TikTok: Systematic Kernel TOCTTOU Protection

Anonymous Submission #94

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. We propose TikTok to mitigate double-fetch bugs. TikTok creates on-demand snapshots and copies of accessed data, enforcing our key invariant that throughout a syscall's lifetime, every read to a userspace object will return the same value.

TikTok shows no noticeable drop in performance when evaluated on compute-bound workloads. On system call heavy workloads, TikTok incurs 0.2-18% performance overhead, while protecting the kernel against any TOCTTOU attacks. On average, TikTok shows a 4.3% overhead on diverse workloads across two benchmark suites.

1 Introduction

The operating system (OS) kernel enables 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 and 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 that then trigger kernel bugs, elevating the privileges of the process. A common class of kernel bugs are so-called *double-fetch* bugs [28, 31, 32, 35]. 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 type of *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 kernel. An adversary may concurrently overwrite the length of the buffer to a larger number after the kernel allocated its buffer, causing the memory copy to overflow the buffer. Double-fetch bugs are a frequent problem in kernels and hypervisors [1–10]. Watson [33] blames an unfixable TOCTTOU constellation as a reason for the generic insecurity of *system call wrappers*. System call filtering wrappers require that data read from userspace for the initial check remains the same when the kernel later uses it for computation, and can currently only check arguments passed by value. TikTok enables "deep argument inspection" for SecComp [11, 12] i.e., checks on arguments passed by reference. Without TikTok, such inspection is inherently impossible since such checks introduce double fetches, and consequently TOCTTOU bugs.

To mitigate double-fetch bugs in the kernel, a system must prohibit *concurrent changes* to memory accessed by the system call. Adversaries may find thrifty ways to trigger such concurrent writes: *i*) direct writes from userspace (e.g., from concurrent threads), *ii*) kernel writes from system calls (e.g., from concurrent system calls), *iii*) modifying address space mappings, *iv*) concurrent `writes` to a file that alters mapped file pages, or *v*) storing arguments on device-backed pages, leveraging devices to trigger concurrent writes. To prevent attacks, all kinds of 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***. Based on this invariant we derive a *security property* that ensures that every read during the execution of a system call is tracked. Subsequent reads of the same address will always return the same value. For performance, multiple versions of an object may exist at the same time depending on when the system call was started and depending on how many concurrent system calls are in flight.

Orthogonal, we derive a *correctness property* that ensures the sharing of the correct version among the different system calls that are in flight. All writes end up on the most recent version of the objects and therefore allow forward progress. We implement our invariant in our TikTok prototype for the Linux kernel, but the defense applies to any modern operating system kernel.

Our evaluation shows overall low performance overhead for our mitigation. On parallel workloads from the NAS Parallel Benchmarks suite, TikTok shows an average performance overhead of 3.7% while its performance overhead on more kernel-intensive workloads from the Phoronix Test Suite is 5% with negligible memory overhead. The security evaluation demonstrates how TikTok successfully stops all attacks against vulnerable system calls, along with providing the developer with sufficient information about the location of the bug. The main contributions of this paper are:

- Distillation of TOCTTOU attack vectors into a core invariant that protects the kernel against malicious concurrent modifications;
- TikTok, 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 system call interposition and validation;
- An efficient implementation of TikTok for the Linux kernel that exhibits low (3.7%) performance overhead.

2 Background

TikTok 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. TikTok protects these pages by temporarily marking them read-only in the page tables. This section provides the background information necessary to reason about why and how TikTok 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. Some architectures also include segmentation-based protection working in tandem with page tables, but segmentation is irrelevant for TikTok. 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 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 *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 a 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, such as 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, and mapped pages have separate read/write/execute permissions. Generally, programs have write-execute exclusion which means that 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 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 top half of the address space holds kernel mappings, and the PTE entries for such pages have the U/S bit set. The kernel mappings are identical for all address spaces, and are kept consistent across the corresponding page tables. The bottom half of the address space is used for userspace mappings, and the PTE entries 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 as that of virtual memory (rooted at `/`). However, when files are read

or written, parts of the file are cached in the kernel's page cache consisting of pages mapped in the kernel's address space. Further, programs can explicitly map pages from a file, in which case the corresponding pages from the page cache are also mapped in 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, and come with the risk of the kernel confusing data structures stored in userspace memory for actual kernel data structures. A class of attacks can exploit this behavior via bugs in the kernel. Essentially, the adversary 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 to execute this code.

