TikTok: Kernel TOCTTOU Protection

Anonymous submission #303 to Security & Privacy '21

Abstract—**Double-fetch bugs are a plague in all major operating system kernels. They occur when data is fetched twice across the trust-boundary without checking that it has not changed. Such bugs may force the kernel into an inconsistent state, allowing an attacker to illegally access memory, cause denial of service, or to escalate their privileges.**

So far, the only protection against double-fetch bugs is to detect and fix them. However, they exhibit illegal behavior only under specific conditions, making them incredibly hard to find. Even worse, the bug may be in unchangeable code (e.g., a binary driver blob). We propose TikTok to mitigate double-fetch bugs. TikTok leverages on-the-fly page-table hardening to prevent changes to the system call arguments, while the system call executes.

TikTok shows no noticeable drop in performance when evaluated on CPU-bound workloads. On extremely system call heavy workloads, TikTok shows 2-6% overhead when protecting select system calls, and 17% when all system calls are guarded.

Index Terms—**Double-fetch bugs, TOCTTOU, kernel security, page-tables, mitigation**

I. Introduction

Modern systems consist of *untrusted* user-space processes and the *trusted* kernel. All data that crosses this trust boundary must be checked. Double-fetch bugs [27], [32], [38], [34] occur when higher-privileged code (e.g., the kernel) reads the same data twice from a lower-privileged address space (e.g., user-space). As attackers may change the data between the two reads, this enables a plethora of attack vectors. 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 the operand while the second read is used to modify state. Double-fetch bugs are a frequent problem in kernels and hypervisors [7], [10], [1], [8], [2], [5], [4], [3], [9], [6]. Considering that double-fetches also appear in drivers, legacy systems with binary-only drivers cannot be patched, even if a bug is found. Furthermore, Watson [36] blames an unfixable TOCTTOU bug as a reason for the insecurity of *system call wrappers*.

This state of affairs calls for a solution that *mitigates* double-fetches in system calls to protect the kernel. To mitigate double-fetch bugs, a system must prohibit changes from concurrent threads to memory accessed by the system call. The comprehensive system that protects against the modification of arguments must consider all possible approaches to change the arguments. Possible attack vectors are: **(i)** direct writes from user-space, **(ii)** kernel writes from system calls, **(iii)** remapping the pages with different permissions, **(iv)** `write` calls to a file that alter mapped file pages, or **(v)** storing arguments on device-backed pages. Simply preventing direct writes from user-space is insufficient to stop these attacks. A mitigation must monitor redundant mappings, system calls, and write calls to files.

Our proposed mitigation *marks* user-space pages touched by kernel code as *read-only* to prohibit the first two attacks (direct writes from concurrent user-space threads or concurrent system calls). Whenever a page is linked into an address space, its security properties are checked to mitigate the third attack (remapping). If a file is mapped to memory then write system calls are paused if the target page is marked to mitigate the fourth attack (write buffers). We prevent the fifth attack through through *Discretionary Access Control (DAC)* for device-backed pages. Our mitigation relies on a well-defined interface for *reading* (`read_in`) from user-space and on the paging infrastructure to protect marked pages. Threads that write to a marked page are paused in the *page-fault handler* and continue after the page is unmarked. The kernel interface for *writing* (`write_out`) to user-space is extended to execute all writes to marked pages at the end of the call. This prevents the system calls from writing to the pages they have marked. A small set of system calls (e.g., `pollfd`, `futex`, or `sys_nanosleep`) depend on "double-fetch-like" behavior and need to be manually verified and whitelisted.

We implement our mitigation, TikTok, as a memory marking extension to the Linux kernel that does not require any changes to user programs. TikTok provides system calls with a memory snapshot of when pages have first been read during that system call. TikTok groups all writes that modify protected memory and batches them at the end of the system call. By mitigating double-fetches, TikTok protects against the double-fetch bugs in all kernel code. Furthermore, TikTok renders previously unavoidable double-fetches (such as a TOCTTOU in system call wrappers) completely benign.

Our evaluation shows that extremely system call heavy multithreaded programs, such as Apache and NginX, suffer an 17% drop in performance with TikTok protecting all possible system calls. Whitelisting high frequency system calls drops the overhead to 2-6%. CPU-bound programs show negligible performance loss, even when they use multiple threads (OMP [16]) or multi-process message-passing (MPI [30]).

The main contributions of this paper are:

- A *confused deputy attack* on memory-marking protection mechanisms that mark the pages with a *superuser* bit,
- A solution to the problem of protecting the arguments of system calls that write to their arguments,
- A comprehensive technique for temporarily preventing memory from being changed, by safely postponing updates from both user-space and kernel, and
- TikTok, a mitigation for the double-fetch and TOCTTOU attacks on system call arguments in the Linux kernel.

II. Background

TikTok orchestrates several Linux subsystems to provide its protection. It uses *page-tables* to mark *shared memory* storing the system call arguments as *read-only*. Different system calls have different relationships with shared memory, and interact with TikTok differently in practice. This section provides the background information necessary to reason why and how TikTok protects the arguments from change while enabling the threads to execute correctly.

Section II-A introduces two different ways of communication between processes - *shared memory* and *message-passing*. TikTok affects both of these methods because the arguments of the message-passing system calls are stored in (potentially shared) memory. The waits caused by writing to marked pages can interact with already present synchronization primitives and cause deadlocks. In Section IV-C we explain why they do not appear in practice.

Section II-B covers the organization of *virtual memory*, *page-tables* and the interface the Linux kernel uses to access the user-space memory. This section is essential to understanding the implementation of TikTok.

Section II-D details the *double-fetch* bug that TikTok is mitigating. It explains its cause and the different types of double-fetch bugs.

A. Interprocess Communication

The two methods of inter-process communication are *shared memory* and *message passing* [28].

Shared memory relies on two processes having a section of memory that both can access. Processes communicate by reading and writing to the shared memory. Message passing consists of one process calling `send`, and another one calling `receive` to fetch the message. Synchronous message passing blocks execution until the calls have finished.

Modern operating systems support both of these approaches. TikTok interferes with message-passing system calls by linking them to shared memory. Storing arguments for blocking message-passing system calls in shared memory can cause deadlocks if those arguments are written to (see Section IV-C).

B. Linux Memory Subsystem

Linux implements *paged virtual memory* and uses *virtual addresses* which map to *physical addresses* in RAM. The mapping function is defined for each process by a multi-level *page-table*.

The leaves of the page-table store the access information about the corresponding page:

Present bit (P) is set if the page is present in memory

Read/Write bit (R/W) denotes if the page is writable or just readable

User/Superuser bit (U/S) represents if the page can be accessed by the user, or only by the superuser

Not Executable bit (NX) is set if the code stored on the page cannot be executed

Page Frame Number denotes the page frame the entry points to

SW1-SW4 Four bits free for the OS to use

Considering that the page-table traversal is frequent, it is implemented in hardware by the *memory management unit* (MMU). Reading the page-table from memory is slow, so a small cache—*translation-lookaside buffer* (TLB) — is added to the MMU to store frequently accessed page entries. Kernel developers can *flush* (clear) certain TLB ranges in software.

On invalid access (e.g., wrong permissions, page not present) the MMU will trigger a page-fault. The page-fault handler executes in the kernel context of the faulting thread and performs the appropriate action (e.g. load a page, kill the thread). With the advent of cloud computing, *user-space page-fault handling* has been added to the Linux kernel. TikTok relies on the page-fault handler for protection, so user-space page-fault handling must be disabled.

Memory in Linux can be either *file-backed* or *anonymous*. File-backed pages have a backing file where their data is stored. Anonymous pages do not have a backing file (e.g., stack and heap) and the data on them disappear when they are unmapped.

