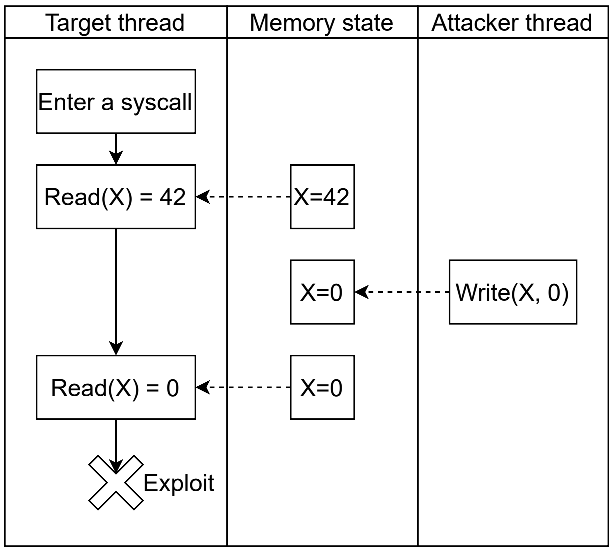

Figure 1: Example of a double-fetch bug.

	Userspace	Kernel	Device
Existing mapping	Intra AS Cross AS	User mapping Kernel mapping	DMA MMIO page

New mapping	`mmap`
	`clone`
	`swap`
		`mm_populate`	New DMA/
			MMIO page

Table 1: Attack vector classification for TOCTTOU exploits.

3 Threat Model

The attacker has access to a user account on the target machine. They can execute arbitrary userspace code, including syscalls. Some of the system calls have double-fetch vulnerabilities which the attacker wishes to exploit (e.g., for privilege escalation). The attacker may execute arbitrary sequences of syscalls on multiple CPU cores in parallel, or concurrently on the same core.

Midas mitigates any unintended corruption or information leakage *in the kernel* or *in other user processes* that arises through double-fetch bugs. Hardware attacks such as Rowhammer [30] or side-channels [26], and file-system TOCTTOU attacks [32, 33, 37, 43] are out of scope.

4 Attack Classification

Midas guards data processed during a syscall's execution against concurrent modification. We label the data fetched twice as vulnerable data. In this section, we classify attacks based on two criteria: the privilege level of the writer, and whether the mapping used for writing exists at the time of the first read. Table 1 summarizes our classification. Importantly, this classification helps understand existing attacks and how to protect against them, and where future attacks (bugs) may arise. The device column corresponds to attacks where a device (e.g., a network card, GPU, FPGA) is responsible for modifying vulnerable data. Watson [42] describes a subset of the following attack vectors.

Existing userspace mappings to a page can be used to modify vulnerable data which the targeted syscall is reading. Userspace can directly write to a mapped page, irrespective of whether the mapping is in the same address space or not. Alternatively, a concurrently executing syscall can also modify the vulnerable data in a *confused-deputy* attack. When the attacker passes a pointer to the vulnerable data to the syscall as a user buffer in which the syscall can return some data, the kernel's write to the buffer can modify vulnerable data. For example, the `read` syscall takes an argument pointing to a user buffer where the contents of a file will be copied to. Another example is `rt_sigaction`, where the kernel writes to a user buffer pointed to by the `oldact` argument. In both of these attacks, the malicious write uses a userspace mapping. *A protection mechanism must account for all userspace mappings to pages containing vulnerable data at the time of the targeted syscall's first read.*

Existing kernel mappings to a page also mapped in userspace can be leveraged by an attacker in a confused-deputy attack. Here, the attacker maps a file-backed page from the page cache into a userspace process and then passes as an argument in this page to the target syscall. The attacker then triggers a concurrent `write` syscall to modify the vulnerable data using kernel mappings for the page cache pages. The kernel does not explicitly track kernel addresses mapping to a page, but the file-system driver does explicitly find the page before writing to it. *A protection mechanism must instrument file-system drivers to account for writes via kernel mappings to vulnerable data.*

The kernel might create new mappings to the vulnerable data between the double fetches by the target syscall, bypassing any protective permissions installed by the transfer function in PTEs at the time of first read. An attacker can call `mmap` and `clone` to create a new mapping to the vulnerable data before writing to it. The page-table mapping might not be created at the time of the malicious syscall, but lazily when the attacker writes to the vulnerable data due to demand paging. In a more involved variant, the attacker can use the kernel as a confused deputy which touches the unmapped page and maps it in, then writes to the vulnerable data. In all of the above vectors, the function populating pages for a process (`mm_populate` on Linux) is creating the new mapping. *A protection mechanism must instrument any syscalls and other kernel mechanisms which can create new mappings.*

Swapping may also create a new page-mapping. If the attacker writes to a page that was previously swapped to disk, but later swapped in to be read by the target syscall in a different address space, the kernel might lazily reinstate the attacker's mapping to the page. *The swapping mechanism must be protected.*

Midas protects against all of the previously-listed attack vectors. In the absence of any other syscall which can create new userspace mappings to vulnerable data, Midas' protection is complete against writes from both user and kernel code.

Finally, a device might modify vulnerable data if it is either allowed to DMA (direct memory access) to the page, or if the page is memory mapped (MMIO) and is actually backed by the device. In the latter case, external factors can change the vulnerable data. Existing discretionary access control rules typically prevent users (except a superuser) from mapping device-backed pages into their address spaces. Such users are also disallowed from configuring DMA devices. Thus, device modifications to vulnerable data fall outside our threat model and are not protected by Midas. However, Midas can be extended to protect against modifications by DMA devices on processors supporting IOMMUs or similar methods for access control [31]. As a superuser can modify kernel code via kernel modules, protecting against attacks from this user falls outside of our threat model.

5 Midas Design

Midas maintains a single, core, *invariant*: ***through a syscall's lifetime, every read to a userspace object will return the same value***. By construction, the invariant guarantees that double-fetches in syscall code will read the same data, *eliminating TOCTTOU bugs*. Midas maintains the invariant by tracking *snapshots* of objects when first accessed, lazily making *copies* when the object is concurrently written and accessing the correct copy on subsequent reads. Copies are only maintained during syscalls' lifetimes, and are released as soon as no syscall needs it. Consequently, each userspace object has a single copy when no syscalls are running. The invariant also means that only accesses to userspace objects by the kernel need to be protected. Accesses to userspace objects from userspace and kernel objects by kernel code remains unaffected.

Midas' implementation builds on the protection mechanisms provided by existing virtual memory implementations. On modern platforms, virtual memory protection is set up by the OS at page granularity by setting bits in pagetable entries (PTEs). These permission bits are checked by the hardware on memory access, efficiently enforcing the permissions, and raising a fault when they are violated. For performance, Midas implements its invariant at page granularity, not object granularity: when a syscall reads from userspace, every page touched by that read is covered, not merely the bytes read. Page-granularity protections are conservative compared to byte-granularity protection and Midas maintains its invariant. As a side-effect of its implementation, Midas does not distinguish accesses to different parts of a page (intra-page false sharing). False sharing leads to unnecessary page duplications, incurring performance overhead on highly shared pages, but does not affect correctness.

For an object spanning multiple pages, Midas' design sequentially protects each page before reading from it. The leading pages containing the object are protected before the later pages, allowing an attacker to potentially modify the later pages before the syscall first reads them. However, the attacker is prevented from modifying any of these pages after the syscall's first read, ensuring that double fetches respect the invariant. If the syscall code contains a TOCTTOU bug, the modification will be visible to the first fetch itself (which is used for checking for validity of the data) and will lead to the data being rejected straightaway. Midas' invariant therefore prevent exploitation of double-fetch vulnerabilities even when the fetched objects span multiple pages. We elaborate on this case with an example in Section 5.2.

A major requirement for Midas is to allow concurrent access to pages by user/kernel code running in parallel with a syscall which reads from the same pages. This requirement prevents deadlocks and improves performance *vis-a-vis* a naïve design which blocks all other tasks writing to pages already read by a syscall until the syscall completes. The naïve design can deadlock because it introduces dependencies between tasks for forward progress, which we illustrate in the following example of a system with two tasks (A and B): *i*) Task A issues a blocking syscall which reads a user page and blocks, then *ii*) Task B writes to the same user page before issuing a syscall which resumes task A. In this case, if Task A's read to the page preceeds Task B's write, Task B will be blocked waiting for A to complete its syscall. Task A will also remain blocked waiting for Task B's syscall, introducing a circular dependency, leading to deadlock. The naïve design also introduces unnecessary delays in other cases, such as the one described below, again with two tasks (C and D): *i*) Task C reads from a page and sleeps for a long while, but does not read from the page a second time, then *ii*) Task D writes to the same page after task C has read from it, and blocks until Task C completes and is unnecessarily delayed. A more performant approach is to duplicate the concurrently accessed page: the copy is kept for task C for future fetches, and task D can write to the original and proceed without delays.

Midas must maintain multiple versions of a page read by a syscall to maintain its invariant in the face of concurrent writes. Midas introduces *snapshots* and *copies* to keep track of page versions. Snapshots are logical views of the page's contents at a particular time, while the actual contents are stored in one of many copies. Each snapshot maps to a copy, allowing the contents of the page at the time of creating the snapshot to be read. If multiple snapshots are taken without intervening writes to the page, these snapshots will map to a single copy, reducing Midas' space overheads and performance overheads for creating copies. Midas maintains a snapshot of every page when first read by a syscall. On a double fetch by the same syscall, the copy mapped to the snapshot is accessed, ensuring that the data read is the same as the first time. The latest copy of the page is used for all writes, by the syscall as well as from concurrently running tasks, updating the page as seen from userspace. Midas' design draws parallels to multi-version concurrency control methods for databases based on snapshot isolation [45]. Transactions read from a snapshot of the database state from when they started, and writes update the up-to-date state of the database. *Essentially, Midas is a multi-versioning system for pages where syscalls read from immutable versions to prevent TOCTTOU bugs and syscalls and userspace both write to a single mutable version holding the latest state of the page.*

5.1 Page State Machine

To track multiple versions of the contents of a page when being concurrently accessed by numerous tasks, from userspace or during a syscall, Midas implicitly maintains a per-user page state machine. For a page, its corresponding state machine *i*) tracks snapshots for currently executing syscalls which have read it, *ii*) tracks copies of the page, and *iii*) maintains the mapping between snapshots and copies necessary for providing the correct contents to subsequent reads.

