



# OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering

Dae R. Jeong[†*] Yewon Choi[‡] Byoungyoung Lee[§] Insik Shin[‡¶] Youngjin Kwon[‡]
[†] *School of Cybersecurity and Privacy, Georgia Tech*
[‡] *School of Computing, KAIST*
[§] *Department of Electrical and Computer Engineering, Seoul National University*
[¶] *Fluiz*

## Abstract

Kernel concurrency bugs are notoriously difficult to identify, while their consequences severely threaten the reliability and security of the entire system. Especially in the kernel, developers should consider not only locks but also memory barriers to prevent out-of-order execution from breaking the correctness of concurrent execution. Incorrect use of memory barriers may cause non-intuitive concurrency bugs that manifest due to out-of-order execution, which we refer to as OOO bugs.

This paper aims to identify OOO bugs in the kernel. We devise a mechanism to emulate out-of-order execution while kernel code is executed, called OEMU. Inspired by how a processor reorders memory accesses, OEMU makes the subtle and non-deterministic behavior of out-of-order execution systematically controllable. Based on OEMU, we propose OZZ, a new testing tool designed to effectively identify kernel OOO bugs. The key feature of OZZ is its ability to deterministically control both out-of-order execution and concurrent execution caused by thread interleavings, enabling comprehensive testing of their combined effects. Our evaluation shows that OEMU is effective in reproducing previously-reported kernel OOO bugs, demonstrating its strong capability of controlling out-of-order execution. Furthermore, with OZZ, we identify 11 new OOO bugs in the latest version of the Linux kernel, subsequently confirmed and patched by kernel developers.

***CCS Concepts:*** • **Computer systems organization** → **Reliability**; *Processors and memory architectures*; • **Security and privacy** → *Operating systems security*.

[*] This work is mostly done while at KAIST.


*SOSP '24, November 4–6, 2024, Austin, TX, USA*

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695944

## 1 Introduction

In the multicore era, concurrency has been a paramount driver in optimizing the performance of systems. To maximize the benefits of concurrency, system software such as the kernel has employed a number of advanced techniques including read-copy-update [45, 47], reference counters [14], and lock-free data structures [46, 104]. The advantages of these techniques come with an increased burden on developers. If developers do not properly synchronize threads, concurrency bugs can arise, compromising the reliability and security of the system software.

This challenge is further compounded by the growing adoption of machines with weak memory models, such as Apple Silicon chipsets [107]. Kernel developers should consider not only using locks but also employing memory barriers. Memory barriers are instructions that prevent out-of-order execution of memory accesses that should not be reordered. Misusing memory barriers may cause concurrency bugs that manifest due to out-of-order execution of concurrent threads, which we call OOO *bugs*. Consequences of OOO bugs are severe, such as denial-of-services [8], system crashes [60, 120], data loss [62], or even well-known security risks such as memory corruptions [17, 82]. Unfortunately, OOO bugs are notoriously difficult to identify, as they manifest by the combination of *two non-deterministic behaviors*: thread interleaving and out-of-order execution.

Over the decades, we have been armed with mechanisms to address the non-determinism of thread interleaving. Because several mechanisms are proposed [9, 18, 20, 40, 44, 48, 59, 63, 64, 76, 99, 105, 111] to deterministically control thread interleaving, automated testing techniques (*e.g.*, fuzzing) emerge to effectively identify concurrency bugs. They control thread interleaving in various ways, such as leveraging breakpoints [13, 18, 39–41, 43, 65] or temporary suspending virtual CPUs [20, 22, 23] and show remarkable capability in the identification of concurrency bugs.

In contrast, addressing the non-determinism of out-of-order execution remains under-explored. While existing processor features, such as breakpoints, can be utilized for testing thread interleavings, testing out-of-order execution requires a completely new approach. This is because out-of-order execution occurs internally within the processor and has traditionally

been considered beyond external control. Furthermore, techniques used to control thread interleavings can unintentionally obscure the effects of out-of-order execution. For example, using breakpoints enforces an order on memory accesses, which prevents the natural observation of memory access reordering caused by out-of-order execution.

This work introduces a new way for systematically testing OoO bugs. We propose a new mechanism to control out-of-order execution. The important property of the mechanism is making memory access reordering observable and controllable even when using a breakpoint. Therefore, a testing framework can utilize an existing mechanism for controlling thread interleaving [20, 22, 23, 39], while simultaneously employing the proposed mechanism to deterministically control the behavior of out-of-order execution. This powerful combination enables the development of a bug-finding tool for kernel OoO bugs.

**In-vivo out-of-order execution emulation [§3].** Because controlling processors' out-of-order execution is practically challenging during runtime, a number of previous approaches [10, 21, 32, 33, 37, 71, 80] adopt an alternative, which collects memory accesses after running a target program, and simulates out-of-order execution through offline analysis. While these approaches might partially expose behaviors stemming from out-of-order execution, they fail to apply kernel runtime contexts (*e.g.*, a list of freed memory objects), limited in identifying bugs such as double-free bugs [82].

Unlike previous approaches, we present an *in-vivo*[1] out-of-order execution emulation (shortly, OEMU). OEMU is integrated into the kernel in the compiling time. While executing the kernel, OEMU operates as a part of the kernel code and reorders memory accesses explicitly during the execution of the kernel. OEMU exposes interfaces to a bug-testing tool to specify how to reorder memory accesses in the kernel. Therefore, the bug-testing tool can fully consider kernel runtime contexts (*e.g.*, a list of freed memory objects) and leverage bug-detecting oracles already deployed in kernel (*e.g.*, sanitizers [66, 88, 93] or kernel assertions). This in-vivo integration enables building automated bug-finding tools.