Another classification is based on privacy: *private* and *shared*. Private memory is part of only one virtual memory space. This memory space can be accessed by multiple threads in a process, but not by other processes. Shared memory can be accessed by different processes.

Shared file-backed pages are of interest because their content does not disappear when they are unmapped, and they can be mapped by multiple processes at different times. This makes them suitable for mounting elaborate attacks on memory-marking systems.

C. Copy-from-User and Copy-to-User

Linux differentiates between accesses to user-space and kernel memory and therefore uses a well-defined interface when accessing user-space memory. User-space memory can be written to the disk and evicted from the main memory. Accessing absent pages triggers a page-fault handler, where they are read back to memory. Kernel memory pages are always present and triggering a page-fault handler in the kernel is considered a serious error. The kernel uses a well-defined interface that handles possible page-faults when accessing user-space: `(__)copy_(from/to)_user`, `(__)(get/put)_user`, `user_str(cpy/len)`. When the actual implementation is not important, this API is refered to `read_in` and `write_out`. TikTok extends this interface to perform additional checks and the bookkeeping of marked pages.

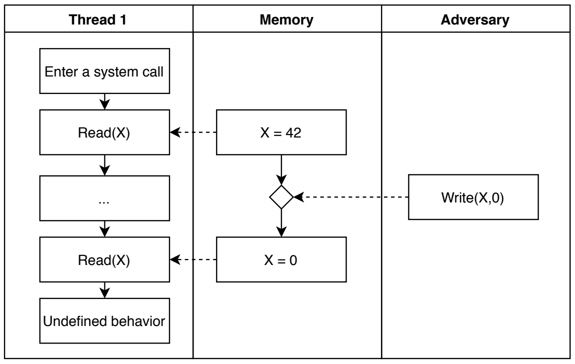


Fig. 1: Diagram of a double-fetch bug.

D. *Double Fetch Bugs*

```
 1  static long vbg_misc_device_ioctl(
 2          struct file *filp,
 3          unsigned int req,
 4          unsigned long arg)
 5  {
 6    size_t size;
 7    struct vbg_ioctl_hdr hdr;
 8    void *buf;
 9
10    if (copy_from_user(&hdr, (void *)arg, sizeof(hdr)))
11      return -EFAULT;
12
13    if (hdr.version != VBG_IOCTL_HDR_VERSION)
14      return -EINVAL;
15
16    if (hdr.size_in < sizeof(hdr) || (hdr.size_out && hdr.
       ↪ size_out < sizeof(hdr)))
17      return -EINVAL;
18
19    ...
20
21    if (copy_from_user(buf, (void *)arg, hdr.size_in)) {
22                ret = -EFAULT;
23                goto out;
24    }
25
26    ...
27
28    ret = vbg_core_ioctl(session, req, buf);
29
30    ...
31  }
```

Listing 1: Abridged CVE-2018-12633 Double Fetch.

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 (Figure 1). In between those two reads, memory could have been changed by an unprivileged adversary. Considering that the bug relies on carefully timed accesses for two different threads, it is a race condition. The situation where the first fetch validates the value of the fetched variable, but the computation is only performed on the second fetch, is called a *time-of-check to time-of-use* (TOCTTOU) bug. TOCTTOU bugs have been widely studied in file systems, where the API makes it possible to swap the file after validating the access rights [23], [24], [37], [31].

Listing 1 displays the double-fetch bug in the Virtual Box drivers for the Linux kernel [7]. The first fetch occurs on line 10. The code gets the header, checks the arguments (lines 13 – 17) and fetches the whole argument into the variable `buf` (line 21). The new value is then used (line 27).

Note that the header is fetched twice—on lines 10 and 21. This gives the opportunity for the attacker to change the values. Considering that the header is not verified the second time, the attacker can leave the kernel in an inconsistent state. The fix for this bug (Listing 2) does not fetch the header the second time. It copies the header into `buf` and the second fetch skips it.

```
18  ...
19
20    *((struct vbg_ioctl_hdr *)buf) = hdr;
21    if (copy_from_user(buf + sizeof(hdr), (void *)arg +
        ↪ sizeof(hdr), hdr.size_in - sizeof(hdr))) {
22      ret = -EFAULT;
23      goto out;
24    }
25
26  ...
```

Listing 2: CVE-2018-12633 Double Fetch Fix [13]

Wang et al. explain in [34] that double-fetches appear not only in kernels, but wherever there is a trust boundary to cross (e.g., kernel — hypervisor [38] and hardware—kernel boundaries [20]). Double-fetches have been responsible for many vulnerabilities in different kernels [18], [34].

III. Threats and Attacks

We now describe the threat model and attacker capabilities along with the attacks that can be mounted to change the data in memory, to exploit a double-fetch bug.

A. *Threat Model*

The adversary has access to a user account on the target machine. They can execute arbitrary 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 attacker may execute an arbitrary sequence of system calls. 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 [22] or side-channels [19], and file-system TOCTTOU attacks [23], [24], [37], [31] are out of scope.

B. *Attacks Classification*

TikTok guards data passed into the kernel against different forms of modification. While writes from user-space and the kernel are straight forward, there are also subtle ways to evade the protection of a memory-marking system. A non-obvious attack vector focuses on memory remapping, e.g., concurrent calls to `mmap`, remapping the same region multiple times, or writing to an opened but mmapped file. Most of these attacks could be prohibited by disabling file-mappings but this would negatively impact software compatibility. Linux relies on multiple page mappings and file mappings to provide shared-libraries and load programs into memory. The following attacks are possible:

1) *Direct double fetch* consist of a malicious user trying to directly write to the argument stored in memory, while the system call is executed in another thread.
2) *Privileged double fetch* are indirectly triggered. The user performs a `read` system call with the target address as the destination. The data is then written to this location by the `read` system call from the kernel.
3) *Reflected double fetch* is accomplished by mapping marked pages in another process. The malicious user stores the arguments on a file-backed page and maps the file again as writable, while the system call is executing. The user is then free to write to the new mapping, even though *other mappings* are read-only.
4) *Inception double fetch* bypasses the protections present in user-space mappings by directly changing the data present in the files.
5) *Device double fetch* can change independently of writes. Storing arguments on such pages can result in reading different values from them, even if no writes occur in-between the reads.

Watson [36] in his analysis of CerbNG introduces Attack 1, Attack 3 and Attack 4. We extend the discussion of known attacks and introduce Attack 2 and Attack 5.

IV. TikTok Design

Designing a memory-marking system is prone to pitfalls. A naive design would instrument `read_in` to *mark* the pages it accesses with a *superuser* flag in the page-table and force the writers to *wait* in the page-fault handler. When returning from the system call marked pages are *unmarked*, and waiting writers are woken-up. Marking the page in all VM spaces renders it unwritable by all user processes. However, calling `read` with the page as the destination will bypass such a system. System calls execute with superuser privileges and are allowed to write to protected pages.

Based on our study of memory marking systems, we introduce the following design principles to mitigate double-fetches:

- *Temporary Immutability:* Pages storing system call arguments become immutable after being read;
- *Exactness*: Writes to protected pages do not affect the execution of the threads performing them; and
- *Correctness:* The only way to change the protected pages is by writing to them.

By enforcing these three principles, the system mitigates double-fetch bugs and protected programs execute correctly.

The naive design violates several of our design principles. Data can still be changed because it does not protect against arbitrary memory mapping. Users can bypass the protection by mapping guarded pages as writable in another process (Attack 3). Self-writing calls do not execute correctly—they deadlock when writing to their arguments. Watson [36] mentions self-writing calls as a fundamental problem for memory-marking systems. Finally, the naive design prevents writes from updating the memory. Memory locations that can change independently, such as memory mapped device registers, invalidate the protection (Attack 5).