Architectures and OSs have mitigated this class of vulnerabilities by introducing Supervisor Memory Protection. Essentially, 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, and bits in the CR4 register are used to enable/disable the features. The architecture also exposes privileged instructions `stac`/`clac` to quickly disable and enable this access control. On the OS side, all accesses to userspace memory are made explicit, using special functions to read from and write to userspace memory. Any unintended access, outside these functions, causes a hardware fault, indicating either a kernel bug or an attack. Linux implements the functions `copy_{from/to}_user` which use the access control instructions to disable SMAP before accessing userspace data, and then re-enabling SMAP afterwards. Kernel accesses to userspace data therefore become explicit, allowing TikTok to reliably track all kernel fetches from userspace memory, and therefore protect them.

2.4 Double-Fetch Bugs

Double-fetch bugs occur when a privileged environment (such as the kernel) reads untrusted memory two or more times, and the read values are not identical. An example of such a situation is depicted in Figure 1. In between two reads by the target thread, the value of `X` in memory is changed by an adversary. The bug is a race condition since it requires accesses to memory in a particular order across threads. The situation where the first fetch validates an object's value in memory and the second fetch uses the same object's value is called a *Time-of-check to time-of-use (TOCTTOU)* bug. TOCTTOU

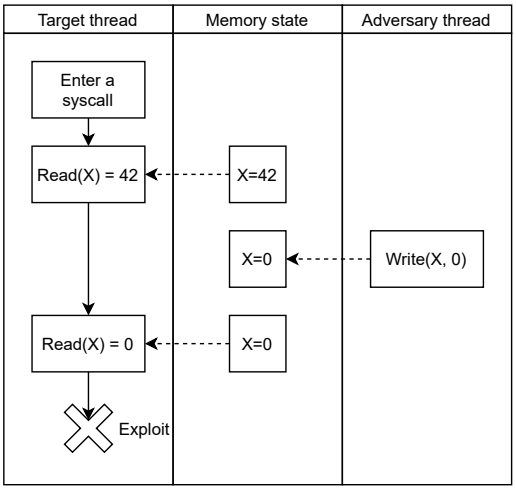

Figure 1: Example of a double-fetch bug.

bugs have been widely studied in file systems, where the API makes it possible to swap the file after validating the access rights [26, 27, 30, 34]. TOCTTOU bugs affect kernel code [20, 32] as well as dynamically loaded driver code [7, 13]. Wang et al. [32] showed that double-fetches appear not only in kernels, but wherever there is a trust boundary to cross (e.g., kernel — hypervisor [35] and hardware—kernel boundaries [23]).

3 Threat Model

The adversary has access to a user account on the target machine. They can execute arbitrary userspace code, including system calls. Some of the system calls have double-fetch vulnerabilities, and the adversary wants to exploit them, e.g., for privilege escalation. The adversary may execute arbitrary sequences of system calls on multiple CPU cores concurrently.

TikTok 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 [24] or side-channels [21], and file-system TOCTTOU attacks [26, 27, 30, 34] are out of scope.

4 Attack Classification

TikTok guards data processed during a syscall's execution against concurrent modification. We denote 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 existed at the time of the first read (see Table 1). 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, or FPGA) is responsible for modifying vulnerable data.

	Userspace	Kernel	Device
Existing mapping	Intra AS	User mapping	Device page
	Cross AS	Kernel mapping	Device DMA page
New mapping	mmap	mm_populate	New device page
	clone	–	New device DMA
	swap	–	New device DMA

Table 1: Attack vector classification for TOCTTOU exploits.

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, whether that mapping is in the same or in a different address space. Such attacks are called *direct double fetch* attacks [33]. Alternatively, a concurrently executing syscall can also modify the vulnerable data in a confused-deputy attack. When the adversary 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` system call 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 adversarial write uses a userspace mapping. *A protection mechanism must, therefore, 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. The adversary maps a file-backed page from the page-cache in a userspace process and then passed as an argument in this page to the targeted syscall. The adversary then triggers a concurrent `write` syscall to modify the vulnerable data using kernel mappings for the page-cache pages [33]. 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, therefore, 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 protection mechanisms which instrument the first read to protect the accessed page. An adversary can call `mmap` and `clone` syscalls to create a new mapping to the vulnerable data before writing to it. The first version is called a *reflected double fetch* attack [33]. The mapping might not be created at the time of the adversarial syscall, but lazily when the attacker writes to the vulnerable data. In a more involved variant, the adversary 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` for Linux) is creating the new mapping. *A protection mechanism must, therefore, instrument `mm_populate` and the code of any other syscall which may create new mappings.*

A new mapping might also be created due to swapping. If the adversary 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 adversary's mapping to the page. *The swapping mechanism must, therefore, be protected.*

TikTok protects against all of the aforementioned attack vectors. In the absence of any other syscall which can create new userspace mappings to vulnerable data, TikTok's 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 to the page, or if the page is memory-mapped and is actually backed by the device. In the latter case, external factors can change the vulnerable data. Existing discretionary access control rules generally bar users except a superuser from mapping device-backed pages into their address spaces. Such users are also disallowed from configuring DMA devices. Therefore, device modifications to vulnerable data fall outside our threat model and are not protected by TikTok. As a superuser can modify kernel code, protecting against attacks from the superuser is outside of our threat model. However, on processors supporting IOMMUs, TikTok can be extended to protect against modifications by DMA devices.