**Identifying OoO bugs through fuzzing [§4].** This work proposes OZZ, a fuzz testing tool focusing on discovering OoO bugs. Using OEMU, OZZ is capable of testing both out-of-order execution and thread interleaving while most previous approaches [13, 22, 23, 39, 41–43, 96, 112, 113] consider in-order execution only. To our best knowledge, OZZ is the first fuzz testing to to detect OoO bugs

The key idea of OZZ is to test the necessity of a hypothetical memory barrier. In each test, OZZ operates under the assumption that a memory barrier is missing while executing concurrent system calls. It then reorders memory accesses using OEMU in the scenario that the missing memory barrier would prevent if it were present. If it causes a kernel malfunction, OZZ concludes that an OoO bug occurs due to the missing barrier. Additionally, after identifying an OoO bug, OZZ provides the location of the hypothetical memory barrier, providing a hint to developers about how to fix the bug.

**Contributions.** We run OZZ against mature versions of Linux, and identified new 11 OoO bugs from popular subsystems such as TLS and BPF. We report them to kernel developers, and developers confirm and fix them accordingly. We also show that OEMU has the strong capability in reproducing OoO bugs previously found in a deployed kernel.

We want to underscore that OZZ has the practical advantage in its cost-saving potential. Google allocates substantial resources to maintain continuous SYZKALLER operations on x86-64 machines. However, OoO bugs predominantly manifest on machines with weak memory models like ARM. Without OZZ, Google would need to acquire numerous ARM machines to utilize SYZKALLER for discovering OoO bugs. Although SYZKALLER utilizes QEMU's TCG, a dynamic binary translation engine to execute different architectures' instructions, on x86-64 machines to execute ARM Linux, it is incapable of detecting OoO bugs as TCG does not reorder memory accesses.

The contributions of this paper are threefold:

- **In-vivo emulation of out-of-order execution.** This paper presents OEMU, a runtime mechanism to control out-of-order execution. It makes out-of-order execution controllable and observable, enabling an automated bug-finding tool for identifying OoO bugs.
- **A testing tool for OoO bugs.** OZZ is the first runtime testing tool to identify kernel OoO bugs. OZZ deterministically controls both out-of-order execution and thread interleaving, effectively disclosing OoO bugs.
- **Practical contributions.** We identify 11 new real-world OoO bugs in the Linux kernel, which have been confirmed and patched by developers. OZZ and OEMU are publicly available at https://github.com/casys-kaist/ozz.

## 2 Background and Motivation

This section provides background on how out-of-order execution causes a concurrency bug if a memory barrier is omitted.

### 2.1 Out-of-Order Execution and Memory Barrier

**Out-of-order execution.** Modern processors often do not execute machine instructions in the order written in the binary. Instead, they perform *out-of-order execution* [100, 109], which may reorder memory accesses to reduce pipeline stalls or to improve the cache utilization. Due to its huge performance benefits [110], out-of-order execution is widely used in most modern high-performance processors [4, 5, 15, 34].

[1] *In-vivo* is a Latin word meaning 'in the living body.' In this paper, we refer to OEMU as an in-vivo emulation because it is transplanted to the kernel and reorders memory accesses as a part of the kernel's operation.

| Type | Memory barrier API | Precedent accesses | Subsequent accesses |
|---|---|---|---|
| Full | smp_mb() | loads/stores | loads/stores |
| Load | smp_rmb() | loads | loads |
| Store | smp_wmb() | stores | stores |
| Release | smp_store_release(&a, v) | load/stores | store to &a |
| Acquire | smp_load_acquire(&a) | load from &a | loads/stores |
| Relaxed | READ_ONCE()/WRITE_ONCE() | none | none |

**Table 1.** Examples of memory barriers provided by Linux. Each memory barrier guarantees that precedent accesses are completed before subsequent accesses.

```
1  /******* Thread A *******/
2  /* kernel/watch_queue.c */
3  void post_one_notification() {
4    buf = &pipe->bufs[head];
5    buf->len = len;
6    buf->ops = &wq_pipe_ops;
7  + smp_wmb();
8    head += 1;
9  }
10
```

```
11 /******* Thread B *******/
12 /* fs/pipe.c */
13 void pipe_read() {
14   if (head > tail) {
15 +   smp_rmb();
16     buf = &pipe->bufs[tail];
17     len = buf->len;
18     buf->ops->confirm();
19   }
20 }
```

**Figure 1.** OoO bug example in the Linux kernel [31]. In this example, `pipe->bufs` is a ring buffer, and `head` and `tail` denote the head and the tail of the ring buffer. Initially, `head` and `tail` have the same value. In the buggy implementation, memory barriers in both functions were missing (*i.e.*, `#7` and `#15`).

It is worth noting that different processors can reorder memory accesses differently. For example, ARM-based architectures (*e.g.*, Apple Silicon M1 or M2) implement out-of-order execution more aggressively than x86-64 machines. Specifically, in the Apple Silicon M1 architecture, if a thread A changes the variables `x` and `y` in order, a thread B may observe the change of `y` before the change of `x` is visible. The x86-64 architecture, however, always ensures that the thread B observe the change of `x` before the change of `y` is visible to the thread B.