TikTok marks guarded pages as *read-only*. While read-only pages will prohibit privileged code from writing to it, some system calls (e.g., `rt_sigaction`) write to the arguments they read and will deadlock. TikTok accommodates these special system calls following the introduced design principles.

A. Temporary Immutability

Temporary Immutability of argument pages guarantees that repeatedly reading from the same location returns the same data.

TikTok relies on the page-table and the page-fault handler to provide immutability. The interface for reading from the user-space (`read_in`) is extended to change the permissions of the pages being read to *read-only*. The pages need to be marked in all VM spaces. Otherwise, the adversary could write to the page from a different process.

TikTok extends the mapping of pages to make sure the guarded pages are marked in all processes. This protection holds even if they are mapped as writable after being marked by another process. If the OS supports *on-demand paging* the marking should happen when the page is accessed and loaded by the process.

Temporary immutability prevents Attack 1, Attack 2 and Attack 3.

B. Exactness

Self-writing system calls are problematic for memory-marking systems. Some system calls rely on double-fetches and user-space writes to the arguments to execute correctly. TikTok also introduces additional synchronization points to user-space programs, requiring a deadlock prevention strategy. TikTok preserves the equivalent execution of these system calls and user-space programs.

Self-writing system calls are addressed by preventing them from writing to marked pages and triggering a page-fault. When a page is marked, it is also added to a data structure containing marked memory memory ranges for each VM space. We change the interface for writing to user-space (`write_out`) to check if it is writing to a marked page. Our added data structure records the writes to marked pages, and executes them at the end of the call, after unmarking the arguments. This guarantees that system calls will not wait for themselves to unmark the pages. Watson [36] stressed that no system call wrappers had successfully addressed this problem.

System calls that depend on double-fetches cannot be protected by TikTok, so they must be *whitelisted* so that they do not mark pages. Table I provides a list of these system calls and a reason why each system call needs whitelisting.. They must be manually inspected for bugs. Whitelisting them does not diminish the protection of other calls. Whitelisted calls still stop before writing to marked pages.

The best representatives are `pollfd`, `futex`, and `execve`. `pollfd` checks if any of the file-descriptors in the array passed as an argument are ready to perform I/O. The

Whitelisted System Call	Reason
`futex`	Relies on outside writes
`poll`	Relies on outside writes
`ppoll`	Relies on outside writes
`select`	Relies on outside writes
`pselect6`	Relies on outside writes
`rt_sigtimedwait`	Relies on outside writes
`execve`	Remaps memory
`nanosleep`	Keeps pages marked while it sleeps

TABLE I: Whitelisted system calls.

user can write `-1` to memory to indicate that the loop in the call needs to skip the descriptor. `futex` is a mostly user-space lock. Most of `futex` synchronization depends on atomic user-space writes, with the system call being performed only to awake or make threads wait. If `futex` marks a user-space address as read-only, other user-space threads will be incapable of unlocking the `futex` by writing to that address. `execve` detaches the VM space and creates a new one. Whitelisting it simplifies the detaching and destruction of VM spaces in TikTok.

Even though they execute correctly under TikTok, long-lasting calls (such as `sys_nanosleep`) should be included in this group. Otherwise, they could keep the pages marked for long periods, sometimes preventing the program from executing in reasonable time.

Finally—if a system call can be verified not to have double-fetches, it can be whitelisted to improve performance. This is especially important in case of system calls with large arguments such as `write`.

C. Preventing Deadlocks

TikTok provides its guarantees by introducing additional synchronization points to executing programs. It neither introduces deadlocks by interacting with itself, nor by interacting with blocking system calls.

TikTok cannot cause a circular locking dependency as a consequence of its *system call reads, system call writes, or markings*. Every dependency involves one marked page and a write to it. A self-deadlocking execution trace would involve a write from a system call to a page marked by another system call. The other system call would also need to write to the page marked by the first call to complete the cycle. This situation is impossible in TikTok because system calls postpone writes to marked pages until the end of the call, when the pages are unmarked.

The deadlocking pattern requires three ingredients:

- A synchronized marking call `S`
- Storing the arguments of `S` in shared memory `A`
- A thread to write to `A` while `S` is executing
- An already existing waits-for dependency between the thread executing `S` and a write

Figure 2 shows an example of a deadlocking communication pattern. Thread 1 enters the system call `S` and marks a shared page `A` (**1**). The system call `S` blocks until the corresponding call `unblock_S` is called in Thread 2 (**3**). While the page `A` is still marked, Thread 2 attempts to write to it, causing it to wait for `S` to finish (**2**). All the deadlocking patterns need to exhibit a similar combination of paired blocking calls (`S`, `unblock_S`) with arguments in the shared memory (page `A`).

```
1: S(A,T1);
```

(a) Thread 1

```
2: write(A);
3: unblock_S(T2);
```

(b) Thread 2

Fig. 2: Executing instructions in the specified order causes a deadlock with TikTok.

The only candidates for `S` that we have encountered are blocking message-passing calls. They are synchronous and thus provide the necessary waits-for dependency, and they read buffers from user-space memory, marking pages. However, the deadlock would imply that data on the page `A` is shared both by message-passing and shared-memory. We have not encountered such a pattern and there are a few reasons why it will not occur in practice:

- Most programs use only one IPC method (either message-passing or shared-memory)
- System call arguments are usually stored on the stack, and accessed only by the executing thread
- The programs would need to use both IPC methods at the same time, on the same data
- Both system calls and shared-memory are relatively rare in programs, and used in the well-defined patterns

TikTok adds a synchronization point to the kernel to prevent write calls to files from having marked mapped pages. While it is possible to deadlock a thread by mapping a file, and then passing the mapped pages as the arguments of `write` to the same file, we have not encountered such a case in practice. There is no reason to use `write` to update the file. It is already mapped an can be edited directly by updating the mapped memory.

In any case, if a system call interacts with TikTok to cause a deadlock, such a call can be *added to the whitelist*. During our extensive evaluation and testing we have not encountered any deadlocks.

D. Correctness

TikTok stops all writes to marked pages to prevent changes to the arguments. It is necessary to close off all other ways of changing data on marked pages to show TikTok is *correct*. We identify and disable two other ways of changing pages—write system calls and device-backed pages.

File-systems with shared *mapped memory* and *write buffers* can bypass the memory-marking protection in a `write` call [36]. To prevent this attack, TikTok pauses all writes (`vfs_write`) to files having marked mapped pages.

Discretionary Access Control (DAC) can be used to prevent the mounting of devices to user-controlled processes. Device-backed pages may have a state independent of user actions, and even the kernel, making any form of preventing data changes impossible. We do not consider this a flaw in TikTok because users usually are not allowed to access devices directly.

This design principle prevents Attack 4 and Attack 5.

V. Implementation

The TikTok prototype targets Linux x86-64. However, TikTok can be ported to any operating system that uses a defined interface for reads and writes to the user-space (`read_in` and `write_out`), and any architecture that has page-tables encoding access control information. Depending on the available resources (e.g., the number of free bits in the PTE) information can be stored in different places (e.g., PTE or global structures).

Figure 3 illustrates some of the most important data structures used in the prototype—the PTE with the permissions, the marked page metadata (number of threads marking the page frame, number of threads waiting for the unmarking, the reverse mapping information).

Section V-A describes the data about the physical page frame with some of the limitations due to the need to keep `struct page` as small as possible. Considering that one `struct page` exists for every page frame on the system, increasing its size would drastically increase the memory overhead. Section V-B explains the data stored in the page table (permissions, free bits and page frame number).