Figure 2 shows the state machine for a single page. At every state, the page has two associated sets: *i*) the copies set $C = \{C_L, C_0, \dots\}$ holds multiple copies of the page over time, and *ii*) the snapshots set $S = \{L, S_0, S_1, \dots\}$ tracks logical versions of the page, each corresponding to one executing syscall and each mapping to a copy. Reads from kernel code in a syscall use the *snapshot's corresponding copy*. Writes from user/kernel code and reads from userspace access the *latest copy* C_L, which is mapped in processes' address spaces. All other copies are read-only (no matter what the original page protection is), and are used for providing snapshots to syscalls. Read-only pages only use states 0 and 1, and writes lead to segmentation faults (as they do on non-Midas systems). Knowing which state the page is in allows Midas to differentiate between faults due to Midas protecting pages and faults due to actual permissions violations in userspace programs or the kernel. The latest copy C_L of read-only pages remains read-only in both protected states (1 and 3). In the following paragraphs, we describe how the state machine for a single, writable user page transitions between its states, what triggers each transition, and what changes are made to the copies and snapshot sets on a transition. In Figure 3, we illustrate how the state machine protects the syscall from Figure 1.

State 0. A page starts as `(unprotected, unduplicated)`. In this state, there is a single copy C_L and a single "snapshot" L. The snapshot L refers to the latest version of the page which changes over time, and is

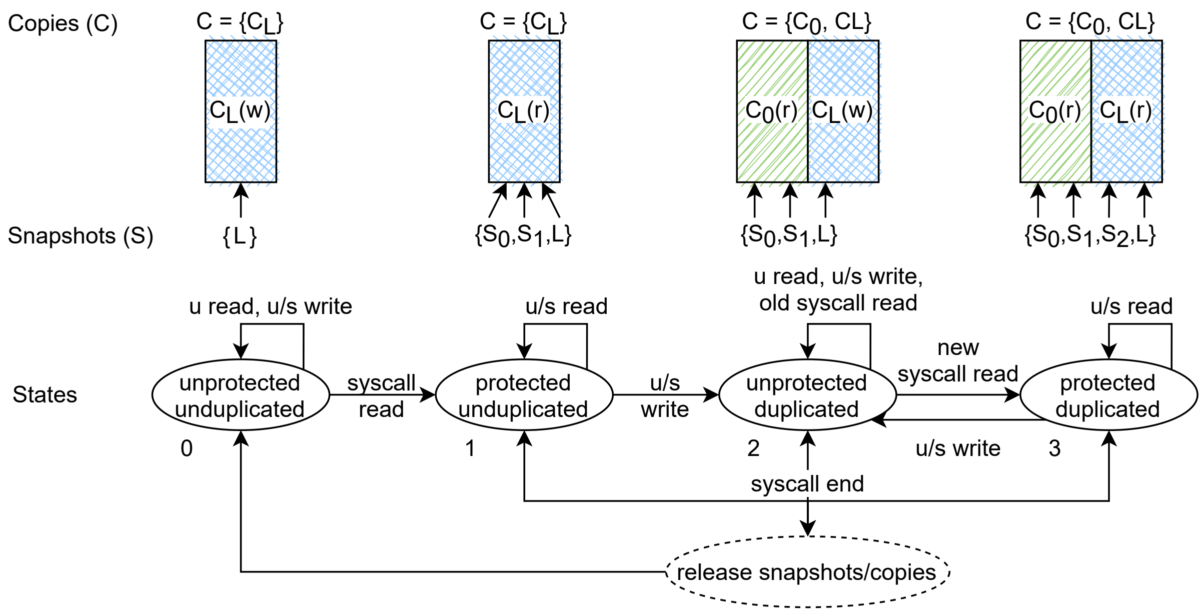

Figure 2: State diagram for a page in Midas. Reads/writes from userspace/syscall code are marked (u)/(s) respectively. Shading is used to represent the mapping from snapshots to copies.

the only mutable snapshot. All processes where this page is mapped have unrestricted userspace read and write access, and unrestricted kernel write access. The remaining operation, a read from kernel code, triggers a transition to State 1. In Figure 3, the snapshot L initially contains the value 42.

State 1. The page in State 0 transitions to the `(protected, unduplicated)` state as soon as a syscall reads from it. Midas first marks the page's latest copy C_L read-only in all processes, trapping writes to the page but allowing concurrent userspace reads to continue. A new snapshot, S_0 linked to this syscall is allocated for this page. For the rest of its lifetime, this syscall will only read this page from this snapshot. Both snapshots S_0 and L refer to the same copy C_L (shown by the blue cross-thatch in Figure 2). Prior to any writes to this page, any other syscalls which also read the page get their own snapshots (e.g., S_1) all pointing to the single copy C_L. The page's read-only status causes the hardware to fault on any write, notifying Midas to transition the page to State 2. In Figure 3, the page transitions to State 1 when the syscall first reads it, and adds a snapshot S_0.

State 2. A page in State 1 transitions to the `(unprotected, duplicated)` state on any write from user or kernel code. Midas duplicates the old contents of the page from copy C_L, creating a read-only copy C_0 (shown by green shading in Figure 2). Snapshots except L (i.e. S_0 and S_1) previously mapping to C_L are mapped to the copy C_0. The write then modifies the latest copy C_L, which is made writable again. Note how, in this state, any read using the snapshots S_0 or S_1 reads from the unmodified copy C_0 while writes directly affect C_L. Certain syscalls such as `rt_sigaction` both read and write from the same user page. A write by `rt_sigaction` to the page it has previously read will update the page's latest copy C_L, but not the duplicate copy C_0. Midas' write policy ensures that the copy C_L always holds the latest contents of the page, up-to-date with all the writes to the page, from both user and kernel code. Further, Midas does not need to merge writes from userspace and syscall code on a syscall's completion, since both directly modify the same copy C_L. All other copies C_i are immutable. When the attacker writes to the page in Figure 3, the page moves to State 2, linking the snapshot S_0 to a copy holding the original value 42. The writes from both the attacker and the syscall itself both affect the copy C_L, but the read from the syscall accesses the snapshot S_0 and reads the same value as the first time.

State 3. A separate syscall subsequently reading the page in State 2 transitions it to the `(protected, duplicated)` state. The new snapshot, S_2, points to the latest copy C_L. State 3 is similar to State 1, except that there are different copies of the page used for reading by different syscalls. The syscall for which S_0 was allocated will read from the copy C_0, while the syscall for which S_2 was allocated will read from copy C_L. On a write, the page transitions to State 2 and is duplicated again, creating another copy C_1: snapshot S_2 maps to C_1 while snapshots S_1 and S_0 continue to map to C_0.

Releasing snapshots. Midas uses snapshots to enable a syscall to read the same data from a page during its lifetime and releases snapshots when syscalls complete. Releasing a snapshot is possibly accompanied by a state transition and the release of the mapped copy. If S_i mapped to the latest copy C_L, Midas cannot free the copy since userspace is using it. In

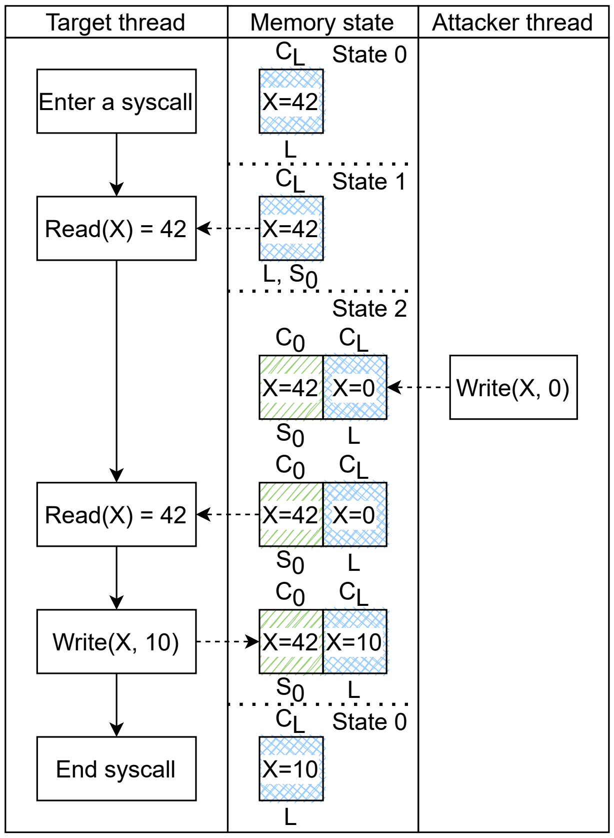

Figure 3: Diagram illustrating Midas preventing exploitation of a double fetch of object X.

this case, the page must be in State 1 or 3, and C_L is read-only. After removing S_i, if L is the sole remaining snapshot mapped to C_L, Midas makes the page writable, moving to State 0 or 2 from State 1 or 3 respectively. If S_i is mapped to any other duplicate C_i, Midas frees the copy along with the snapshot if S_i is the last remaining snapshot mapped to C_i. If the page was in State 2, C_L was writable and unmapped by any snapshot, so Midas changes the page to State 0. This transition is shown in Figure 3, where the snapshot S_0 and the copy C_0 are both discarded. If the page was in State 3, C_L was read-only and mapped by some other snapshot, so Midas moves the page to State 1. Recall that all snapshots S_i except L are immutable. Any data written by the syscalls directly affect L. Therefore, dropping a snapshot S_i is trivial and does not require writes from the syscall to be merged into the latest copy.