5 TikTok Design

TikTok 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*. TikTok 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.

TikTok's 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 efficiency, TikTok 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. As a side-effect of its implementation, TikTok does not distinguish accesses to different parts of a page, and may incur performance overhead due to false sharing within a page. Page-granularity protections are more conservative compared to byte-granularity protection and, therefore, TikTok maintains its invariant nonetheless but may incur performance overhead for false sharing on highly contended pages.

For an object spanning multiple pages, TikTok's design sequentially protects each page before reading from it. The leading pages containing the object are protected before the later pages, allowing an adversary to potentially modify the later pages before the syscall first reads them. However, the adversary 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. TikTok's invariant therefore prevent exploitation of double-fetch vulnerabilities even when the fetched objects span multiple pages.

A major requirement for TikTok 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 system call 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.

TikTok must maintain multiple versions of a page read by a syscall to maintain its invariant in the face of concurrent writes. TikTok 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 TikTok's space overheads and performance overheads for creating copies. TikTok 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. TikTok's design draws parallels to multi-version concurrency control methods for databases based on snapshot isolation [25]. 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, TikTok 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, TikTok 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 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-TikTok systems. Knowing which state the page is in allows TikTok to differentiate faults due to protecting pages from faults due to programs actually writing to an originally read-only page. 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

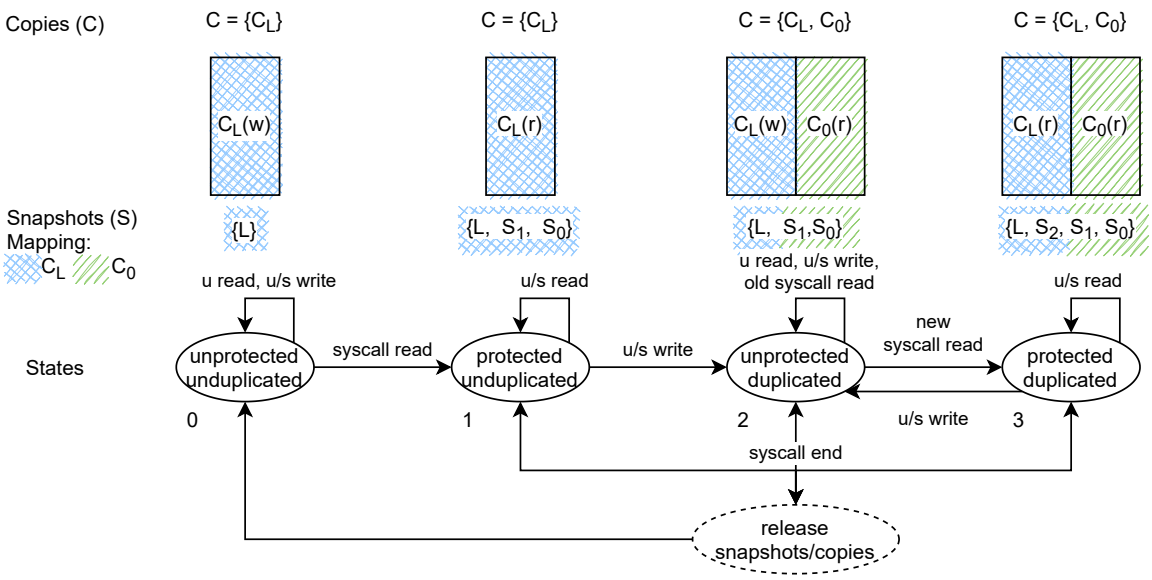

Figure 2: State diagram for a page in TikTok. Reads/writes from userspace/syscall code are marked (u)/(s) respectively. Shading is used to represent the mapping from snapshots to copies.