A. Storing the Page Frame Information

Linux divides physical memory into page frames. Each page frame is represented by a `struct page`. This structure is replicated millions of times and every additional field has a tremendous impact on memory consumption.

To keep the memory consumption low, TikTok uses a single, so far unused, bit in `struct page` to mark page frames. On x86-64 we repurpose one of several unused bis in the flag field for this purpose. Architectures that have fewer flag bits (such as x86) can instead use some other bit (e.g., Kernel Shared Memory, or NUMA domains). Figure 3 denotes this field as the *Page Marked Flag*. The `struct page` stores a pointer to the *reverse mapping* information. The reverse mapping enables TikTok to find all PTEs that require (un)marking.

The marking metadata is stored in a hashmap based on the *page frame number*, allowing us to store only the information on the pages which are currently marked and reduce memory impact. TikTok protects access to these entries by separate mutexes to improve its scalability. The metadata consists of the page frame number, the count of threads marking the page (*owners*), the number of threads waiting for the page to get unmarked (*guests*), and a *completion* they are waiting on.

B. PTE Information

When a page is marked, TikTok changes some of the flags in the PTEs to mark it:

R/W gets set to *read-only* to prevent writes to the page
SW2 gets the old value of **R/W**
SW3 gets set to 1

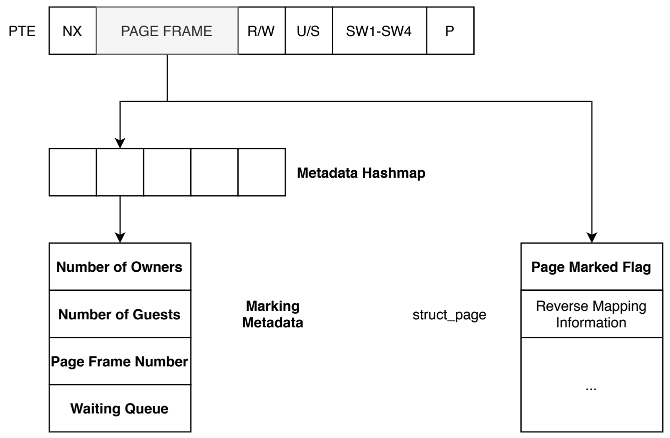


Fig. 3: TikTok bookkeeping information. New data structures are in bold. The page frame and marking information are fetched based on the page frame number. The marking information is allocated on demand and stored in a hashmap to preserve memory.

SW2 is used by the Software Dirty Pages feature of Linux. This feature cannot run alongside TikTok in our prototype. Other architectures may have less, or more bits, available for the OS to use. In the case of lack of space in the page table, `SW2` and `SW3` could be stored in a separate data structure.

Copy-on-write pages have a peculiar optimization. They are marked only in the process requesting the marking. Writes from other processes will trigger a copy-on-write mechanism, preventing them from changing the data.

TikTok does a partial flush of the TLB and updates the MMU cache to make these changes visible immediately. Section VI evaluates the performance impact of this flush.

C. Marking and Unmarking

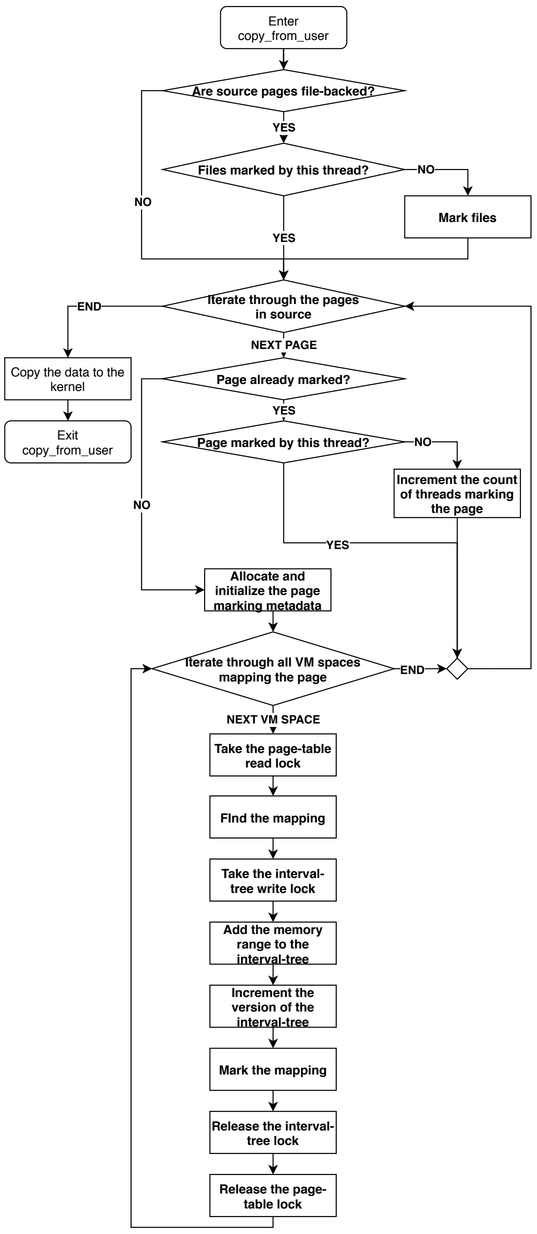
TikTok extends Linux's interface for reading and writing to user-space. The abstractions `read_in` and `write_out` introduced in Section IV are implemented in Linux as `copy_from_user` and `copy_to_user`. TikTok extends `copy_to_user` to mark the pages before reading the data (Figure 4).

Marking the page involves taking the appropriate locks, denoting the physical page frame as marked, and incrementing the number of page owners. TikTok then uses the reverse mapping information to mark the page in all the VM spaces mapping it.

Unmarking follows a similar procedure (Figure 5). The appropriate locks are taken, and the number of owners is decremented. Only if no owners are marking the page anymore, it is unmarked in all VM spaces. If threads are waiting on the page, they are woken up. The last guest to wake up will deallocate the marking data for the page.

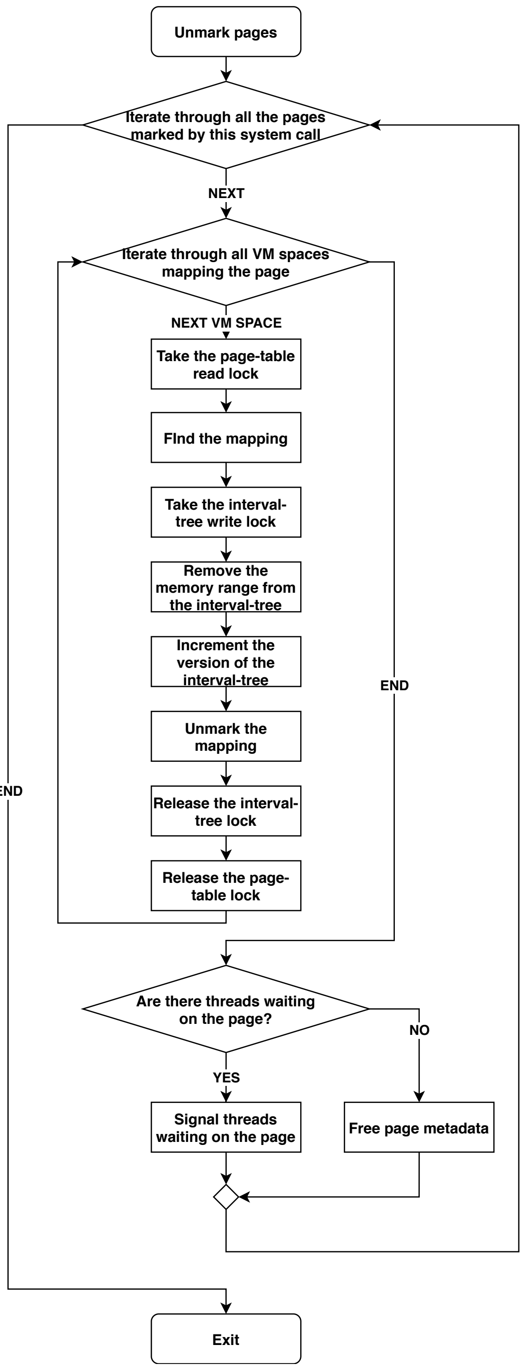
D. Deadlock Prevention

Our implementation shares several data structures among multiple threads (marking information, pages, completions,