5.2 Discussion

Correctness of syscalls directly updating snapshot *L*. Midas' design lets all writes, including those from syscalls, to directly update the latest copy of the page C_L and this property maintains correctness of system execution. We now show that there is a valid, safe execution trace of a system not protected by Midas which generates the same sequence of writes to the page, and therefore generates the same contents of the page when the syscall ends. We define a *safe* trace as one that has no writes to vulnerable data between double fetches by the kernel, and therefore does not trigger any existing TOCTTOU bugs. By showing that the final contents of memory after a Midas syscall has a corresponding execution without Midas (which we assume to be correct) leading to the same contents, we can conclude that the execution of the Midas syscall is also correct. For this proof, we assume that no syscall reads the same object after writing to it (r-w-r pattern). Such syscalls do not exist in the Linux kernel, and are discussed below. Therefore, our syscalls write to an object after completing all of their reads of that object.

Consider a page holding a single-byte object O_0, and the sequence of operations to this byte during a Midas syscall be $Ops = \{Op_0, Op_1, \dots\}$. Each operation is a tuple $(r/w, k/u)$ specifying whether the operation was a read or a write, and whether the operation was due to a user or kernel instruction. Suppose there was no attempt to exploit a TOCTTOU bug, i.e., between any two read operations by the same syscall, there was no write to this object. In this case, Midas reads the same value from its snapshot of the object as is present on the latest version. The same sequence of operations on a non-Midas system would be valid and safe, since the object value does not change between the kernel's double fetch and the syscall reads the same value on this system.

Assume there was an attempt to exploit a TOCTTOU bug: a write Op_1 exists between two syscall reads Op_0 and Op_2. Midas protects the syscall ensuring that Op_2 does not see the effect of Op_1 by reading from a snapshot instead of the latest copy C_L. Since our syscalls are assumed to not contain any r-w-r pattern, any writes by the syscall happen after Op_2. Let us assume that the syscall's write is Op_3. We can generate a valid, safe execution on a non-Midas system by moving the attacker's write to after the last read by the syscall, i.e., $Ops = \{Op_0, Op_2, Op_1, Op_3\}$. The syscall in this system reads the same value both times, and hence has the same execution as that in the Midas case. The value of the object when the syscall completes is that written by Op_3 in both cases (or that written by Op_1 when the syscall does not have a final write). Since the syscall has the same execution and the final value of the object is the same, the execution of the Midas system is the same as that of the non-Midas system. In general, any trace of operations on a Midas system can be translated to a valid,

System Call	Exemption reason
`futex`	Relies on concurrent write
`execve`	Remaps address space
`write`	Invulnerable, improves performance

Table 2: System calls uninstrumented by Midas.

safe trace on a non-Midas system by moving malicious writes to an object to just after the last double fetch of that object. Multiple syscalls in Midas can therefore write to the same object without affecting correctness, because an equivalent, valid, safe non-Midas trace exists where all of the writes have been postponed, in the same order to after the double fetch reads.

Exemptions. Syscalls such as `futex` rely on user data changing between double fetches to implement their functionality and cannot be protected by Midas. These syscalls are listed in Table 2. The `futex` syscall implements a fast synchronization mechanism for userspace and relies on atomic writes from concurrent userspace threads to update a condition the syscall is waiting for. Subjecting a `futex` syscall to Midas' invariant will prevent it from ever waking up the waiting task. Such syscalls cannot be protected by Midas, and we implement an exemption list to prevent transitions in the state machines of pages read by these syscalls. The code for exempted syscalls must be manually inspected for double-fetch vulnerabilities. Crucially, exempting these syscalls from Midas' protection does not affect the security of other syscalls containing double fetches. Any writes from these syscalls are subject to the same rules described in the state machine, and cannot break Midas' invariant. Midas can also implement finer-grained exemptions based on syscall parameters. Those were not necessary for Linux.

Syscalls with read-write-read patterns. A (hypothetical) syscall that reads from an object, writes to it, and then reads back the updated object cannot be protected using Midas. Midas' invariant will ensure that the second read is identical to the first, and does not reflect the intermediate write. Such syscalls must remain exempt from Midas' instrumentation. During extensive tests, we did not find any other syscall which exhibits this behavior in the Linux kernel.

Syscalls with false sharing. Another hypothetical type of syscall could struggle with Midas' instrumentation due to false sharing. Suppose a page contains two objects, O_0 and O_1, and a syscall sequentially reads O_0 then O_1. Due to Midas' invariant being enforced at page granularity and false sharing of the page between these objects, Midas guarantees that the value of object O_1 read is the same as what was contained when it first read object O_0. A syscall requiring the value of O_1 to change between these two points in time would not work with Midas' protections. Such a hypothetical syscall, requiring concurrent modifications to its arguments, could exist to support some synchronization mechanism similar to a `futex` and can be safely exempt from Midas' invariant. During extensive tests, we did not find any other syscall which exhibits this behavior in the Linux kernel.

Example: Objects spanning multiple pages. Figure 4 shows Midas protecting a syscall which has a double fetch for an object spanning multiple pages. Here, the two pages containing the object `X` are accessed as `X[0]` and `X[1]`. The attacker tries to attack the syscall by changing the value of

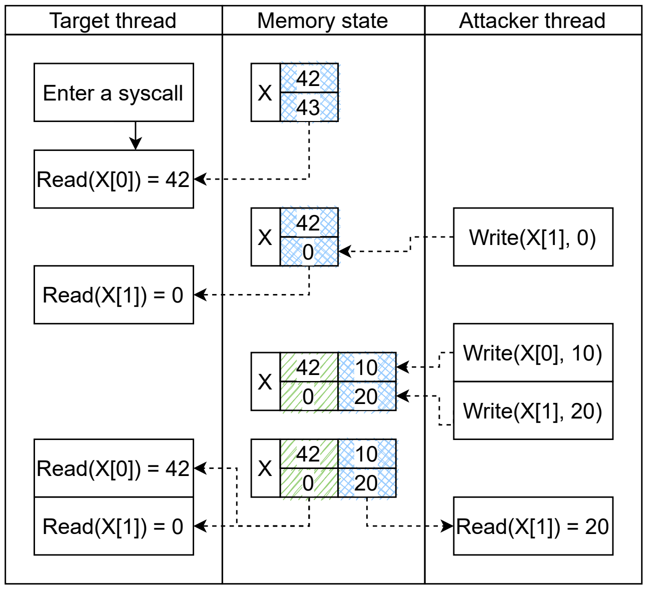

Figure 4: Diagram illustrating Midas preventing exploitation of a double fetch of an object `X` spanning two pages.

the second page: *i*) between the syscall's first reads of `X[0]` and `X[1]`, and *ii*) between the first and second fetches of X. Midas ensures both fetches return `X=(42,0)`. Critically, any existing TOCTTOU bugs are not triggered since both fetches read the same, possibly invalid, value of the object. Note how the situation is identical to one where the malicious write to `X[1]` happens before the syscall starts.

Preventing deadlocks by design. Midas' design is free of deadlocks, and exempts syscalls which require violation of its invariant from triggering particular state-machine transitions. Userspace reads always succeed, using the latest copy C_L of the accessed page. Writes from userspace and kernel code succeed directly if the page is in State 0 or 2, and trigger a fault otherwise. Handling these faults involves creating a new copy of the page and setting the page writable. Reading from kernel code involves creating a new snapshot and setting the page read-only. None of the aforementioned operations relies on other operations on the same page to complete and all are finite time. None of the operations on a page rely on operations on other pages. A single, per-page lock can serialize operations on that page and assure forward progress.

Detecting double fetches. Midas' state machine for pages enables the precise detection of double fetch bugs, turning it into an effective sanitizer and developer debugging tool in addition to being an efficient mitigation. When a syscall first reads from a user page, it creates a snapshot of that page. On future reads, the snapshot is used in order to maintain the invariant. While reading from a page, implementations must check if a snapshot exists for the syscall: if yes, the snapshot is used for the read, otherwise a new snapshot is created and then used for the read. The existence of a snapshot means the syscall had previously read from this page and had then created this snapshot, implying a double fetch. Unfortunately, this approach is prone to false positives due to false sharing. The two reads might read from the same page, but access entirely disjoint bytes. Midas currently reports double fetches at page granularity. A precise sanitizer could maintain a bitmask of accessed bytes to prune false positives.

6 Midas Implementation

Our Midas prototype implements the state machine described in Section 5 on Linux version 5.11, targeting the x86-64 architecture. A page protected by Midas transitions between states on either a kernel read to user memory, or when user or kernel code writes to protected, read-only memory (see Figure 2). Midas can be implemented on any operating system kernel that *i*) systematically uses transfer functions for reading from userspace, and *ii*) on any architecture which implements hardware-controlled access control to memory through page tables. The first requirement enables Midas to implement transitions on kernel reads from user memory. The Linux kernel uses the `raw_copy_from_user` interface which we instrument for our prototype. The second requirement causes the hardware to raise a fault on writes to Midas-protected pages, directing execution on the processor to a pre-defined exception handler in the OS. Our prototype instruments Linux' fault handler in the function `handle_pte_fault` to implement the write-triggered transmissions from states 2 and 4. Overall, our prototype adds around $1,100$ lines of code and modifies 17. Our design allows the changes to be mostly limited to the memory subsystem, and in general does not require individual syscalls to be modified. Only one syscall (`clone`) required code modification.