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 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. TikTok 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 TikTok 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. TikTok 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, at 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. TikTok's 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, TikTok 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 adversary 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 adversary 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. TikTok uses snapshots to enable a

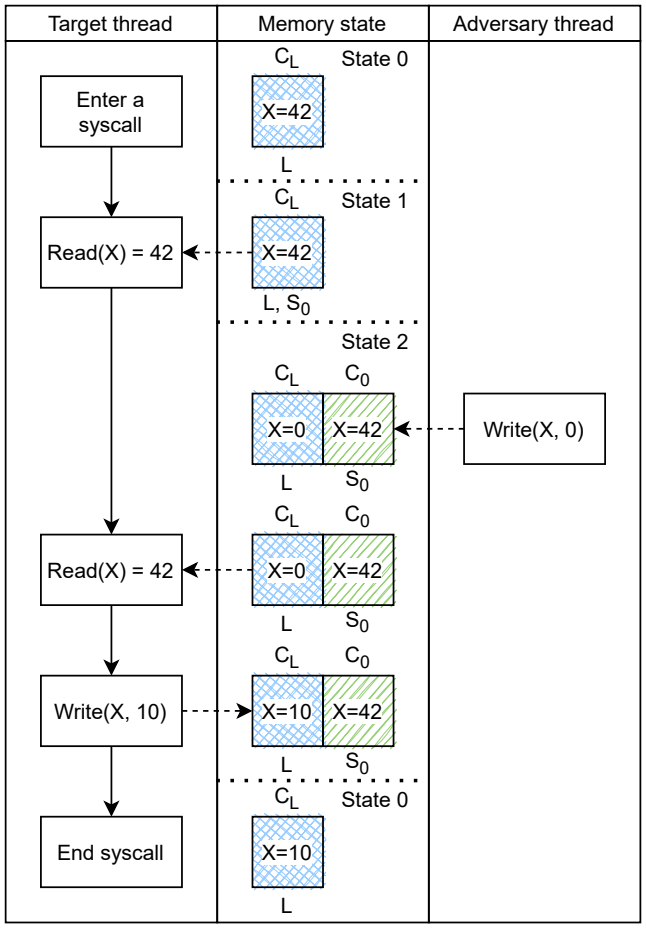

Figure 3: Diagram illustrating TikTok preventing exploitation of a double fetch.

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, TikTok cannot free the copy since userspace is using it. In 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, TikTok 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, TikTok 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 TikTok 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 TikTok 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.

System Call	Exemption reason
`futex`	Relies on concurrent write
`poll`	Relies on concurrent write
`ppoll`	Relies on concurrent write
`select`	Relies on concurrent write
`pselect6`	Relies on concurrent write
`rt_sigtimedwait`	Relies on concurrent write
`execve`	Remaps address space

Table 2: System calls uninstrumented by TikTok.

5.2 Discussion

Correctness of syscalls directly updating snapshot L. TikTok's 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 TikTok 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 TikTok syscall has a corresponding execution without TikTok (which we assume to be correct) leading to the same contents, we can conclude that the execution of the TikTok 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.

Let us consider a page holding a single-byte object O_0, and the sequence of operations to this byte during a TikTok 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, TikTok 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-TikTok 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.

Let us now assume that there was an attempt to exploit a TOCTTOU bug: a write Op_1 exists between two syscall reads Op_0 and Op_2. TikTok 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-TikTok system by moving the adversary'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 TikTok 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 in case 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 TikTok system is the same as that of the non-TikTok system. In general, any trace of operations on a TikTok system can be translated to a valid, safe trace on a non-TikTok system by moving adversarial writes to an object to just after the last double fetch of that object. Multiple syscalls in TikTok can therefore write to the same object without affecting correctness, because an equivalent, valid, safe non-TikTok trace exists where all of the writes have been postponed, in the same order to after the double fetch reads.

Exemptions. Certain syscalls such as `futex` rely on user data changing between double fetches to implement their functionality and cannot be protected by TikTok. 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 TikTok's invariant will prevent it from ever waking up the waiting task. Such syscalls cannot be protected by TikTok, and we implement an exemption list to prevent transitions in the state machines of pages read by these syscalls. The code for these syscalls must be manually inspected for double-fetch vulnerabilities. Crucially, exempting these syscalls from TikTok's protection does not affect the security of other syscalls. Any writes from these syscalls are subject to the same rules described in the state machine, and cannot break TikTok's invariant.

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

Syscalls with false sharing. Another hypothetical type of syscall could struggle with TikTok's 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 TikTok's invariant being enforced at page-granularity and false-sharing of the page between these objects, TikTok 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 which requires the value of O_1 to change between these two points in time would, therefore, not work with TikTok 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 TikTok's invariant. During extensive tests, we did not find any syscall which exhibits this behavior in the Linux kernel.

Preventing deadlocks by design. TikTok's 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. TikTok's 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 prior existence of a snapshot means that the syscall had previously read from this page and had then created this snapshot, implying a double fetch. This approach, however, is prone to false positives due to false sharing. The two reads might read from the same page, but access entirely disjoint bytes. So far, TikTok reports double fetches at page granularity. A precise sanitizer could maintain a bitmask of accessed bytes to prune false positives.