**Memory barrier.** Programs often do not want to reorder memory accesses that have semantical ordering requirements. For example, a program may have an ordering requirement that a store operation `x=1` should be finished and the value of `x` is visible to other threads before another store operation `y=1` is executed (as shown in §2.2).

Then, *memory barriers* are used to enforce ordering requirements. When positioned between instructions, memory barriers ensure that preceding memory accesses (*e.g.*, `x=1`) are finished before subsequent ones (*e.g.*, `y=1`). In the absence of a memory barrier, the processor might reorder memory accesses in contrast to the ordering requirement, breaking the correctness of the program. The Linux kernel provides a number of memory barriers (as shown in Table 1) which can be used for various ordering constraints.

## 2.2 Missing Memory Barriers Cause Concurrency Bugs

A certain reordering of memory accesses can lead to concurrency bugs, which we call *out-of-order bugs* (shortly, OoO *bugs*). Many bug reports show that OoO bugs can cause severe issues in deployed kernels such as denial-of-service [95, 115], system crashes [60, 120], data loss [62], or even well-known security risks such as memory corruptions [17, 82].

Figure 1 shows an example of a real-world OoO bug [31] in the Linux kernel. In this example, `post_one_notification()` initializes an entry in the ring buffer, while `pipe_read()` reads the initialized entry if there exist items to read (`head > tail`). Developers must place memory barriers for these functions to correctly communicate data with the ring buffer. Specifically, 1) `post_one_notification()` should use a write barrier (`#7`[2]) to complete the initialization of an entry before incremented `head` is visible to other cores, and 2) in `pipe_read()`, a read barrier must be placed at `#15` to prevent a processor from executing instructions `#17` and `#18` before executing instruction `#14`.

The both barriers are necessary. Otherwise, architectures that do not preserve the order between store operations (*e.g.*, Apple Silicon M1) lead to an incorrect concurrent execution of the two threads, causing an uninitialized pointer access to `buf` in `pipe_read()`. If `smp_wmb()` is missing, it is possible that in `post_one_notification()`, `head` increases (`#8`) before the function pointer `buf->ops` is initialized to `wq_pipe_ops` (`#6`). In that case, the two functions can access `buf->ops` and `head` in the order of (`#8` → `#14` → `#18` → `#6`). This allows `pipe_read()` to access the *uninitialized* function pointer `buf->ops`, jeopardizing the reliability and security of the kernel. Similarly, if `smp_rmb()` is missing in `pipe_read()`, it can speculatively read the value of `buf->ops` (`#18`) *before* checking the value of `head` (`#14`). Also in this case, `pipe_read()` may access uninitialized `buf->ops` with the execution order of (`#18` → `#6` → `#8` → `#14`).

## 2.3 It Is Challenging to Identify OoO Bugs

It is significantly challenging to pinpoint OoO bugs. As depicted in Figure 1, OoO bugs manifest when two or more threads access shared objects concurrently. Consequently, comprehending OoO bugs requires developers to reason about two distinct types of non-deterministic events: thread interleavings and out-of-order memory accesses. Given that this paper aims to identify OoO bugs in the Linux kernel, which comprises substantial lines of code, manual investigation by developers becomes practically infeasible.

One might contemplate employing existing testing tools (*e.g.*, concurrency fuzzers) to identify OoO bugs. Unfortunately, applying existing tools is impractical for detecting OoO bugs because out-of-order execution inherently contains a *non-deterministic* behavior occurring *inside a processor*. Existing tools assume memory accesses happened

[2] In the real patch, developers used `smp_store_release()` and `smp_load_acquire()` which show the better performance at runtime. For brevity, we use `smp_wmb()` and `smp_rmb()` in this paper.

```
// Original source code
x = 1;
r1 = y;
```

```
// Transformed code
store_value(&x, 1);
r1 = load_value(&y);
```

**Figure 2.** During the kernel compilation, OEMU transforms the original source code (represented in the left) as represented in the right). Note that `r1` is a local variable on the stack.

in order [13, 22, 23, 39, 41–43, 96, 112, 113, 116]. However, to systematically explore potential out-of-order memory accesses, a tool must explicitly govern the behavior of out-of-order execution. To the best of our knowledge, we have not seen an approach that offers a runtime mechanism for controlling out-of-order execution.

Moreover, certain mechanisms for controlling thread interleaving *conceal* the manifestation of OoO bugs. For instance, popular methods involve using a breakpoint [39, 41, 43] or temporarily pausing vCPUs [20, 22, 23]. Once a method is employed to govern thread executions, it imposes an ordered memory access for the thread, eliminating the opportunity to observe arbitrary out-of-order memory access.

**Our approach.** To tame the non-determinism of out-of-order execution, this paper introduces a mechanism called *in-vivo out-of-order execution emulation* (§3) (shortly, OEMU). OEMU presents two noteworthy properties. First, it restores the observability of reordering of memory accesses, while preserving the ability to control thread interleaving of previous studies. For example, in Figure 1, suppose thread A stops after running `#8` due to a breakpoint. Then, OEMU allows us to observe that the value of `head` has been changed, but the value of `buf->ops` has not (*i.e.*, reordering of two store operations at `#6` and `#8`). Second, OEMU allows a userspace program to specify which memory accesses to reorder (*e.g.*, `#6` and `#8` in `post_one_notification()`). During runtime, OEMU enforces the specified reordering memory accesses unless there is a memory barrier between the memory accesses.