6.1 Tracking Page State

Midas needs to track the state for every userspace page, including its snapshot and copy sets. Figure 5 shows the data structures used to track a page's state in our prototype. Linux maintains a `struct page` object for every frame of physical memory. We augment `struct page` with a list holding the snapshots for this page, excluding the latest snapshot L. Each snapshot has a pointer to its copy. In the figure, the snapshots S_1 and S_0 share the copy C_0. We are aware of the strong aversion of the Linux kernel developer community towards increasing the size of `struct page`. An alternate implementation can use a hashmap to map from a page's frame number to its snapshots list or reuse existing data members (e.g., `struct list_head lru` which can be used as a generic list by page owners).

Each pagetable entry for a user page in different address spaces maps the copy C_L, enabling userspace to directly access the page with reads (and writes for writable pages). We

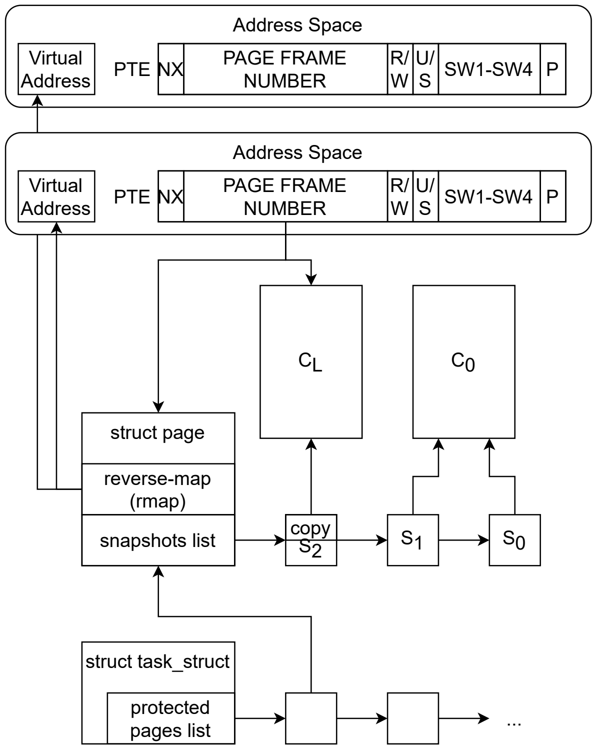

Figure 5: Bookkeeping information for a page.

use one software-controlled bit (SW3) in the pagetable entries to track the protection status of the page, and another (SW2)[2] to track the original protections for the page. SW3 is set whenever the page is in one of the two protected states (1 and 3). On a write-triggered protection fault, SW3 can be read to efficiently determine if the fault was due to Midas' protection mechanisms, triggering a state change, or due to buggy software accessing a page with illegal permissions, triggering a signal to the task. Other architectures might have fewer software-usable bits in the page table, and implementations of Midas would require storing the protection status of pages in a separate data structure. The duplication status of the page is implicitly encoded in the snapshots: the page is duplicated when any of its snapshots holds a pointer to a copy other than C_L.

Changing a page's protection state requires PTE updates in all address spaces where the page is mapped. The page's `struct page` structure includes a reverse-map listing for all of these pages, and the corresponding virtual address in each. Our prototype uses this mapping to change PTE permissions across all address spaces for a page.

[2]The SW2 bit is alternatively used by the experimental Software Dirty Pages feature of Linux, and cannot be run alongside Midas in our prototype.

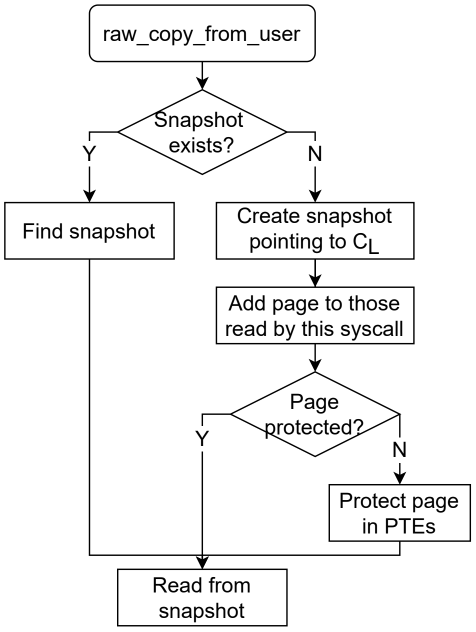

Figure 6: Flowchart for a syscall using the transfer function `raw_copy_from_user` for reading from userspace.

6.2 Kernel Reads from User Memory

Syscalls reading from user memory the first time triggers the allocation of a new snapshot. If the page is not protected (states 1 and 3), the read also triggers a state change where the kernel protects the page in all address space that it is mapped in. Figure 6 shows the flowchart of the steps implemented by the kernel function `raw_copy_from_user` for reading from user memory. This function also uses the kernel's `mark_page_accessed` interface to move the page to the "Active" state for the kernel's swapping mechanism, making the page ineligible for being swapped out. We also implement `get_user` and `unsafe_get_user` (used by the kernel for small reads) as a call to `raw_copy_from_user`.

Exemptions. Our prototype Midas kernel exempts a couple of functions from Midas' invariant (in addition to those described in Section 5.2), and these functions are therefore not instrumented to follow the aforementioned steps while accessing userspace memory. First, `raw_copy_from_user_inatomic` is a special transfer function used by the kernel to read user memory in special situations such as a kernel oops[3] where the kernel reads user memory to provide a backtrace. In this severe situation, the kernel's goal is to collect debug information before its imminent termination and no TOCTTOU protection is needed. Second, we also exempt the `write` system call's reads from user memory from instrumentation. The `write` syscall takes three arguments: a file descriptor passed as a register, a pointer to a user buffer and a count of bytes to be written to the file. While the write to the file's pages is sensitive, and Midas takes care to ensure that it follows the page state machine, the read from userspace is not. The syscall reads from userspace only once, and its data is only used for copying into the file. An attacker who modifies the user buffer concurrently with the syscall only manages to change the contents written to file, which it could have done anyway since it has access to this buffer. A kernel developer can similarly exempt other syscall which they can prove to be secure from double-fetch bugs.

[3]A kernel oops is triggered when the kernel detects a problem while running which can affect its proper functioning, such as corrupted data structures. A more severe version, a kernel panic, causes the kernel to stop executing, expecting data loss or damage if it does.

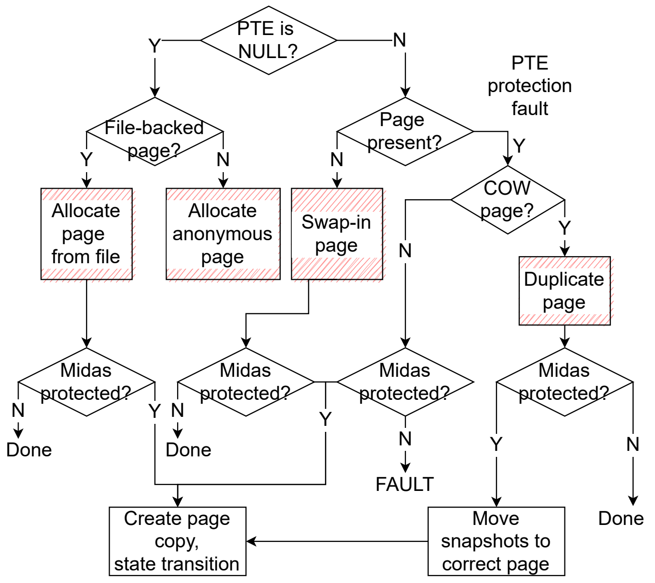

Figure 7: Flowchart for handling a page fault. Shaded operations are unmodified.

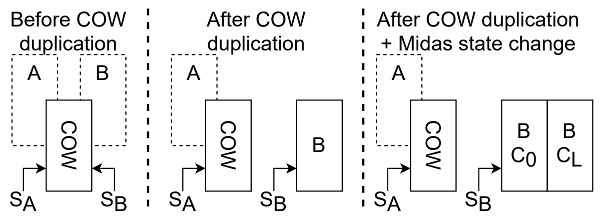

Figure 8: Flowchart for handling a page fault to a COW page.

6.3 Handling Faults

The memory management unit generates a fault when kernel or user code accesses a page without having the correct permission in the corresponding PTE. Midas marks writable pages read-only to protect them in states 1 and 3, allowing the kernel to detect writes to these pages. A common OS mechanism, copy-on-write (COW) pages, also uses permissions in the PTE to detect when COW pages need to be copied.

The PTE's present bit are used to store pointers to file-backed pages when they are swapped to disk. Figure 7 shows the flowchart implemented by `handle_pte_fault` to handle faults for userspace addresses.

The page-fault handler first checks if the PTE is NULL, and if so knows that it must allocate a page. If the required page is anonymous, the page can be allocated as usual. Otherwise, for file-backed pages, the handler has to check if the page is already in a protected state (states 1 and 3) by reading the SW3 bit of the PTE and if so, transitions to the required state and allocates a new copy. Pages in states 0 and 2 can be directly mapped, and subsequently accessed.

For non-NULL PTEs, the handler checks if the PTE indicates that the page is present. Non-present pages need to be swapped in. After finding the page, Midas then checks if the page was previously swapped in by any other task and is now in a protected state. For protected pages, Midas implements the required state change based on whether the faulting access was a read or a write.

In the remaining case, faults for a present page indicate a permission fault (for example, a write to a read-only page). If the page is not a COW page, the handler then checks if the page is in a protected state by checking the SW3 bit. If the page was protected, a new copy is allocated and the page transitions to the following state. For non-protected pages, however, the fault implies a real access violation, sending a signal to the process.