Enter copy_from_user
Are source pages file-backed?
YES
NO
Files marked by this thread?
NO
Mark files
YES
Iterate through the pages in source
END
Copy the data to the kernel
Exit copy_from_user
NEXT PAGE
Page already marked?
YES
Page marked by this thread?
NO
Increment the count of threads marking the page
NO
YES
Allocate and initialize the page marking metadata
Iterate through all VM spaces mapping the page
END
NEXT VM SPACE
Take the page-table read lock
FInd the mapping
Take the interval-tree write lock
Add the memory range to the interval-tree
Increment the version of the interval-tree
Mark the mapping
Release the interval-tree lock
Release the page-table lock

Fig. 4: `copy_from_user` page marking. New steps are in bold. Read pages are marked in all VM spaces, as well as their backing files.



Unmark pages
Iterate through all the pages marked by this system call
NEXT
Iterate through all VM spaces mapping the page
NEXT VM SPACE
Take the page-table read lock
FInd the mapping
Take the interval-tree write lock
Remove the memory range from the interval-tree
Increment the version of the interval-tree
Unmark the mapping
Release the interval-tree lock
Release the page-table lock
END
END
Are there threads waiting on the page?
YES
NO
Signal threads waiting on the page
Free page metadata
Exit

Fig. 5: Page unmarking at the end of the system call.

page-tables, or interval-trees). To prevent deadlocks, locks are always taken in increasing order. The only two locks taken in two different orders are the *page-table lock* and the *interval-tree lock*, which protect the marked memory ranges. TikTok features a deadlock prevention mechanism for these locks.

In `copy_to_user` Figure 6 the interval-tree lock must be taken first to check which memory ranges are marked. However, when marking a page the page-table lock needs to be acquired first to locate the page-table entry mapping the page Figure 4. TikTok implements the deadlock resolution mechanism where if the page-fault handler notices that the interval-tree lock is already taken, it gets released.

The prevention mechanism leads to a way for system calls to write and block on marked pages. Ultimately, it cannot cause a deadlock because a user-thread executing in kernel context would need to read and write to user-space at the same time.

The first thread executes `copy_to_user` Figure 6 and attempts a write to page that is not present. This triggers a page-fault which loads the page. Before the first thread re-executes the write to the page, the second thread marks it. The first thread cannot abort the write and is forced to wait. For the second thread to wait for the first, it is necessary for the same race to happen the second time, with the roles reversed. This is impossible, as the first thread is waiting due to its attempted write.

E. Loading File-Backed Pages

Following our proposed design, file-backed pages need to be marked as they are loaded to memory. On-demand paging takes place in the page-fault handler, but invokes the functions of the file-system storing the file. TikTok extends the functionality of `set_alloc_pte` to mark the page. The needed structures are preallocated and passed into the *atomic context* where `set_alloc_pte` is called.

When a marked page is loaded, its data is also added to the interval-tree storing the marked memory ranges. TikTok takes the lock before entering the atomic context, even though the loaded page may not be marked. The version number of the interval-tree is incremented after adding the page, informing all other threads that it has changed.

F. Protecting System Call Arguments from Kernel Writes

TikTok stores the marked memory ranges for every VM space (`mm_struct`) in an interval tree. Before writing to user-space, `copy_to_user` (Figure 6) acquires the lock for the marked memory ranges tree and checks for the intersections with the destination buffer. Writes to marked pages are stored to freshly allocated memory, along with the destination address, and written at the end of the call.

Writes to unmarked pages proceed as normal, unless the page is not present. Then a page-fault exception happens (Figure 6). The page-fault handler takes the page-table lock, loads the page, and restarts the write. If the page-fault handler detects a potential deadlock, it will release the interval-tree lock. The page-table read-lock is not exclusive, so it does not need to be released (Figure 7).

`copy_to_user` will attempt to write to the page after exiting the page-fault handler. In the case it is indeed marked, it will block and wait for its unmarking. This is the only case where TikTok forces threads to wait in the middle of a system call and has been explained in Section V-D. After the write, the thread reacquires the lock, checks if the interval tree has changed, and restarts the write if so.

The restart of the write is necessary because any change to the interval-tree invalidates the iterator. TikTok is optimized for the case of rare markings, making `copy_to_user` calls that do not encounter marked pages fast as only a single, large write occurs. The alternative is to check and write to every page individually. Such implementation does not need to restart the write but incurs the overhead of individually writing to pages.

VI. Evaluation

To fully evaluate the impact of our memory marking system, we need to carefully consider a variety of different usage scenarios both in user-centric (desktop) environments as well as server environments. We build on the large body of existing systems benchmarks to demonstrate performance impact of our system.

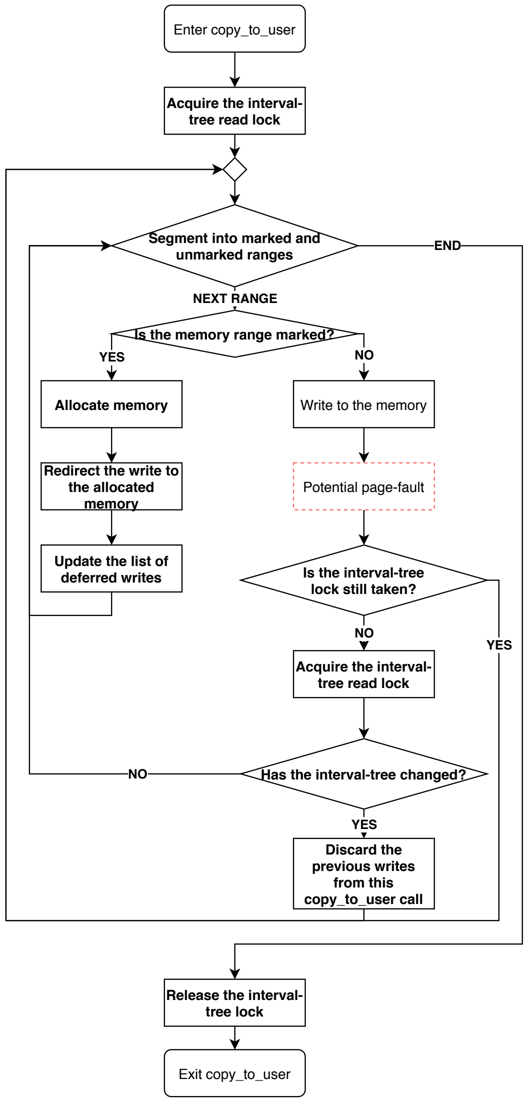

Enter copy_to_user
Acquire the interval-tree read lock
Segment into marked and unmarked ranges
END
NEXT RANGE
Is the memory range marked?
YES
NO
Allocate memory
Write to the memory
Redirect the write to the allocated memory
Potential page-fault
Update the list of deferred writes
Is the interval-tree lock still taken?
NO
YES
Acquire the interval-tree read lock
NO
Has the interval-tree changed?
YES
Discard the previous writes from this copy_to_user call
Release the interval-tree lock
Exit copy_to_user

Fig. 6: `copy_to_user` write handling. New steps are in bold. The write destination is checked for marked ranges. Unmarked ranges are written to, while the writes destined for marked ranges are deferred until the end of the system call. Deadlocking in the page-fault handler is prevented by releasing the interval-tree lock.