With mechanisms to tame non-determinisms of out-of-order execution (*i.e.*, OEMU) and thread interleaving (which is borrowed from previous studies [20, 22, 39]), we propose a fuzzer called OZZ (§4) to identify OoO bugs in the kernel. Specifically, OZZ conducts *a hypothetical memory barrier test*. By assuming a memory barrier is missing in the kernel (*e.g.*, `#7` in Figure 1), it runs memory accesses in the order that the memory barrier is supposed to prevent (*e.g.*, `#8` → `#14` → `#18` → `#6`). Then, it detects if the kernel malfunctions during the runtime with the help of bug-detecting oracles. Furthermore, after identifying an OoO bug, OZZ provides the location of the hypothetical memory barrier and the order of memory accesses that should be prevented, helping developers in comprehending how to fix detected bugs.

## 3 In-Vivo Out-of-Order Execution Emulation

In the architectural viewpoint, the processor may delay committing the value of a store operation to memory, or run a load operation ahead of other instructions that appear earlier in the binary instructions. However, exactly emulating processor's behaviors requires the full architectural simulation, prohibitively expensive when testing the complex kernel code. Instead, we *emulate* out-of-order execution of kernel instructions. To that end, we devise new mechanisms that reorder store and load memory accesses explicitly: *delayed store operation* (§3.1) and *versioned load operation* (§3.2).

**In-vivo emulation.** Controlling out-of-order execution explicitly and deterministically is non-trivial because the execution order of instructions is decided in a processor. Therefore, instead of reordering instructions, we opt to regulate the ordering of memory accesses carried by instructions. During compilation, OZZ replaces memory accesses with callback function calls (as shown in Figure 2). These functions are executed according to the sequential order of their corresponding instructions, while they communicate with OEMU to reorder memory accesses. OEMU is integrated into the kernel, and reorders memory accesses while executing the kernel. We call this approach *in-vivo emulation*.

**Benefits of in-vivo emulation.** When identifying kernel bugs, one needs to comprehend kernel runtime contexts such as what and when memory objects were freed, and what locks a thread has acquired. These runtime contexts are utilized in revealing kernel malfunctions (*i.e.*, bugs) including slab-out-of-bound accesses or deadlocks. As such, various bug-detecting oracles (*e.g.*, sanitizers [66, 69, 75, 88, 91, 93] or lockdep [68]) also leverage runtime contexts in their design.

Previous studies [10, 32, 33, 37, 71, 80], however, are limited in effectively utilizing runtime contexts for identifying kernel bugs. They collect memory accesses after executing system calls, and partially figure out behaviors stemming from out-of-order execution through offline analysis. We call this approach *in-vitro testing*. They lose runtime contexts when analyzing behaviors induced by out-of-order execution, and suffer in revealing kernel bugs (*e.g.*, a double free bug) [82]. In contrast, OEMU actually reorders memory accesses while the kernel is running, enabling use of all bug-detecting oracles which are readily deployed in the kernel.

**Scope of emulation.** Generally, there are four types of out-of-order execution: store-store[3], store-load, load-load, and load-store reordering. Among them, we exclude load-store reordering from the scope of this paper, but support the remaining three types of out-of-order execution. Load-store reordering also can cause a OoO bug theoretically. However, a load-store reordering is rarely implemented in practice [58][4], mainly because it barely provides a hardware optimization chance. We leave load-store reordering as future work.

[3] `A-B` reordering means the former memory access `A` is performed after the latter memory access `B`.

[4] Only two (old) implementations of ARMv8 (*i.e.*, Qualcomm's Snapdragon 820 and Cortex A73) [3, 58] implement load-store reordering, and none of Power hardware [1, 86] does.

| System call interface | Description |
|---|---|
| *delay_store_at(I)* | When an instruction *I* is executed, its store operation will be delayed. |
| *read_old_value_at(I)* | When an instruction *I* is executed, its load operation will read an old value. |

**Table 2.** Two interfaces that OEMU exports to a userspace program. Delayed store operations (§3.1) are implemented using *delay_store_at(I)*, and versioned load operations (§3.2) utilizes *read_old_value_at(I)*, where *I* is an instruction carrying a memory access operation.

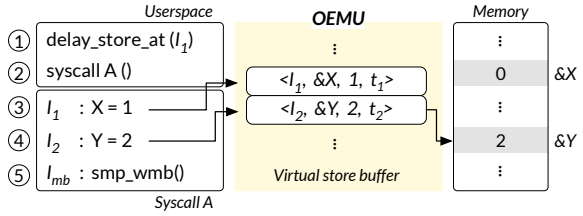



**Figure 3.** Delayed store operation example right after time $t_2$, where $I_1$ and $I_2$ are executed at $t_1$ and $t_2$ respectively. Due to *delay_store_at*($I_1$) (①), the virtual store buffer does not write 1 to $\&X$ when executing $I_1$ (③), and holds the value even after executing $I_2$ (④). Note that the value at $\&X$ will be updated when executing a memory barrier (⑤).

### 3.1 Delayed Store Operation

For store operations, OEMU may delay writing a value of a store operation until a few subsequent instructions are executed. Therefore, the subsequent instructions are executed as if the delayed store operation is not executed, effectively emulating reordering of store-store and store-load instructions.