6 TikTok Implementation

Our TikTok prototype implements the state machine described in section 5 on Linux version 5.11, targeting the x86-64 architecture. A page protected by TikTok 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). TikTok can be implemented on any operating system kernel that uses a defined interface for reading from userspace and on any architecture which implements hardware-controlled access control to memory through page tables. The first requirement enables TikTok 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 faults, directing execution on the processor to a predefined exception handler in the OS. Our prototype instruments Linux' fault handler in the function `handle_pte_fault`

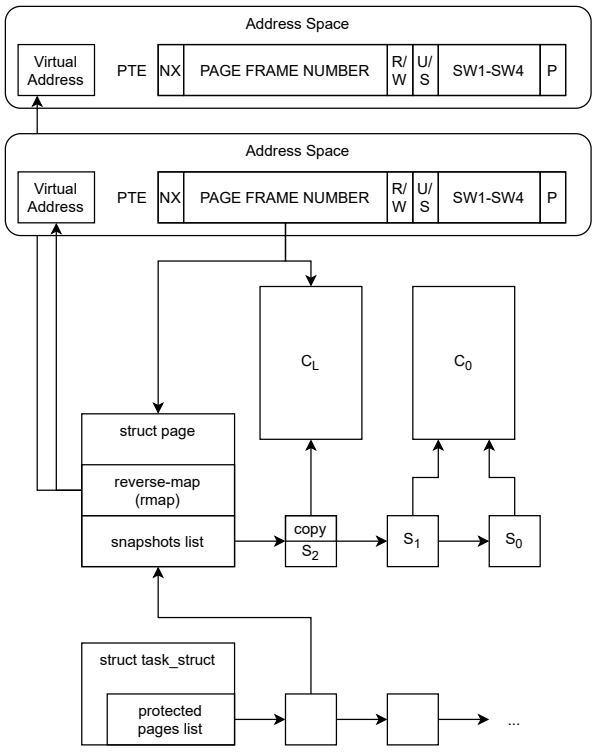

Figure 4: Bookkeeping information for a page.

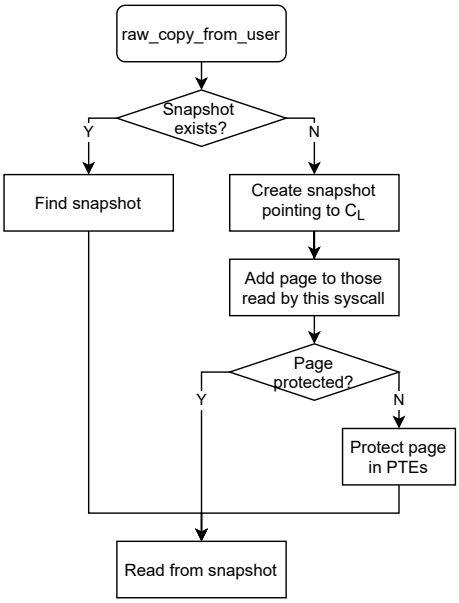

Figure 5: Flowchart for syscall reading from userspace.

to implement the write-triggered transmissions from states 2 and 4.

6.1 Tracking Page State

TikTok needs to track the state for every userspace page, including its snapshot and copy sets. Figure 4 shows the data structures used to track this 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 use one software-controlled bit (SW3) in the pagetable entries to track the protection status of the page, and another (SW2)[1] 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 TikTok's 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 TikTok 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 the protection state of pages requires PTE updates for the page 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.

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 5 shows the flowchart of the steps im-

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

plemented 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.

Exemptions. Our prototype TikTok kernel exempts a couple of functions from TikTok's invariant, and they are therefore not instrumented to follow the aforementioned steps while accessing userspace memory. `raw_copy_from_user_inatomic` is used by the kernel to read user memory in special situations such as a kernel oops[2] where the kernel reads user memory to provide a backtrace. In this severe situation, the goal of the kernel is to collect debug information before its imminent termination and no TOCTTOU protection is needed. In our prototype, we also exempt the `write` system call's read from user memory from instrumentation. The `write` system call 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 TikTok 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 adversary 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.

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. TikTok 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 6 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 has to 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.

[2] 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.