COW pages represent separate virtual pages from different address spaces mapped to the same physical page. An example of a COW page protected by Midas is shown in Figure 8, where logically-separate pages A and B are actually mapped to the COW page. COW pages cannot be in states 2 or 3, since they cannot have multiple Midas copies. COW pages in state 0 can be dealt with by the kernel's standard duplication method (not Midas' duplication). For a COW page in state 1, its list of snapshots can correspond to reads from syscalls for threads in different address spaces. In Figure 8, we show snapshots S_A and S_B corresponding to syscalls for threads in different address spaces (containing A and B respectively). These snapshots correspond to different logical pages, but are all squashed into the snapshots list of the single COW page. Therefore, after the kernel duplicates the COW page (new page B created, in Figure 8), Midas moves the snapshots for the faulting process (S_B) to the new page. Here, Midas also updates the protected page list in the affected syscalls' `task_struct`s so that these structures correctly refer to the new page. Finally, the new page is transitioned to its next state to allow for the write to occur, creating a new copy (C_0) for the snapshot S_B to read from.

We ensure that Midas' modifications to the fault handler correctly handle concurrent faults and do not cause additional nested faults. During concurrent faults for the same page, only one thread changes the page's state whereas the other directly uses the new state. The kernel's split page-table lock is reused to serialize state changes. We also ensure that the only additional accesses to user memory within the handler (used for duplication) happen when the page is assured to be in memory and correctly mapped. All nested faults are therefore caused by existing kernel code and do not interact with Midas' modifications.

6.4 Syscall Completion

On syscall completion, Midas cleans up snapshots allocated for the syscall by instrumenting the end of `do_syscall_64`. Midas goes through the list of all the pages for which the executing syscall has a snapshot, and frees those snapshots. For snapshots which were the last to point to a copy, that copy is also freed.

6.5 File System Writes

Midas instruments file-system writes to protect the kernel from modifications via kernel mappings. When a `write` syscall writes to a file, it actually writes to copies of pages of the file stored in memory within a page cache. In the spirit of abstraction, the kernel does not directly write to these pages, but calls the relevant file-system (FS) driver instead. The FS driver will access the page using kernel mappings when writing to pages in the page cache. Since Midas only protects userspace mappings for protected pages, writes by FS drivers will not raise a fault. To comprehensively protect the page, any implementation needs to instrument FS drivers' write functions. Fortunately, FS drivers provided with the kernel follow a simple recipe: for pages not in the page cache, the driver executes FS-specific code to read the page into the page cache and then call a generic function (`generic_file_write_iter`) to actually write the data into the page. Instrumenting this generic function, therefore, protects the kernel for a wide range of common file-systems (including ext4, nfs and ntfs).[4] The added instrumentation checks whether the target page is protected, and if so, transitions it to the next state and creates a copy of the page before writing to the latest copy.

Our current prototype does not, however, protect out-of-tree drivers which are not distributed with the kernel if they do not use the `generic_file_write_iter` function. A user with superuser privileges can load a insecure module implementing a FS driver which does not implement Midas checks. A malicious superuser is, however, outside our threat model.

6.6 New Mappings to Protected Pages

Our Midas prototype preserves the state machine for user pages across operations which create new mappings to a page

[4]A more comprehensive list of kernel-provided FS drivers protected via `generic_file_write_iter` includes v9fs, ADFS, AFFS, AFS, BFS, CIFS, eCryptfs, extFAT, ext2, F2FS, FAT, FUSE, HFS, HFS+, hostfs, HPFS, JFS, JFFS2, Minix, NILFS2, OMFS, OrangeFS, ramfs, ReiserFS, SystemV, UBIFS, UDF, UFS, VboxSF, shmem.

to prevent attacks which rely on mappings being created between double fetches. The `mmap` syscall is responsible for creating new virtual memory mappings for processes, and requires instrumentation. When `mmap` is called with the `MAP_POPULATE` flag, or on the first access to the page, the `mm_populate` function is responsible for actually mapping the correct page in the page table. In our prototype, we check if the page being mapped is protected, and if so, correctly protect the new mapping too. Another syscall, `clone`, duplicates a process' address space when called without the `CLONE_VM` flag, creating new mappings to pages. We instrument `clone` to ensure that new mappings for protected pages are also correctly protected.

6.7 Discussion

Optimizations on capable hardware. To protect a page in an address space, a Midas implementation needs to change the permissions in the page table for that page. Modern CPUs cache virtual memory translations in per-core Translation Lookaside Buffers (TLBs) which need to be (partially) flushed on page-table updates (TLB shootdown). On most CPUs, the core updating permissions will perform a global shootdown to ensure that other TLBs for cores executing in the same address space are also updated. Implemented with inter-processor interrupts, global shootdowns are expensive. In our evaluation, 21% of the runtime of the load generator `bombardier` used for stressing the Nginx server was spent performing TLB shootdowns when running on the Midas kernel.

A more efficient solution would be to have special hardware support for invalidating TLB entries globally, not just on the executing core. The AMD64 architecture manual [16] lists such an instruction (`INVLPGB`), though it is yet to be implemented in any commercially available x86 processors. The ARM v8-A architecture manual [27] lists similar instructions `TLBI ASIDE1`**`IS`** and `TLBI ASIDE1`**`OS`** which invalidate all PTEs of a page within a cluster of cores but not for cores in other clusters (Inner Shareable Domain) and cores across clusters (Outer Shareable Domain) respectively.

Alternate architectures [18, 22] with a single, system-wide translation table would also benefit Midas by having a single page table to update instead of multiple page tables for each address space a page is mapped in.

Porting to other OSs. Midas can provide TOCTTOU protection on other operating systems by tracking the states of each page and implementing state transitions as required. OSs track page state in per-page state structures, such as `vm_state_t` for BSD-based OSs (*BSD) such as FreeBSD and XNU. An implementation on these OSs must instrument the read transfer function(`copyin` for *BSD) to transition to states 1 and 3. The OS' fault handler (`vm_fault` on *BSD) will trap on writes to protected pages, and needs to be modified to implement the required page duplication and state change.

The remaining OS modifications for Midas support depends on the OS' features. If an OS allows userspace to map

```
1    //First fetch
2    if(get_user(count,&argp->dest_count))
3    {...}
4    //Using first fetch
5    size = offsetof(..., info[count]);
6    //Secong fetch
7    same = memdup_user(argp, size);
8  + //Added check for bug
9  + if(same->dest_count != count)
10 +   printk("Bug triggered");
11 + // Fix: copy over original count
12 + same->dest_count = count;
13   //Using second fetch
14   ret = vfs_dedupe_file_range(file,same);
```

Listing 1: CVE-2016-6516: Vulnerable double fetch in `ioctl_file_dedupe_range`. Lines in green show the fix and testing code.

file pages, filesystem code to write to these page needs to be modified. Other syscalls which create/modify mappings to userspace pages will also have to be instrumented to ensure that the new mapping respects the page's state. Such modifications are OS-specific, making it difficult to recommend a generic methodology.

7 Evaluation

In this section, we empercically verify Midas' ability to mitigate a known double fetch vulnerability, and quantify Midas' overhead on workloads with different characteristics including both compute-bound applications which rarely use syscalls and a mix of syscall-heavy applications which heavily rely on the kernel's performance.

7.1 Mitigation of CVE-2016-6516

CVE-2016-6516 is a known vulnerability in kernels prior to version 4.7 in a file-system `ioctl`. The vulnerable code is shown in Listing 1 and is triggered when the value of the `dest_count` object differs between the two fetches (in lines 2 and 7). `memdup_user` uses the value from the first fetch for allocating a buffer and copying in an array of descriptors from the user in line 7. `memdup_user` also contains the second fetch of `dest_count` which is later used in the function `vfs_dedupe_file_range`. An attacker who increases the size of `dest_count` between the two fetches will cause the kernel to access the copied array out-of-bounds, causing a heap buffer overflow.

For verifying Midas' defense, we introduce a non-faulting assertion check into the function (lines 9–10) and run a known exploit.[5] The condition checks whether the fetched value of the user object (`dest_count`) had changed, indicating a successful attack, and prints a message. Finally, we re-introduce the fix for the bug (line 12), fixing the value of `dest_count` in `same` to that from the first fetch. In this setup, we can detect when the conditions for triggering the bug are met, but also revert to a correct state allowing the kernel to safely continue. The exploit was able to successfully trigger the bug on the baseline kernel every time over 10 runs. With Midas enabled, the exploit was never triggered, i.e., both fetches returned the same value on every call.

7.2 Performance evaluation

We evaluate Midas on *i*) microbenchmarks targeting specific common syscalls, *ii*) workloads from two benchmark suites: the NAS Parallel Benchmark (NPB) [17] and select workloads from the Phoronix Test Suite (PTS) [15], and *iii*) the webserver Nginx. NPB includes compute-intensive multiprocessing workloads with a low, but non-negligible syscall rate. NPB therefore demonstrates the ability of Midas to scale to systems where pages are protected across numerous address spaces. PTS includes a variety of benchmarks, both compute bound and I/O bound representative of both desktop and server workloads. PTS includes syscall-heavy applications with varying degrees of parallelism. The Nginx webserver is capable of both high request service rates and scalability with multiple worker processes. We do not include the SPEC CPU2017 benchmarks as they are heavily compute bound and designed to isolate userspace performance without syscalls, and are impervious to kernel performance. They would unfairly bias performance in favor of Midas. **Mat** *Can we add a sentence like "For the sake of completeness, we measured X% overhead of Midas on SPEC CPU2017."?*

The testbench for the evaluation consists of a desktop machine with an 8-core Intel i7-9700 processor and 16GB DRAM running Ubuntu 20.04 LTS. This configuration and CPU is commonly used on desktop machines and workstations. To eliminate the effect of dynamic frequency and voltage scaling (DVFS), we set the processor to run at constant frequency of 3.0GHz which is this model's base frequency. In the *baseline* configuration, we run the testbench with the mainline kernel v5.11 available from Ubuntu's package repository. The *Midas* configuration runs our prototype Midas kernel also based on kernel v5.11. For particular benchmarks, we also run the *Midas+write* configuration which also runs our prototype Midas kernel but instruments all syscalls including `write`.