**Virtual store buffer.** The key component to delay store operations is a *virtual store buffer*, which is a per-thread, temporary storage that holds values of store operations before committing them to memory. When a store operation is performed to a memory location, the updated value are temporarily held in the virtual store buffer first. The virtual store buffer defers committing the value to memory; The updated value held in the virtual store buffer is not visible to threads running in other cores until it is committed to memory.

Unless specifically instructed, the virtual store buffer commits values immediately, meaning it performs in-order execution by default. Two interface exist for a user-space program (*e.g.*, a fuzzer) to specify what operations are delayed. Using the interfaces, one can explicitly control commit orders of store operations to emulate store-store and store-load reorderings. As shown in Table 2, OEMU provides these interfaces as custom system calls. A userspace thread uses these system calls to instruct OEMU to control out-of-order execution during execution of system calls issued by the thread. The virtual store buffer commits all delayed store operations when it meets either of the following conditions: encountering a store, full, or release memory barrier (refer to Table 1) or experiencing an interrupt on the processor executing the thread.

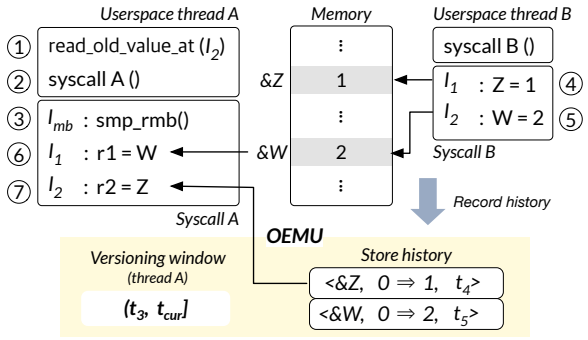



**Figure 4.** Versioned load operation example to reorder $I_1$ and $I_2$. In this example, smp_rmb() is executed at time $t_3$ (③). Then, before Syscall A executes $I_1$, Syscall B writes values to $\&Z$ and $\&W$ at $t_4$ (④) and $t_5$ (⑤), as reflected in the store history. At this point, the versioning window starts from $t_3$ due to the smp_rmb() executed at $t_3$, allowing subsequent load operations to read old values written after $t_3$. Thus, Syscall A can retrieve 0 from the store history for $\&Z$ (⑦), while reading 2 from the memory location at $\&W$ (⑥).

**Example of delayed store operation.** Figure 3 demonstrates how a userspace program communicates with the virtual store buffer. Let us assume a userspace thread wants to reorder two instructions $I_1$ and $I_2$ (i.e., making $I_2$ run before $I_1$). At first, the thread runs the system call, *delay_store_at*($I_1$), to request the virtual store buffer to delay the commit of the value of the instruction $I_1$ (①). Now when running the target system call (②), the virtual store buffer holds the value `1` of `X`, instead of writing it to memory. Thus, the value at `&X` has not been changed even after executing $I_1$ (③), so threads running on the other cores see the old value of 0. In contrast, the virtual store buffer commits `2` to `&Y` immediately when executing $I_2$ (④), as it is the default behavior of the virtual store buffer. At this point, the values at `&X` and `&Y` are `0` and `2`, showing the same effect with a situation that $I_2$ is reordered and executed before $I_1$ is executed. The value of $I_1$ will be flushed into memory when `smp_wmb()` is executed so the virtual store buffer ensures that it does not reorder stores across a store memory barrier (⑤).

**Forwarding values to subsequent loads.** After a thread runs a store operation, subsequent load operations on the same core should read the value from the in-flight store operation in the virtual store buffer if they access the same memory location. Therefore, OEMU performs the following hierarchical search. When the processor runs a load operation, OEMU searches the store buffer first to check that there are any in-flight store operations that took place on the same memory location with the load operation. If so, the load operation reads the value, and otherwise, the load operation reads a value from memory as it does in a normal case.

### 3.2 Versioned Load Operation

A versioned load operation emulates load-load reordering by allowing a load operation to read an *old version* at a memory

location even if its value has been updated in memory. For example, in Figure 4, if the two load operations are reordered in the thread A such that $I_2$ (⑦) is executed before $I_1$ (⑥), it is possible that the values of `r1` and `r2` have `2` and `0` respectively, due to the execution order of ⑦ → ④ → ⑤ → ⑥. This instruction reordering is emulated by making $I_2$ (⑦) read the old value of `0` while making $I_1$ (⑥) read the updated value.

**Store history.** To allow a load operation to read an old value, OEMU constructs a *store history*, which records how values are changed in the past. When the value of a store operation is written to memory, OEMU records the store operation in the global history. Each entry of the history contains various information such as the address on which the store operation writes, the previous value overwritten by the store operation, and the timestamp of when the store operation takes place.

When reading a value, OEMU prioritizes the local store buffer over the store history if both contain a value for the same memory location. This happens when a value was changed in the past, and the thread executing the versioned load operation wrote a new value to that memory location later. Thus, if the store buffer does not contain a value, a versioned load operation reads a value from memory as a default behavior. If a userspace program instructs reading an old value using the interface presented in Table 2, OEMU searches for the store history before reading memory.