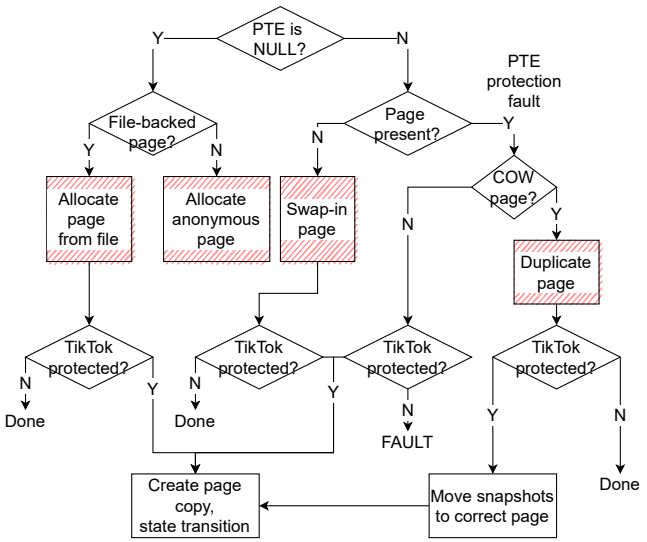

Figure 6: Flowchart for handling a page fault. Shaded operations are unmodified.

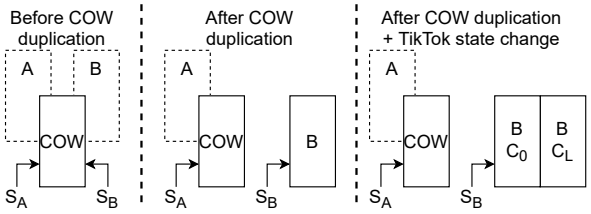

Figure 7: Flowchart for handling a page fault to a COW page.

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, TikTok then checks if the page was previously swapped in by any other task and is now in a protected state. For protected pages, TikTok implements the required state change based on whether the faullting access was a read or a write.

In the remaining case, faults for a present page indicate a permission fault (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 TikTok is illustrated in Figure 7 where logically separate pages A and B are actually mapped to the COW page. A COW page cannot be in states 2 or 3, since they cannot have multiple TikTok copies. COW pages in state 0 can be dealt with by the kernel's standard duplication method (not TikTok's 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 7, 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 7), TikTok moves the snapshots for the faulting process (S_B) to the new page. Here, TikTok also updates the protected page list in the affected syscalls' `task_structs` so that they point 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.

6.4 Syscall Completion

On syscall completion, TikTok cleans up snapshots allocated for the syscall by instrumenting the end of `do_syscall_64`. TikTok 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

TikTok 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. When writing to pages in the page cache, the FS driver will access the page using kernel mappings. Since TikTok 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). [3] 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 module implementing a different, insecure FS driver which does not implement TikTok checks. A malicious superuser is, however, outside our threat model.

[3] 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.

6.6 New Mappings to Protected Pages

Our TikTok prototype preserves the state machine for user pages across operations which create new mappings to a page 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 TikTok 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 and account for the majority of TikTok overhead.

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 [15] lists such an instruction (`INVLPGB`), though it is yet to be implemented in any commercially available x86 processors. The ARM v8-A architecture manual [22] lists similar instructions `TLBI ASIDE1IS` and `TLBI ASIDE1OS` which invalidates all entries of a page within a cluster of cores but not for cores in other clusters (called an Inner Shareable Domain) and cores across clusters (called an Outer Shareable Domain) respectively.

Alternate architectures [17, 19] with a single, system-wide translation table would also benefit TikTok by having a single page table to update instead of multiple page tables for each address space a page is mapped in.

7 Evaluation

In this section, we quantify TikTok's overhead on workloads with different characteristics, both compute-bound applications which rarely use syscalls and syscall-heavy applications which heavily rely on the kernel's interface. TikTok's overhead depends on the number of address spaces where protected pages are mapped. Relevant benchmarks where we expect overhead therefore include multiprocessing, parallel benchmarks.

We evaluate TikTok on two benchmark suites: the NAS Parallel Benchmark (NPB) [16] and select workloads from the Phoronix Test Suite (PTS) [14]. NPB includes compute-intensive multiprocessing workloads with a low, but non-negligible syscall rate. NPB therefore demonstrates the ability of TikTok 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. 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. SPEC CPU2017 benchmarks would unfairly bias performance in favor of TikTok.

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 *TikTok* configuration runs our prototype TikTok kernel also based on kernel v5.11. For particular benchmarks, we also run the *TikTok+write* configuration which also runs our prototype TikTok kernel but instruments all syscalls including `write`.

7.1 NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB) [16] 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 [18] 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 [29] 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 TikTok's protection. Additional global TLB shootdowns (for snapshot synchronization) added by TikTok will also affect the performance of such parallel benchmarks.