TikTok has most impact when marking and unmarking pages during system calls. This overhead is proportional to the number of VM spaces (processes) that have the pages mapped. Benchmarks must measure the TikTok overhead for the multithreaded and multiprocessed workloads, as well as the workloads with many marking system calls.

We evaluate TikTok using two different benchmark suites: the NAS Parallel Benchmark (NPB) [15] and the Phoronix Test Suite (PTS) [14]. NPB tests TikTok on multithreaded or multiprocessed CPU-bound workloads. NPB is a suitable benchmark to measure a drop of performance due to multithreading

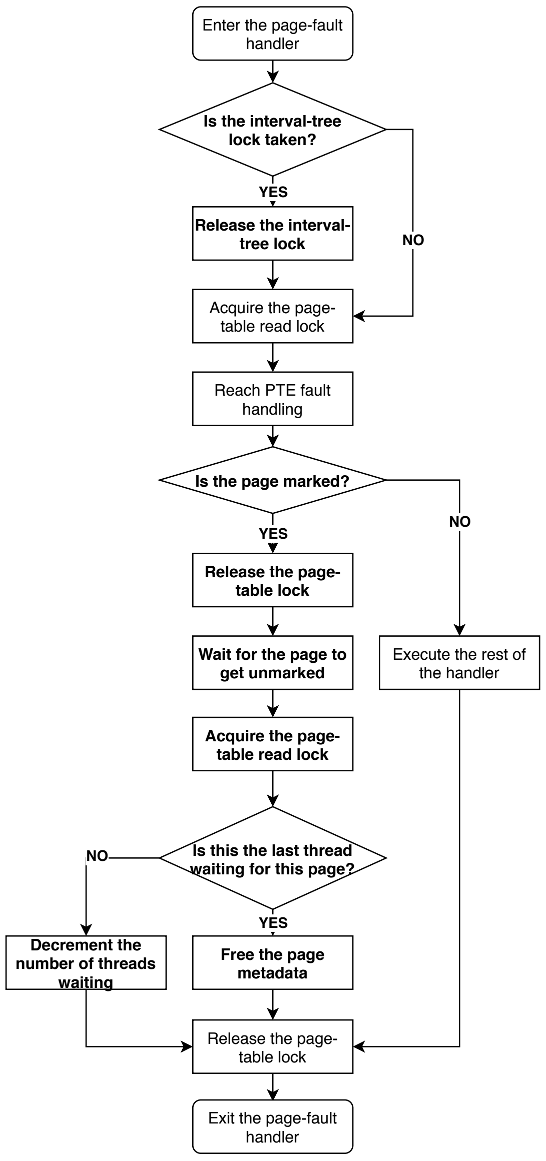

Enter the page-fault handler
Is the interval-tree lock taken?
YES
NO
Release the interval-tree lock
Acquire the page-table read lock
Reach PTE fault handling
Is the page marked?
YES
NO
Release the page-table lock
Wait for the page to get unmarked
Acquire the page-table read lock
Execute the rest of the handler
Is this the last thread waiting for this page?
NO
YES
Decrement the number of threads waiting
Free the page metadata
Release the page-table lock
Exit the page-fault handler

Fig. 7: TikTok's handling of the writes in the page-fault handler. New steps are in bold. Interval-tree lock is released at the start of the handler to prevent deadlocks.

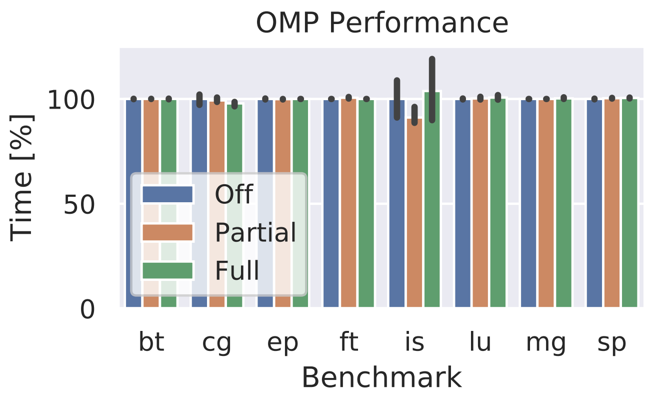


Fig. 8: TikTok evaluated on NPB with OMP.

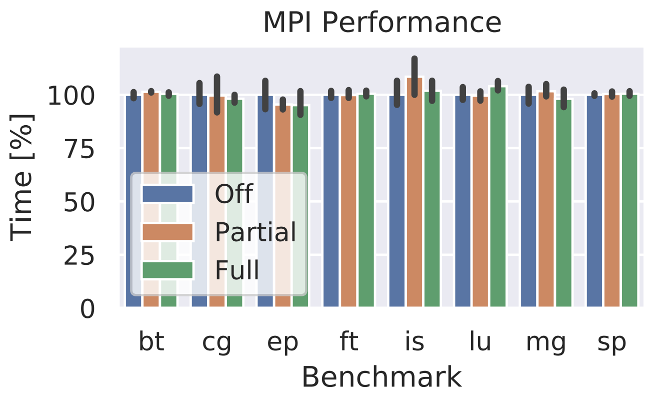


Fig. 9: TikTok evaluated on NPB with MPI.

and multiprocessing. PTS evaluates TikTok on several real-world applications. They complement NPB in that they consist of system call heavy applications, with varying degrees of parallelism. In this evaluation we do not include results of SPEC CPU2006 or SPEC CPU2017 as that benchmark is heavily CPU bound and optimized to have minimal interaction with the operating system, i.e., the number of system calls is extremely low resulting in negligible overhead.

All benchmarks have been performed on Ubuntu Server 18.04 LTS (Linux 5.4.0-rc3) with Intel i7-9700 and 16 GB of RAM. We compare three main TikTok configurations:

TikTok On full protection of system calls,
TikTok Partial whitelists the `write` system call and variants (`pwrite`, `pwrite64`, `pwritev`), and
TikTok Off is the Linux kernel compiled without TikTok.

A. NAS Parallel Benchmark

NAS Parallel Benchmarks (NPB) [15] is a benchmark introduced by NASA. It consists of several parallel programs using different communication patterns. TikTok introduces additional synchronization points between threads, so it should affect multi-threaded programs more than single-threaded ones. NPB is available for multiple technology stacks for parallel programming. TikTok was evaluated on *OpenMP* [16] and *MPI* [30] versions of the benchmark with workloads of class A.

OpenMP [16] is a compiler extension that splits the execution to multiple threads. All threads still use the same VM space, keeping the overhead minimal. TikTok was evaluated with 8 threads.

MPI [30] implements parallel execution by launching multiple processes which communicate by message-passing. Due to the limitations of some benchmarks, TikTok was evaluated with 16 processes.

The benchmarks have been run with TikTok protecting almost all calls, with `write` variants disabled and without TikTok at all (Figure 8 and Figure 9). Both benchmarks show no significant difference between different setups. MPI has wider standard-error margins, probably introduced by the heavier startup and communication. Shorter benchmarks magnify relative differences (explaining the artifacts for `is` and `ep`). TikTok does not incur a noticeable performance penalty for CPU-bound workloads, even when they involve multiple threads/processes.