**Versioning window.** OEMU maintains a versioning window, a per-thread time window which defines *valid* past values. If a versioning window is from a specific time `t` to the current, versioned load operations can read values that are committed to memory *after* `t` only. Specifically, suppose $t_{cur}$ is the current time, and $t_{rmb}$ is the time when one of a load, full, or acquire memory barrier was most recently executed on this processor. Then a versioning window is defined as $(t_{rmb}, t_{cur}]$. Because any load operations coming after a barrier cannot be reordered before the memory barrier, the versioning window limits a versioned load operation to read an old value only if it is written after the memory barrier.

**Example of versioned load operation.** Figure 4 demonstrates how to reorder load operations. Suppose that a userspace thread wants to reorder two instructions $I_1$ and $I_2$ executed by `syscall A()`. Then, the thread calls *read_old_value_at($I_2$)* (①) before running a target system call (②). During the execution of the system call, a load memory barrier (*i.e.*, `smp_rmb()`) is executed at time $t_3$ (③). Thus, subsequent load operations should not be reordered before the memory barrier. To reflect this, the versioning window becomes $(t_3, t_{cur}]$ after running the load memory barrier. When running $I_1$, the thread reads a value 1 from $\&W$ in memory according to its default behavior (④). However, when running $I_2$ (⑤), the thread reads a value from the store history as instructed. The store history describes that the value of $Z$ has been changed at time $t_4$ by a store operation in an other core. As the versioning window is $(t_3, t_{cur}]$, the thread can read the old value 0 as the value of $Y$ (imagine the processor schedules and runs $I_2$ immediately after time $t_3$). As a consequence, the values of $r1$ and $r2$ are 1 and 0 respectively.

**Dependencies from a load operation to later operations.** Values from which load operations read may affect subsequent memory accesses. For example, if a thread reads a value from `&x`, and write it to `&y`, the processor do not reorder the two memory accesses to correctly determine the value to write to `&y`. To faithfully emulate out-of-order execution, OEMU should not reorder of a load operation and subsequent operations if a processor would not do it. There are two types of dependencies from a load operation to subsequent accesses: load-store and load-load dependencies.

In the case of load-store dependencies, OEMU just does not reorder a prior load and a subsequent store operation, ensuring consistency with the behavior of real processors. Load-load reordering is generally allowed between two load operations, even when a dependency exists, unless the first load is annotated with APIs, such as `READ_ONCE()`, atomic APIs (*e.g.*, `atomic_read()`), or memory barriers [5]. Thus, OEMU treats these APIs as indicating a load memory barrier (*i.e.*, `smp_rmb()`), while still allowing reordering between unannotated loads, regardless of dependencies. Further details can be found in §3.3 and the Appendix (§10.1).

### 3.3 Compliance with the LKMM

As OEMU is to emulate out-of-order execution taken place inside a processor, it should not reorder memory accesses in a way that a processor would never do. However, different architectures define different rules on prohibited reordering cases. To cope with such differences, the reordering rule in OEMU complies with the Linux Kernel Memory Model (LKMM) [2, 94]. LKMM is a unified memory model that the Linux kernel implementation should obey, and provides barrier APIs to prevent reordering of instructions. The LKMM defines seven cases that a processor do not reorder two instructions `X` and `Y`, among which five cases are prohibited by memory barriers and two cases are prohibited by dependencies. Due to the page limit, how OEMU complies with the seven cases are explained in the Appendix (§10.1).

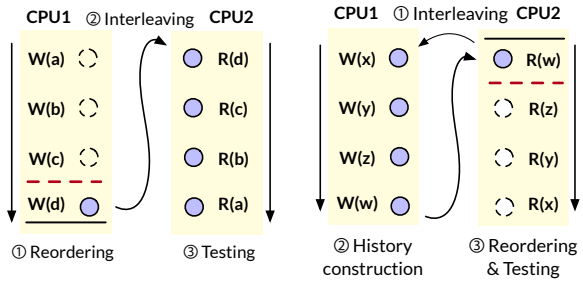
## 4 OZZ

This work proposes OZZ, a kernel fuzzer designed to identify OOO bugs. At its core, OZZ utilizes OEMU to control out-of-order execution as well as adopting a thread interleaving mechanism used in the previous approaches [20, 22, 39].

### 4.1 Hypothetical Memory Barrier Test

The key idea of OZZ is to test the necessity of a hypothetical memory barrier. In other words, when analyzing two concurrent system calls, OZZ assumes a hypothetical memory

[5] Among all architectures supported by the Linux kernel, the Alpha architecture exhibits this behavior.




**(a)** Hypothetical store barrier test through store-store reordering

**(b)** Hypothetical load barrier test through load-load reordering

**Figure 5.** Hypothetical memory barrier test examples. Dotted horizontal lines denote locations of hypothetical memory barriers, and solid horizontal lines denote actual memory barriers (*i.e.*, `smp_wmb()` in (a) and `smp_rmb()` in (b)). Empty circles represent either delayed store operations (W(a)) or versioned load operations (R(x)).

barrier is missing in these system calls, and enforce reordering memory accesses that would not occur if the memory barrier exists. OZZ performs two types of hypothetical memory barrier test: The first type is called *hypothetical store barrier test*. In this test, using a delayed store operation, OZZ emulates store-store or store-load reordering that would not occur if a memory barrier exists. Whereas, the second type is called *hypothetical load barrier test*, as it tests whether a memory barrier that prevents load-load reordering is missing by utilizing a versioned load operation.