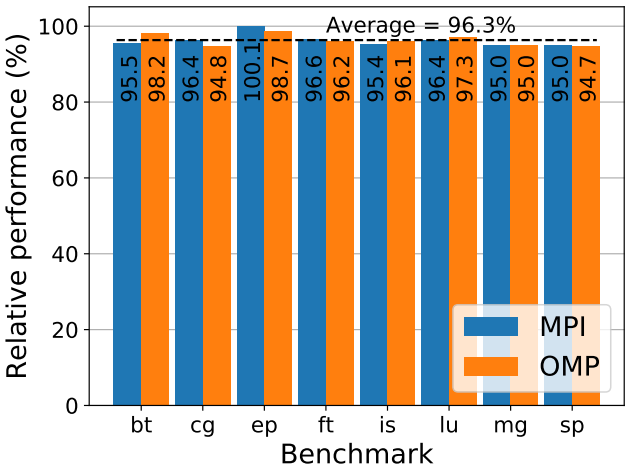

Figure 8: TikTok performance on NPB benchmarks relative to the baseline system with the same parallelization framework.

We evaluated NPB benchmarks of class A on our testbench, running 4 threads/processes in parallel. These 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 of them to run without time-sharing. Figure 8 shows TikTok's performance for both MPI and OpenMP, normalized to the performance of the baseline system with the same parallelization framework. On average, TikTok achieved 96.3% of the baseline system's performance on both frameworks. TikTok's performance for the `ep` (Embarrassingly Parallel) benchmark is closest to that of the baseline, since it has low communication overheads. TikTok shows low overhead (3.7%) for compute-intensive, parallel workloads.

7.2 Phoronix Test Suite

The Phoronix Test Suite (PTS) [14] 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 biased the selection to benchmarks that require (heavy) kernel activity to test the overhead of TikTok's 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-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

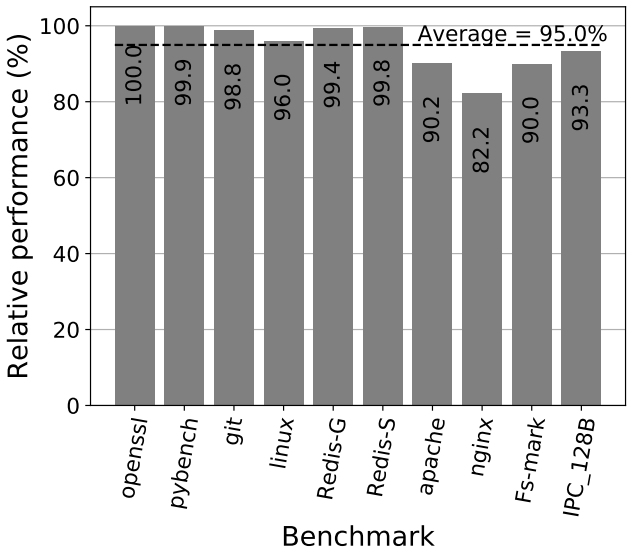

Figure 9: TikTok performance on PTS benchmarks relative to the baseline system.

time in syscalls and whose performance is entirely dependent on kernel IPC performance.

We plot TikTok's 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. TikTok experiences a small, but non-negligible overhead of 4% on this workload. Redis requires syscalls for receiving and replying to requests, but processes its transaction entirely in-memory. While early prototypes of our kernel caused significant degradation of Redis' throughput (up to 69%), demonstrating Redis' dependence on kernel performance, our evaluation prototype achieves almost identical results as the baseline, highlighting the final prototype's competitive performance. The webservers, Apache and Nginx require network and file-system I/O, and rely heavily on syscall performance. We see that Nginx, which is a higher-performance webserver, sees a larger overhead. Fs-mark, which accesses a file system with 5000 files of 1MB concurrently from 4 threads, and IPC, which implements 128 byte transfers between two processes over a TCP connection, are almost entirely bound by kernel performance. These benchmarks see a performance overhead of up to 10% on TikTok.

Our prototype TikTok kernel benefits significantly from exempting particular, proven-safe syscalls from instrumentation. While we exclude `write`-like syscalls from TikTok because they are not vulnerable to double-fetch bugs, we also evaluated the performance cost of an unoptimized implementation (TikTok+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 TikTok+write due to its high frequency of `write` syscalls. With TikTok+write, the performance of the IPC benchmark falls to 12.6% of the baseline, a further degradation of 81% compared to TikTok, showing that developer effort towards properly exempting frequently called *safe* syscalls from TikTok protections is crucial towards for implementations to maintain competitive performance compared to the baseline.

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.