Microbenchmarks. We test Midas on microbenchmarks from OSBench [11]. The programs use `libc` interfaces such as `fopen`, `pthread_create`, `fork` and `malloc` for creating files, threads, processes and for memory allocation respectively.

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

Microbenchmark	Top syscalls used
File creation	`openat`, `fstat`, `write`, `close`
Thread/Process creation	`mmap`, `clone`, `exit`, `wait`
Program launch	`mmap`, `execvereadlink`, `openat`
Mem alloc	`brk`

Table 3: Prominent syscalls used by OSBench microbenchmarks.

Table 3 lists the prominent syscall usage for these workloads. Figure 9 shows Midas' performance (time per operation) on OSBench relative to the baseline kernel, with overheads ranging from zero to 5.3%.

NAS Parallel Benchmarks. NAS Parallel Benchmarks (NPB) [17] is a benchmark introduced by NASA. NPB consists of several parallel programs using different communication patterns and is available for two frameworks for parallel programming: OpenMP and MPI. OpenMP [19] is a compiler extension that splits a program's execution to multiple threads. All threads still use the same address space, keeping the overhead minimal. MPI [36] implements parallel execution by launching multiple processes which communicate by message passing. The two technology stacks have different frequency of syscalls due to different communication methods. Communication through kernel syscalls for either stack will incur overhead due to Midas' protection. Additional global TLB shootdowns (for snapshot synchronization) added by Midas will also affect the performance of such parallel benchmarks.

We run NPB benchmarks of class A on our testbench, executing 4 threads or processes in parallel. The benchmarks' runtime varies between 10 seconds and 8 minutes, and are all long enough for the kernel to reach equilibrium. Certain benchmarks require a parallelism number which is a perfect square. On our 8-core CPU, having 4 compute-bound threads/processes instead of 16 allows all threads to run without time sharing. Figure 9 shows Midas' performance for both MPI and OpenMP, normalized to the performance of the baseline system with the same parallelization framework. On average, Midas achieved 96.3% of the baseline system's performance on both frameworks. Midas' performance for the `ep` (Embarrassingly Parallel) benchmark is closest to that of the baseline, since it has low communication overheads. Midas shows low overhead (3.7%) for compute-intensive, parallel workloads.

Phoronix Test Suite. The Phoronix Test Suite (PTS) [15] includes a large set ($>$ 500) of open-source benchmarks, of which we have chosen a range of benchmarks suitable for evaluating both desktop and server performance. We bias the selection to benchmarks that require (heavy) kernel activity to test the overhead of Midas' instrumentation. A sole benchmark, OpenSSL, is included to represent single-threaded, compute-bound workloads for which kernel performance is less relevant. The benchmarks are also varied, ranging from single-

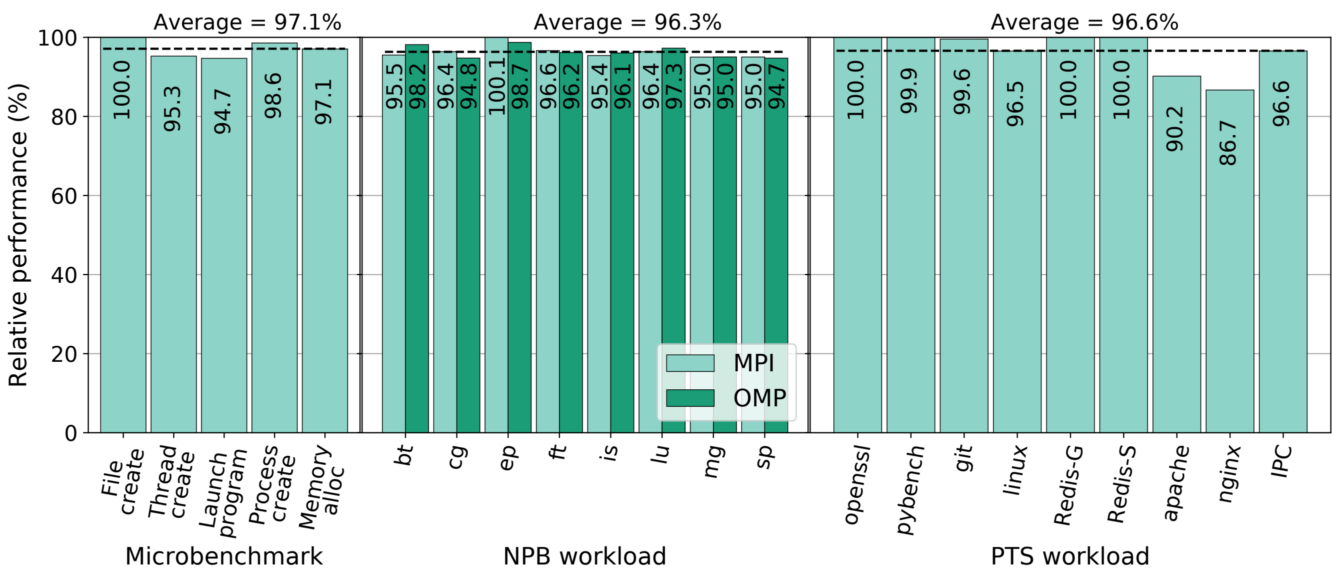

Figure 9: Midas' performance on microbenchmarks, NPB and PTS benchmarks relative to the baseline kernel.

threaded (Pybench) to multi-threaded, multi-process workloads (Apache). At the extreme, we have an IPC benchmark transferring tiny, 128-byte buffers between processes which spends all of its time in syscalls and whose performance is entirely dependent on kernel IPC performance.

We plot Midas' performance relative to the baseline kernel on these benchmarks in Figure 9, roughly ordering workloads in increasing order of syscall dependence from left to right. For benchmarks for which PTS reports runtime, we compute the inverse of the runtime as performance. Benchmarks with low syscall frequency such as OpenSSL, Pybench and Git have correspondingly low dependence on kernel performance. Accordingly, these benchmarks see a negligible overhead when running on our prototype kernel. The benchmark titled "Linux" represents compilation of the Linux kernel. While compilation is mostly compute bound, compiling the Linux kernel requires accessing a large number of source files, resulting in the creation of a large number of compiler processes each of which read and create files. Midas experiences a small, but non-negligible overhead of 3.5% on this workload. Redis requires syscalls for receiving and replying to requests, but processes its transaction entirely in-memory. Our evaluation prototype achieves practically identical results as the baseline, highlighting the final prototype's competitive performance. The webservers, Apache and Nginx require network and filesystem I/O, and rely heavily on syscall performance. We see that Nginx, which is a higher-performance webserver, sees a larger overhead. IPC, which implements 128 byte transfers between two processes over a TCP connection, is almost entirely bound by kernel performance and sees a performance overhead of 3.4% on Midas.

Our prototype Midas kernel benefits significantly from exempting particular, proven-safe syscalls from instrumentation. While we exclude `write`-like syscalls from Midas because they are not vulnerable to double-fetch bugs, we also evaluated the performance cost of an unoptimized implementation (Midas+write) which also instruments these syscalls. To highlight the worst-case performance of the unoptimized implementation, we evaluate the performance of the IPC benchmark on Midas+write due to its high frequency of `write` syscalls. With Midas+write, the performance of the IPC benchmark falls to 12.6% of the baseline, a further degradation of 84% compared to Midas, showing that developer effort towards properly exempting frequently called *safe* syscalls from Midas protections is crucial towards for implementations to maintain competitive performance compared to the baseline.

Memory overhead. Our prototype incurs memory overhead due to metadata, tracking page snapshots and copies. At any instant, the memory overhead mainly depends on the number of executing syscalls (limited by the core count) and the number of page copies for these syscalls. On average, for every 1000 syscalls issued by the PTS benchmarks, our prototype created 236 snapshots (32B each) and 54 copies (4KB each). We can see that the occurrence of copies is low, resulting in negligible memory overhead.

7.3 Overhead breakdown

In this section, we explore the sources of Midas' overhead by analyzing `perf` traces for three workloads: thread creation from OSBench, linux compilation from PTS and Nginx. We aim to classify overheads into the various kernel function we instrumented: *i*) user copy in transfer function, *ii*) page duplication on page fault, *iii*) metadata cleanup on syscall

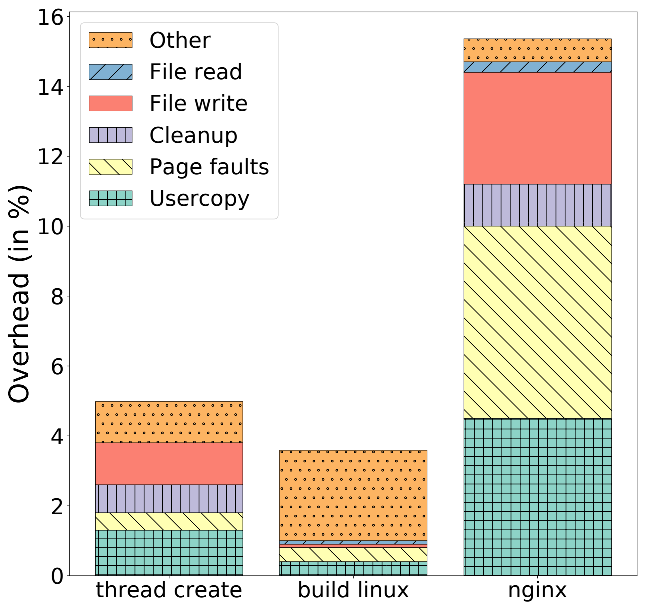

Figure 10: Classification of overheads for various benchmarks due to Midas.