Figure 5a shows how OZZ performs the hypothetical store barrier test through store-store reordering. The red dotted line is the location of the hypothetical memory barrier (*e.g.*, `smp_wmb()`), where this memory barrier is supposed to prevent reordering between all preceding store operations (`W(a)`, `W(b)`, and `W(c)`) against the store operation after the barrier (`W(d)`). OZZ examines the impact of the absence of this hypothetical barrier. Thus, OZZ enforces reorderings of instruction after the hypothetical barrier with delayed store operation mechanism. Specifically, ① **Reordering**: OZZ instructs OEMU to delay the three store operations before the hypothetical memory barrier. After that, OZZ requests the custom scheduler to perform interleaving right before the actual memory barrier (*i.e.*, the solid line). ② **Interleaving**: After reordering store accesses, an interleaving happens from CPU1 to CPU2. Note that at this point, the values of `W(a)`, `W(b)`, and `W(c)` have not been committed to memory from the virtual store buffer while that of `W(d)` is committed. Thus, CPU2 observes that `W(d)` was performed before the preceding three store operations. ③ **Testing**: OZZ executes load instructions in CPU2 and monitors whether an OOO bug occurs. If it happens, OZZ reports the bug as well as the location of the hypothetical barrier.

Note that store-load reordering can be performed similarly with store-store reordering. In Figure 5a, assume that CPU1 executes a load operation (*e.g.*, `R(d)`) instead of `W(d)`. Then, OZZ can perform store-load reordering in CPU1 in the same way as when doing store-store reordering.

In addition, Figure 5b shows the hypothetical load barrier test, where a versioned load operation is used to emulate load-load reordering. Suppose the red dotted line is the location of the hypothetical memory barrier (e.g., `smp_rmb()`). ① **Interleaving**: In this test, OZZ first requests the custom scheduler to perform interleaving right after the actual memory barrier (*i.e.*, the solid line). ② **History construction**: The reason for this interleaving is to construct the store history affecting the execution of CPU2 (as explained in §3.2). Thus, OZZ runs all instructions in CPU1 first. ③ **Reordering & Testing**: to test the absence of the hypothetical memory barrier, OZZ reorders `R(w)` and the three following load instructions, `R(z)`, `R(y)`, and `R(x)`. This reordering is done by requesting OEMU to force them to read old values while `R(w)` reads the updated value from memory. Lastly, OZZ executes instructions in CPU2 and monitors if an OOO bug occurs.

**Caveat.** While OZZ can provide the location of the missing memory barrier, it is limited in suggesting an exact type of memory barrier to effectively fix an OOO bug. For example, in Figure 1, the bug does not manifest if both functions adopt strong memory barrier (*i.e.*, `smp_mb()`) which unnecessarily incur high runtime overhead. After OZZ points out a location of a missing memory barrier, developers need to take a look at what kind of memory barrier is proper to fix the OOO bug.

**Workflow.** For conducting the hypothetical memory barrier test, the workflow of OZZ (shown in Figure 6) consists of three steps: At the first step, OZZ generates and runs *single-threaded inputs*, each of which consists of a sequential set of system calls. In addition, OZZ profiles memory accesses and memory barriers executed by each single-threaded input (§4.2). With profiled memory accesses and memory barriers, OZZ repeatedly conducts the hypothetical memory barrier test. For each test run, OZZ calculates a *scheduling hint*, which describes in what instructions interleaving occurs and what memory accesses are reordered (§4.3). Lastly, OZZ constructs and runs *multi-threaded inputs*. Specifically, a multi-threaded input is annotated with a scheduling hint as well as a pair of two system calls to run concurrently. OZZ runs them while performing interleaving and reordering specified in the scheduling hint, and observes if they cause an OOO bug or not using kernel bug-detecting oracles (§4.4).

### 4.2 Profiling Memory Accesses & Barriers

The first step of OZZ is similar with traditional fuzzers. It constructs single-threaded inputs (referred to as STIs) each of which is a sequence of random system calls $(S_1, S_2, ..., S_n)$. In the construction, OZZ uses predefined templates written in `Syzlang` [24], which is a language to describe available system calls as well as possible values of arguments and return values. Based on the templates, OZZ produces *valid* STIs which preserve necessary resource dependencies across

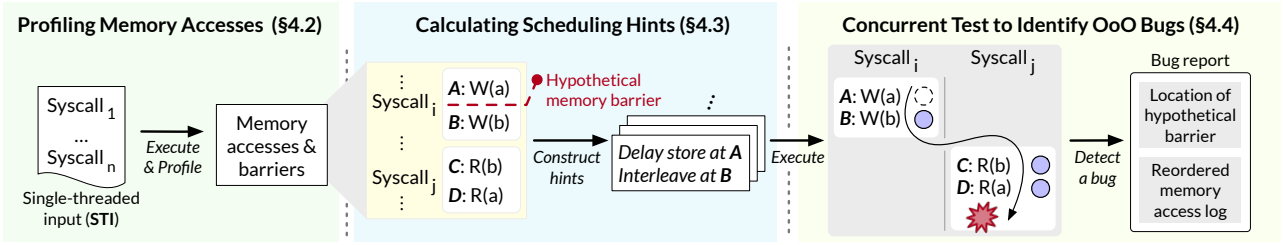



**Figure 6.** Workflow overview of OZZ.

system calls; *e.g.*, get a file descriptor from `open` and use it for `write`. OZZ expands code coverage with the support of the kernel (*i.e.*, `KCov` [67]), while monitoring whether non-concurrency bugs occur with bug-detecting tools deployed in the kernel (*e.g.*, KASAN [66, 88], and `lockdep` [68]).