B. Phoronix Test Suite

Phoronix Test Suites (PTS) [14] evaluates systems on real-world software. PTS tested TikTok on a range of programs complementing NPB: from single-threaded, CPU-bound applications (OpenSSL) to multi-threaded, multi-processed, programs with frequent system calls (Apache).

Figure 10 shows the results of the Phoronix benchmark. All results have been normalized with respect to 'TikTok Off'. Benchmarks marked with an asterisk (*) measure execution time (lower is better), while others measure performance (higher is better).

Linux compilation, OpenSSL, Redis, Git and PyBench do not show a noticable degradation in performance. These programs do not feature a lot of inter-process communication. Even the parallelized benchmarks such as Linux kernel compilation generate independent, non-communicating processes.

However, the two web servers (Apache and NginX) show a significant drop in performance (16.87% for Apache and 16.38% for NginX) with TikTok protecting all calls. Whitelisting write calls improves the situation slightly (15.64% for Apache and 13.41% for NginX). TikTok affects parallel and applications with numerous system calls the most because of the marking overhead. The impact of whitelisting write calls is small due to the low frequency of calls.

The IPC TCP benchmark consists of two system calls that transfer data between two threads. Compared to other benchmarks, IPC benchmark has fewer threads than web-servers, but also executes a significant number of system calls. This places it in-between two groups with respect to the performance penalty.

C. Apache and NginX Overhead breakdown

We evaluated the Apache and NginX benchmarks under two additional TikTok configurations to more accurately assess the effect of system calls on performance (Figure 11):

Frequent system calls whitelisted whitelists the most frequent calls executed in the benchmarks (`epoll_ctl`, `close`, `read`, `fcntl`, `connect`, `recvfrom`, `epoll_wait`, `accept4`, `openat`, `socket`, `shutdown`, `fstat`, `sendfile`, `stat`, `mmap`, `munmap`, `getsockname`, `times`), as well as `write` and its variants. The code of standard system calls has been checked for double-fetches in previous work [33], [39]. `mmap` and `munmap` do not read memory and mark pages. Therefore, they are safe to whitelist.

All system calls whitelisted does not protect any system calls.

Whitelisting frequent system calls leads to a smaller overhead (Apache – 5.34% and NginX – 2.77%). The performance is almost equal to not protecting any system calls at all (Apache – 2.46% and NginX – 2.25%). Some overhead remains even when TikTokis not protecting any calls because `copy_to_user` protection is still active.

To contrast the overhead between two different workloads, we traced Apache and IPC with `perf`. The traces serve as ground truth (TikTok Off) and TikTok with writes whitelisted. Here, the IPC benchmark does not incur any marking overhead, while Apache still marks pages.

Table **??** shows how much time the programs spend in TikTok functions. We measure percentages of individual segments with `perf`, while calculating the total overhead from the results of the benchmarks. While the TLB is flushed both during the marking and unmarking of pages, but is included in the table as an important performance parameter.

The comparison of the Apache and IPC runs shows two very different marking profiles. Apache spends a considerable amount of time marking and unmarking pages, while IPC does not mark pages at all. This is a consequence of the different system calls used by these applications. IPC uses only `read` and `write` calls which do not mark arguments, while Apache also invokes marking system calls (`accept`, `getsockname`).

Apache waits most of the time for the TLB signaling functions to execute. `flush_tlb_mm_range` amounts for 2.51% (marking) and 2.16% (unmarking) of the total time spent executing the benchmark. Frequent TLB flushes are expensive because the CPU needs to make sure that all the cores have processed the signal before continuing. Flushed pages will be accessed right afterward, leading to a TLB miss. This further inflates the overhead.

IPC benchmark executes only `write` and `read` calls. It does not incur any overhead on TLB flushes because none of these calls mark pages. However, `copy_to_user` checks are still active, causing a noticeable locking overhead.

VII. Related Work

The literature related to double-fetches can be broadly divided into two groups. Static solutions are in the first group. These systems use techniques such as source or binary analysis to detect double-fetch bugs. The second group of systems uses run-time information to detect and (rarely) prevent double-fetches.

The most inspiring work related to TikTok is the paper by Watson [36], criticizing the security of system call wrappers.

A. Watson's Critique of System Call Wrappers

Watson's paper [36] scrutinized the security of many system call wrappers. Not only did he discover that all existing system call wrappers were fundamentally insecure, Watson also described the different types of TOCTTOU bugs that compromised them and discussed potential fixes. In a short paragraph, he mentions that Pawel Dawidek, the creator of CerbNG [40], has experimented with marking arguments read-only. CerbNG was an early system call filtering system for BSD that used copy-on-read-and-write to a new memory page.

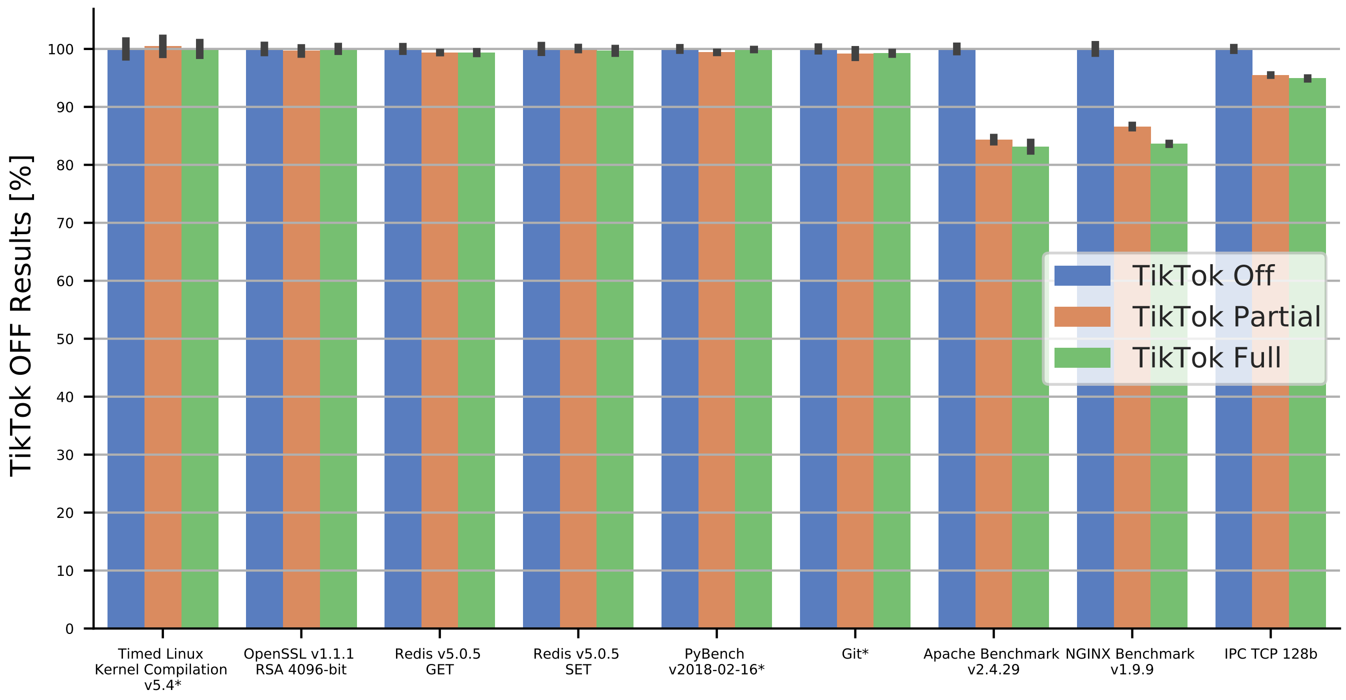


Fig. 10: TikTok evaluation on Phoronix Test Suite.