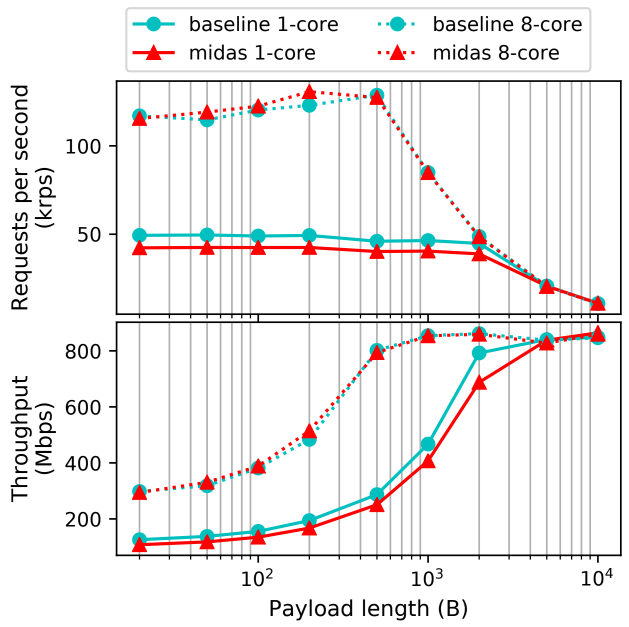

Figure 11: Request rates and throughput served by the Nginx server for static pages.

end, and *iv*) filesystem operations.

To estimate the time spent in each function, we create FlameGraphs for each workload [21] using samples of processor state, including the call stack, collected over 30 second periods by `perf record`. After identifying one binary for the workload from the FlameGraph, we estimate the overhead for a function as the difference in execution time attributed to that function between the baseline and Midas systems. The total overhead is estimated from the throughput figures obtained from Section 7.2.

Figure 10 shows the breakdown of overheads for three workloads. As expected, metadata tracking and duplication causes most overheads for the user copy and fault handling functions respectively. The results for the Linux build breakdown differs from the other workloads in the large portion of the unaccounted overhead (labelled "Other"). This anomaly stems for the fact that Linux's compilation runs a large number (1000s) of processes, of which the compiler accounts for less than 50% of the total execution time. The reported breakdown accounts for overheads on the compiler, but not all the other processes.

Both page faults and user copies cause state changes for a page, and thereby change the page's access permissions in the PTE. The resulting TLB flush accounts for 0.3%, 0% and 1.1% overhead for thread creation, compilation, and Nginx respectively. The load generator `bombardier` used for loading Nginx, however, sees a much larger overhead for TLB flushing, accounting for 21% of its execution time.

7.4 Case study: Nginx

To better understand Midas' behavior under varying syscall rates and different core configurations, we study Nginx's (version 1.18) throughput while varying payload sizes and different worker counts. Each worker is single threaded and uses one core. The server is loaded with requests from a separate machine running a load generater (`bombardier`) with 100 concurrent connections (chosen to maximize throughput) over a 1Gbps link. The clients send http requests for files ranging between 20B and 10000B.

In Figure 11, we plot the request rate and throughput for Nginx servers running with one and eight workers. For all configurations, we can see that the rate of requests served remains almost constant while increasing payload size until the network link reaches saturation. Under a saturated network, the request rates for Midas match that of the baseline system. With a single worker, Midas' overheads cause a consistent 13–14% overhead on the request rate for small packet sizes. However, we see that Midas has practically no overhead when serving requests with 8 workers even when packet sizes are too small to saturate the network link. In this case, both Midas and the baseline system are limited by the scalability of the Linux networking stack.

8 Related Work

Early work on double-fetch bugs relied primarily on manual code analysis to identify bugs in kernel code [38, 48] or in syscall wrappers [42]. Realizing the limited scalability of this approach, particularly when applied to large codebases such as the Linux kernel, subsequent work focussed on automated techniques based on static or dynamic analysis techniques, and on leveraging hardware features to mitigate such bugs.

Static analysis. Static analysis proposals use code analysis to find and fix double fetch vulnerabilities. DFTinker [29] improves the coverage of pattern matching rules for detecting double fetches in code as initially proposed by Wang et al. [39]. Deadline [46] and DFTracker [41] further generalize the analysis by leveraging symbolic execution.

However, static analysis is severely limited by its requirement for source code, which eliminates possibility of protecting of analyzing binary-only modules for which code is not available. In contrast, Midas can also protect such modules since well-behaving module use the kernel transfer functions to access user memory. Symbolic execution solves the generality problem of pattern-matching approaches but has its limitations (path explosion, function pointers, modelling numerous library functions, etc.). Deadline [46] specifically requires the additional assumption that pointer syscall argument do not alias, an assumption that can wilfully be violated by our adversarial model.

Additionally, the protection allowed by static analysis methods are limited: only the bugs which are detected can be fixed, and static analyses are necessarily incomplete. In contrast, Midas mitigates *all* TOCTTOU vulnerabilities. Further, specific cases of double fetches, such as in syscall wrappers cannot be fixed in code, and require a versioning system such as Midas in order to enable deep argument inspection.

Dynamic Analysis. Dynamic analysis techniques leverage runtime information and values to detect double fetches, and are notable in their ability to find bugs in binaries.

To enable the search for various classes of bugs, Google Project Zero's Bochspwn project [23] introduced a comprehensive emulator for `x86` with callbacks to allow tracing of kernel operations, including memory accesses. When paired with a syscall fuzzer, Bochspwn successfully detected and reported double fetches from these access logs, but suffered from a high rate of false positives. Another major shortcoming of Bochspwn was its low execution throughput of 40-80MIPS which limited its ability to explore code paths. Xenpwn [44] extended Bochspwn's trace-driven approach to fuzz hypervisors for double-fetch bugs. Xenpwn found three double fetches in the Xen hypervisor, but no critical vulnerabilities in KVM.

DECAF [34] inverts Midas' adversarial model, leveraging concurrent access to syscall parameters from userspace to detect kernel accesses via a cache side-channel. DECAF is strongly reliant on CPU-specific behavior, which is sensitive to CPU parameters, subject to changes from generation to generation (or even from core to core) and prone to noise and false sharing. Following the discovery of transient-execution attacks [25], proposals such as InvisiSpec [24,47] have tried to prevent information leakage via cache side-channels. Future generations might entirely close this channel, or introduce constraints that limits this information flow.

Dynamic analysis techniques can only detect vulnerabilities on executed code paths, and therefore typically rely on a fuzzer to extensively cover kernel code. However, fuzzers are inherently incomplete, limiting the ability of dynamic analysis to find bugs.

Mitigations. Previous attempts [29,34] to eliminate double fetch vulnerabilities rely on Intel TSX, a hardware transactional memory implementation, to detect malicious writes to data read by the kernel. A defense based on TSX improves upon Midas by reducing the scope for false sharing from a page size to a cache size. However, TSX suffers from major limitations which restrict its useability for general kernel implementations. Of note, TSX requires that the data working set for the critical section experiences no L1 cache evictions, even due to contention from a simultaneously-multithreaded (hyperthreading) core. Further, TSX is limited to processors from a specific manufacturer (Intel), leaving the vast majority of computing devices (mobile, IoT, AMD processors) unprotected.

9 Conclusion

Midas mitigates double-fetch bugs in system calls and protects the operating system kernel by enforcing the invariant: *through a syscall's lifetime, every read to a userspace object will return the same value*. Our Midas implementation creates on-demand snapshots and copies of pages that are read and merges any writes through the execution of the system call. Our mitigation protects the core kernel, as well as drivers by carefully instrumenting functions that interact with the process address space. While our implementation focuses on Linux for x86-64, our concept is generic and empowers other kernels to protect themselves against notoriously hard-to-find and easy-to-exploit double fetch bugs.

The performance evaluation of our prototype implementation is promising. Compute-bound benchmarks have negligible overhead and even syscall-intensive benchmarks exhibit low overhead. On one hand, Midas mitigates all double fetch bugs in the kernel and gives developers a tool to locate such bugs. On the other hand, Midas sets the foundation for efficient, stateful system call filtering and validation. We have released the source code of our prototype as open-source at `https://hexhive.epfl.ch/midas/`.

Acknowledgements

The authors thank Marcel Busch, Adrian Herrera, and the anonymous reviewers for their careful feedback and support during the writing of this paper. This project was supported by European Research Council (ERC) grant No. 850868, DARPA

HR001119S0089-AMP-FP-034, and ONR award N00014-18-1-2674. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

References

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

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

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

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

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

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

[7] Cve-2018-12633. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-12633.

[8] Cve-2019-20610. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-20610.

[9] Cve-2019-5519. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5519.

[10] Cve-2020-12652. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12652.

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

[12] SecComp. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[13] Seccomp and deep argument inspection. https://lwn.net/Articles/822256/.

[14] Cve-2018-12633 fix. https://github.com/torvalds/linux/commit/bd23a7269834dc7c1f93e83535d16ebc44b75eba, Aug 2020.

[15] Phoronix test suite. https://www.phoronix-test-suite.com/, 8 2020.

[16] Advanced Micro Devices (AMD). AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. https://www.amd.com/system/files/TechDocs/24594.pdf.

[17] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

[18] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12(4):271–307, 1994.

[19] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[20] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003.

[21] Brendan Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, 2016.

[22] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. Rebooting virtual memory with midgard. In *Proceedings of the 48th Annual International Symposium on Computer Architecture, ISCA 2021*.

[23] GC Mateusz Jurczyk and Gynvael Coldwind. Bochspwn: Identifying 0-days via system-wide memory access pattern analysis. *Black Hat USA Briefings (Black Hat USA)*, 2013.