8 Conclusion

TikTok mitigates double-fetch bugs in system calls and protects the operating system kernel by enforcing a core invariant: *through a syscall's lifetime, every read to a userspace object will return the same value*. Our TikTok 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, TikTok mitigates all double fetch bugs in the kernel and gives developers a tool to locate such bugs. On the other hand, TikTok sets the foundation for efficient, stateful system call filtering and validation. We will release the source code of our prototype as open-source and continue our interaction with the Linux kernel community.

References

[1] Cve-2013-1332. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1332. dxgkrnl.sys (aka the DirectX graphics kernel subsystem) in the kernel-mode drivers in Microsoft Windows Vista SP2, Windows Server 2008 SP2 and R2 SP1, Windows 7 SP1, Windows 8, Windows Server 2012, and Windows RT does not properly handle objects in memory, which allows local users to gain privileges via a crafted application, aka "DirectX Graphics Kernel Subsystem Double Fetch Vulnerability.".

[2] Cve-2015-8550. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8550. Xen, when used on a system providing PV backends, allows local guest OS administrators to cause a denial of service (host OS crash) or gain privileges by writing to memory shared between the frontend and backend, aka a double fetch vulnerability.

[3] Cve-2016-10433. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10433. In Android before 2018-04-05 or earlier security patch level on Qualcomm Snapdragon Automobile, Snapdragon Mobile, and Snapdragon Wear MDM9635M, MDM9640, MDM9645, MSM8909W, SD 210/SD 212/SD 205, SD 400, SD 410/12, SD 425, SD 430, SD 450, SD 615/16/SD 415, SD 617, SD 625, SD 650/52, SD 800, SD 808, SD 820, and SD 820A, TOCTOU vulnerability during SSD image decryption may cause memory corruption.

[4] Cve-2016-10435. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10435. In Android before 2018-04-05 or earlier security patch level on Qualcomm Snapdragon Automobile, Snapdragon Mobile, and Snapdragon Wear MDM9206, MDM9625, MDM9635M, MDM9640, MDM9645, MSM8909W, SD 210/SD 212/SD 205, SD 400, SD 410/12, SD 425, SD 430, SD 450, SD 615/16/SD 415, SD 617, SD 625, SD 650/52, SD 800, SD 808, SD 820, and SD 820A, in some QTEE syscall handlers, a TOCTOU vulnerability exists.

[5] Cve-2016-10439. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10439. In Android before 2018-04-05 or earlier security patch level on Qualcomm Snapdragon Automobile and Snapdragon Mobile SD 425, SD 430, SD 450, SD 625, SD 650/52, SD 820, and SD 820A, there is a TOCTOU vulnerability in the input validation for bulletin_board_read syscall. A pointer dereference is being validated without promising the pointer hasn't been changed by the HLOS program.

[6] Cve-2016-8438. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8438. Integer overflow leading to a TOCTOU condition in hypervisor PIL. An integer overflow exposes a race condition that may be used to bypass (Peripheral Image Loader) PIL authentication. Product: Android. Versions: Kernel 3.18. Android ID: A-31624565. References: QC-CR#1023638.

[7] Cve-2018-12633. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2018-12633. CVE-2018-12633 kernel: Double-fetch vulnerability in drivers/virt/vboxguest/vboxguest_linux.c:vbg_misc_device_ioctl() allows information leak and local denial of service.

[8] Cve-2019-20610. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-20610. An issue was discovered on Samsung mobile devices with N(7.X) and O(8.X) (Exynos 7570, 7870, 7880, 7885, 8890, 8895, and 9810 chipsets) software. A double-fetch vulnerability in Trustlet allows arbitrary TEE code execution. The Samsung ID is SVE-2019-13910 (April 2019).

[9] Cve-2019-5519. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5519. In Android before 2018-04-05 or earlier security patch level on Qualcomm Snapdragon Automobile, Snapdragon Mobile, and Snapdragon Wear MDM9635M, MDM9640, MDM9645, MSM8909W, SD 210/SD 212/SD 205, SD 400, SD 410/12, SD 425, SD 430, SD 450, SD 615/16/SD 415, SD 617, SD 625, SD 650/52, SD 800, SD 808, SD 820, and SD 820A, TOCTOU vulnerability during SSD image decryption may cause memory corruption.

[10] Cve-2020-12652. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-12652. The __mptctl_ioctl function in drivers/message/fusion/mptctl.c in the Linux kernel before 5.4.14 allows local users to hold an incorrect lock during the ioctl operation and trigger a race condition, i.e., a "double fetch" vulnerability.

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

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

[13] Cve-2018-12633 fix. https://github.com/torvalds/linux/commit/bd23a7269834dc7c1f93e83535d16ebc44b75eba, 8 2020.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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*.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689. ACM, 2015.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

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

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