	Apache (TikTok Off)	Apache (TikTok - No writes)	IPC (TikTok Off)	IPC (TikTok - No writes)
Marking pages read-only	0%	5.22%	0%	0%
Unmarking pages	0%	4.67%	0%	0.94%
Flushing Individual TLB Ranges	1.50%	6.96%	0%	0%
Shared Write Buffers Protection	0%	0.05%	0%	1.72%
`copy_to_user` Protection	0%	1.93%	0%	1.28%
Measured Overhead	0%	18.56%	0%	4.72%

TABLE II: Perf Analysis of the Benchmarks. The percentages are normalized to TikTok Off execution times.

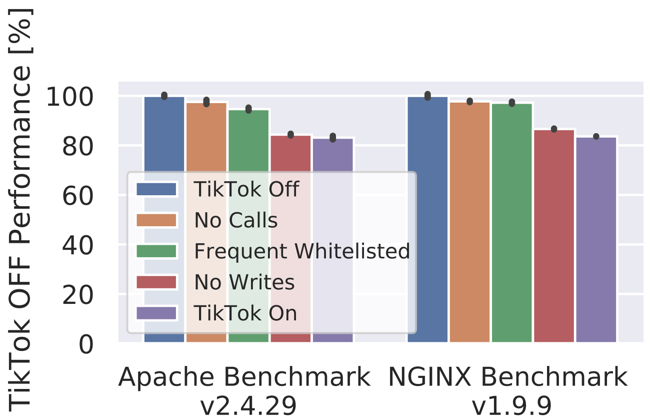


Fig. 11: Detailed comparison of Apache and NginX performance.

Our very first design sketch followed a similar idea but we quickly discovered the potential drawbacks that remained.

Afterward, Watson briefly discusses problems such memory marking systems must solve:

- unnecessary page-faults,
- bypassing memory marking using IO system calls,
- mapping shared memory with different permissions, and
- handling system calls that both read and write to the same memory.

According to Watson, no memory-marking system (including CerbNG) addressed all of these pervasive problems. TikTok does exactly that and even includes protection against new attacks that Watson was not aware of. Unnecessary page-faults are rare and they are used to make the offending threads wait for unmarking. After the page has been unmarked, the write proceeds without any consequences. Write system calls do not proceed until there are no marked pages of the file. Pages are marked when they are mapped and only if needed. TikTok postpones all writes to marked pages coming from the kernel while allowing the system calls to execute correctly.

B. Static Analysis Work

Static analysis techniques analyze the source code to find double-fetch bugs. Wang et al. [33] used pattern matching to find potential double-fetches. They implemented a tool that patches certain double-fetches automatically. However, their method in the general case produces false positives that must be manually inspected. Xu et al. [39] improved on this work by proposing *Deadline*. Deadline does not use the pattern analysis on the source files to detect double-fetches, but a compiler's intermediate representation and constraint solving to reduce false positives.

Static analysis techniques such as these have the benefit of discovering bugs in the code that cannot be run (e.g., mitigating the need for specific hardware to test a particular driver). However, they are meant for bug detection, not mitigation. Even though the tools can discover some bugs automatically, this is not always possible. System call wrappers have, by design, an inherent TOCTTOU bug.

Double-fetches introduced by the compiler are another challenge. While the source code and even the intermediate representation may be bug free, the compiler can introduce such "invisible" double-fetches when allocating registers to variables. TikTok protects the system against all kinds of double fetches.

C. Dynamic Analysis Work

Google Project Zero's Bochspwn [18] uses an emulator to detect double-fetches. It found a large number of bugs in the Windows kernel. Bochspwn works on binaries (i.e., it needs no source code access) and detects bugs introduced by compilers. DFTracker [35] is another dynamic analysis technique, with a lower overhead, that relies on taint tracking. However, these dynamic techniques are limited in their detection of double-fetches. First, similar to work presented in Section VII-B, developers need to manually fix all discovered bugs. Second, in contrast to static techniques, with dynamic analysis the double-fetch must also be executed by some concrete code and input, limiting this technique to the core kernel and to the drivers with the available hardware.

A big leap in dynamic analysis techniques has been presented by Schwartz et al. [26]. The first part of the paper introduces DECAF, a framework that uses side-channel attacks to create a fuzzing oracle for double-fetch bugs. While Bochspwn relies on emulation, slowing the execution significantly, DECAF runs natively. It also eliminates false positives by automatically exploiting found bugs.

Schwartz et al. then discuss DropIt, a real-time mitigation technique for double fetches. DropIt uses Intel's *Transactional Synchronization Extensions* (TSX) [17] in a creative way to prevent double-fetch bugs. By encapsulating the code in a TSX transaction, writes from other threads will result in the transaction being aborted. However, the code executing inside a TSX transaction is severely limited. All reads must fit in the L3 cache, and all writes in L1. Some instructions are also forbidden. TikTok faces none of those limitations. It works on non-Intel processors and relies on page tables for protection, a technique that has been present for several decades.

VIII. Discussion

TikTok can be ported to any kernel that accesses user-space memory through a well-defined interface, and any computer architecture that features page-tables. Android devices frequently have binary-only drivers and porting TikTok to ARM would provide additional safety guarantees to the Android ecosystem.

Current system call filters [25], [29] prohibit TOCTTOU races by implementing filters as Linux Security Module (LSM) [21] hooks. For the filters to work, LSM hooks need to be present in drivers. Even then, the undesired behavior may manifest before a hook is encountered. Integration of TikTok with an existing system call wrapper would solve these TOCTTOU races. SecComp [12] and eBPF [11] are the obvious candidates that would benefit from such an extension. TikTok performance could be improved by marking only the pages that are read by the filter in the system call wrapper.

The performance overhead of TikTok for multithreaded, system call heavy applications is low but not negligible due to the marking overhead. A possibility of batching TLB flushes for multiple pages should be explored as a possible optimization.

IX. Conclusion

TikTok mitigates double-fetch bugs in system calls by using page-tables. It works both on the core kernel, and drivers, even when their source code is not available. It can be ported to any architecture that uses page-tables, and any kernel that has a well-defined interface to access user-space memory.

Our prototype implementation of TikTok for x86-64 protects modern systems (e.g., Ubuntu Server 18.04 LTS) with complex programs running (`systemd`, `gcc`, `Apache`). When protecting rare system calls our mitigation incurs a negligible overhead of 2-6%, which raises to a low overhead of 17% when protecting almost all calls in multithreaded, system call intensive programs. CPU-bound programs do not have a significant overhead, even if they use multiple threads.

Availability

The source code of TikTok is available at LINK. It has been released under the GNU Public Licence.

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] eBPF. https://lwn.net/Articles/740157/.

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

[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] 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.

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

[17] Intel. Intel 64 and IA-32 Architectures Developer's Manual. 1, 2019.

[18] 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.

[19] 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.

[20] 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.

[21] James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, pages 17–31. ACM Berkeley, CA, 2002.

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

[23] 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.

[24] 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.

[25] Mickaël Salaün. Landlock LSM. https://landlock.io/.

[26] 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 *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[27] 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.

[28] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system principles*. John Wiley & Sons, 10 edition, 2018.

[29] K. P. Singh. Kernel Runtime Security Instrumentation. https://github.com/sinkap/linux-krsi.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

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

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

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

[39] 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)*, pages 661–678. IEEE, 2018.

[40] Slawek Zak, Przemyslaw Frasunek, and Pawel Jakub Dawidek. CerbNG. https://sourceforge.net/projects/cerber/files/cerb-ng/.