[24] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*, page 60. ACM, 2019.

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. 2018.

[26] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[27] ARM Ltd. *ARM Architecture Reference Manual (ARMv8, for ARMv8-Aarchitecture profile)*. 2013.

[28] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. Untrusted hardware causes double-fetch problems in the I/O memory. *Journal of Computer Science and Technology*, 33(3):587–602, 2018.

[29] Yingqi Luo, Pengfei Wang, Xu Zhou, and Kai Lu. Dftinker: Detecting and fixing double-fetch bugs in an automated way. In Sriram Chellappan, Wei Cheng, and Wei Li, editors, *Wireless Algorithms, Systems, and Applications - 13th International Conference, WASA 2018, Tianjin, China, June 20-22, 2018, Proceedings*, volume 10874 of *Lecture Notes in Computer Science*, pages 780–785. Springer, 2018.

[30] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[31] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border control: sandboxing accelerators. In Milos Prvulovic, editor, *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, pages 470–481. ACM, 2015.

[32] Mathias Payer and Thomas R Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 215–226, 2012.

[33] Calton Pu and Jinpeng Wei. A Methodical Defense against TOCTTOU Attacks: The EDGI Approach. In *Proceedings of the 2006 International Symposium on Secure Software Engineering*, 2006.

[34] 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 Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 587–600. ACM, 2018.

[35] Fermin J. Serna. Ms08-061 : The case of the kernel mode double-fetch. https://msrc-blog.microsoft.com/2008/10/14/ms08-061-the-case-of-the-kernel-mode-double-fetch/, Oct 2008.

[36] Marc Snir, William Gropp, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: the MPI core*, volume 1. MIT press, 1998.

[37] Dan Tsafrir, Tomer Hertz, David A Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *FAST*, volume 8, pages 1–18, 2008.

[38] twiz and sgrakkyu. From ring 0 to uid 0. *CCC*, 2007.

[39] 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 *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1–16, 2017.

[40] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6):e4345, 2018.

[41] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science*, 13(2):247–263, 2019.

[42] Robert NM Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. *WOOT*, 7:1–8, 2007.

[43] Jinpeng Wei and Calton Pu. Modeling and preventing TOCTTOU vulnerabilities in Unix-style file systems. *computers & security*, 29(8):815–830, 2010.

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

[45] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, 2017.

[46] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 661–678. IEEE Computer Society, 2018.

[47] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, page 1076. ACM, 2019.

[48] Junfeng Yang, Ang Cui, Salvatore J. Stolfo, and Simha Sethumadhavan. Concurrency attacks. In Hans-Juergen Boehm and Luis Ceze, editors, *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012*. USENIX Association, 2012.

A Artifact Appendix

For Midas, we present an artifact including the source code and binaries for the prototype based on Linux, an exploit which demonstrates that Midas mitigates a real CVE, and benchmarks for evaluating Midas' performance, along with scripts to automate the evaluation. In the following sections, we describe the artifact, its requirements and how to run it, and what the expected results are. Visit the project website `https://hexhive.epfl.ch/midas` for more details.

A.1 Description

The primary artifact for this paper is the code implementing Midas on the Linux kernel (v5.11), available on GitHub. We also provide a disk image suitable for recreating experiments from this paper, containing the kernel as both source code and as compiled binaries. The disk image contains the CVE exploit used to test correctness in the paper, all benchmarks evaluated in the paper, and scripts to run these. This image allows recreation of all empirical evidence presented in the paper's evaluation. Finally, we provide further information on the project website including a detailed description of the artifact, its contents, how to run it, and expected outputs.

- Source code: `https://github.com/HexHive/midas`
- Disk image: `https://zenodo.org/record/5753026`
- Project website: `https://hexhive.epfl.ch/midas`

A.1.1 Hardware Dependencies

You can run the disk image within a QEMU virtual machine to test functionality. The host machine requires around 100GiB free disk space and at least 8GiB memory. To evaluate performance, the disk image must run on a real machine. Our Midas prototype supports machines with 64-bit x86 processors, and the results in the paper were obtained on a machine with an Intel i7-9700 CPU. Further, the real machine requires an empty 1TiB disk, and a EUFI-enabled motherboard. In both setups, a SSD is preferred for storage, as the system and kernel can be compiled faster. Evaluating the Nginx benchmark requires a second, networked machine to act as a load generator.

A.1.2 Software Dependencies

Running the Midas disk image requires a guest operating system which supports running QEMU/KVM. The image was tested on QEMU version `4.2.1` on a machine running Ubuntu 20.04 with Linux kernel version `5.4.0-88-generic`. Other virtualization software should also be supported but may require some tuning. Running the disk image on real hardware requires no special software support, apart from a tool to write the image to a disk. On Linux, we can use `dd`.

A.2 Installation

The installation procedure includes downloading and uncompressing the provided compressed disk image, then either running a VM directly from this image, or by writing the image to a disk and booting from it.

On Linux, the following command extracts the image.

```
  pv ae.img.xz | unxz -T <num threads> > ae.img
```

The uncompressed disk image can then either be run with QEMU, or written to a real disk. To run with QEMU, an example command is shown below.

```
qemu-system-x86_64                      \
  -m 4G                                 \
  -cpu host                             \
  -machine type=q35,accel=kvm           \
  -smp 4                                \
  -drive format=raw,file=ae.img         \
  -display default                      \
  -vga virtio                           \
  -show-cursor                          \
  -bios /usr/share/ovmf/OVMF.fd         \
  -net user,hostfwd=tcp::2222-:22       \
  -net nic
```

To run on real hardware, copy the image to a real disk using the command shown below, then install onto the machine and start it.

```
  dd if=ae.img of=/dev/<disk> bs=100M
```

A.3 Experiment Workflow

The experimental workflow compares the modified Midas kernel with the baseline Linux kernel. Detailed steps are available on the website at `https://hexhive.epfl.ch/midas/docs/ae.html`. You can validate the artifact by executing the following steps:

- Check that the code modifications described in the paper correspond to the code.
- Compile the code to re-create the kernel binary.
- Run a script to check that a CVE exploit is mitigated, as claimed in the paper.
- Run scripts to execute the benchmarks presented in the paper, to verify their reported performance.

For the CVE exploitation test, the `dmesg` output must be checked to ensure that Midas prevents exploitation. For the performance experiments, the results must be compiled and compared to get the Midas' relative performance. The general workflow is:

- boot with the correct kernel (baseline or Midas),
- run the script for the benchmark/CVE exploit,
- reboot with the other kernel, and
- run the same script again.

A.4 Expected Results

Midas is evaluated to demonstrate effective mitigation of double-fetch bugs with low overhead. The artifact enables you to verify this claim, that the prototype provides the claimed protection and that it performs as claimed. We demonstrate the first property by including checks in the kernel and running an exploit for CVE-2016-6516 to demonstrate its mitigation. The remaining benchmarks measure performance, either as operations per second or as time taken to finish each operation. Below, we describe how to interpret the outputs of running the exploit and benchmarks.

Midas protects the kernel against double-fetch bugs, and in particular mitigates an exploit for CVE-2016-6516. In our prototype, you will execute the exploit with and without Midas' protections. When run with the baseline kernel, the exploit is triggered, and the string `"Triggered bug: CVE-2016-6516!"` will be printed to `dmesg` output. With the Midas kernel, the string is never printed.

We also run kernel-intensive benchmarks which demonstrate that Midas has a low runtime overhead. Our artifact also contains the performance benchmarks used for testing Midas' performance. The benchmarks must be run separately with both the baseline and Midas kernel. We include a script to plot the relative performance vs. the baseline kernel. Midas' performance is strongly dependent on the CPU used for evaluation, and performance values can vary. However, we expect the trends of performance across benchmarks to roughly follow the following limits.

- Microbenchmarks see results in line with paper.
- NPB benchmarks experience 0-5% overhead, and should follow the numbers from the paper.
- PTS benchmarks - openssl, git, pybench, redis see an overhead <1%.
- PTS benchmarks - apache sees a overhead < 10-15%.
- PTS benchmarks - IPC benchmark sees overhead < 5%.
- Nginx shows a constant overhead as request size changes, until the network link is saturated.

The setup for breaking down Midas' overhead is complicated, and omitted from this artifact.

A.5 Artifact meta-information

- **Program:** NASA Parallel Benchmarks (NPB), Phoronix Test Suite (PTS), Nginx, the Linux kernel, and exploits for CVE-2016-6516. All benchmarks and code are publicly available, and are installed in the provided disk image.
- **Binaries:** The disk image provides the compiled Linux kernel (v5.11) with and without Midas' protections.
- **Hardware:** For functionality evaluation, one machine with $\tilde{1}$00GiB free disk space, and QEMU (version 4.2). For results reproduction, one machine with modern Intel x86 CPU, and a free 1TiB disk. In both setups, a SSD is preferred.
- **Run-time state:** The disk image includes a program for fixing CPU frequency, eliminating run-time variance. This only works on native hardware, not QEMU.
- **Metrics:** NPB workloads report execution rate. PTS workloads report either execution time or operation rate. Nginx reports both request rate and throughput.
- **Output:** Most benchmarks and tests output to a console.
- **Experiments:** Experiments have been prepared within the disk image, and can be run using provided scripts.
- **How much time is needed to prepare workflow (approximately)?:** 3-4 hours, on a machine with an SSD.
- **How much time is needed to complete experiments (approximately)?:** For performance evaluation, approx. 8 hours.
- **Publicly available?:** All code is publicly available.
- **Code license:** GPL v2.0
- **Archived (provide DOI or stable reference)?:** DOI 10.5281/zenodo.5753026 available at `https://zenodo.org/record/5753026`.