While running the STIs, OZZ dynamically profiles memory accesses and memory barriers executed by each input. To this end, OZZ incorporates a LLVM compiler pass, which inserts callback function calls before memory-accessing instructions and memory barriers. The callback function records various information on memory accesses as five tuples: addresses of the instruction, the accessed memory location, the size and the type (*i.e.*, store or load) of the memory access, and the timestamp. When encountering memory barriers, three tuples are recorded: the instruction address, the type of the memory barrier (refer to Table 1), and the timestamp. This information is recorded in a per-thread memory region, which is shared with a userspace program through `mmap()`. OZZ uses this information to calculate scheduling hints in the next step.

### 4.3 Calculating Scheduling Hints

After profiling memory accesses and memory barriers for a single-threaded input $(S_1, S_2, ..., S_n)$, OZZ computes a set of scheduling hints $H_{ij}$ for each pair of $S_i$ and $S_j$. A scheduling hint consists of two information, a scheduling point to perform an interleaving, and memory accesses to reorder. In the next phase (§4.4), OZZ will run $S_i$ and $S_j$ concurrently while running memory accesses according to each scheduling hint.

**Search heuristic.** When conducting the hypothetical memory barrier test, OZZ adopts a greedy heuristic, which prioritizes the test case that maximizes the number of reordered memory accesses, the number of delayed store operations or the number of versioned load operations. The rationale behind this heuristic is as follows: developers typically perceive memory accesses as occurring sequentially. When the degree of deviation from this sequential execution order is significant, developers are more likely to overlook the necessity of inserting a memory barrier, as there is a discrepancy between their intuition and the actual execution. Therefore, OZZ prioritizes tests where as many memory accesses as possible deviate from their sequential order. We validate that our heuristic works empirically with our evaluation bug set. 11 out of the 19 bugs in the bug set are triggered with scheduling hints that maximize the number of reordered memory accesses, and 6 are triggered with scheduling hints with the second largest number of reordered memory accesses.

**Algorithm 1:** Calculating scheduling hints

**Input** : $S_i, S_j$: Sequences of memory access and memory barriers executed by two system calls
**Output** : $H_{ij} = \{h_1, h_2, ..., h_n\}$: A set of scheduling hints

```
▷ Step 1: Filter out memory accesses
S_i, S_j = filter_out(S_i, S_j)
for k ∈ {i, j} do
    for barrier_type ∈ {st, ld} do
        ▷ Step 2: Group memory accesses between
        memory barriers of the same type
        G_t, g = ∅, ∅
        for s ∈ S_k do
            if s is a memory access then
                g = g ∪ {s}
            else if s is a barrier &
            type of s = barrier_type then
                G_t = G_t ∪ {g}
                g = ∅
        ▷ Step 3: Construct scheduling hints
        H_ij = ∅
        for g ∈ G_t do
            if barrier_type = st then sched = g.last
            else sched = g.first
            while g ≠ ∅ do
                h.sched = sched
                h.reorder = g \ sched
                H_ij = H_ij ∪ {h}
                if barrier_type = st then g = g \ {g.last}
                else g = g \ {g.first}
    H_ij.sort(key : len(h.reorder))
return H_ij
```

**Algorithm description.** Algorithm 1 describes an algorithm to construct scheduling hints for two system calls $S_i$ and $S_j$. The algorithm consists of three steps. The first step of the algorithm is to filter out memory accesses that are irrelevant to OOO bugs. Since OOO bugs are a type of concurrency bug, memory accesses that do not access shared variables do not contribute to the manifestation of an OOO bug. Thus, OZZ filters out such memory accesses at its first step (#1). Since

this is straightforward, we leave the algorithm of `filter_out()` function in the Appendix (Algorithm 2).

Afterwards, OZZ calculates scheduling hints for four cases, depending on which system call will reorder memory accesses (#2), and what types of hypothetical memory barrier test OZZ will conduct (#3). For each case, OZZ performs the second step, which groups memory accesses with boundaries of memory barriers. While iterating over memory accesses and memory barriers (#5 - #11), OZZ collects memory accesses into a group $g$ (#6 - #7). If OZZ meets a memory barrier that has the same type (#8 & #9), OZZ finalizes $g$ (#10) and starts building a new group (#11).

In the third step, OZZ computes scheduling hints for each group (#13 - #21). If $g$ is for the hypothetical store barrier test, OZZ selects a scheduling point as the last instruction (#14), whereas the first instruction is selected as a scheduling point if $g$ is for the hypothetical load barrier test (#15). Then, OZZ constructs scheduling hints for the group $g$ (#16 - #21). At first, OZZ constructs a scheduling hint that reorders all memory accesses in the group $g$ (#17 - #18). After that, OZZ moves a hypothetical barrier by one instruction. If $g$ is for the hypothetical store barrier test, OZZ moves the hypothetical barrier upward by one instruction (#20), and if it is for the hypothetical load barrier test, OZZ moves the hypothetical barrier downward by one instruction (#21). OZZ repeats the construction of scheduling hints until $g$ becomes empty.

Lastly, OZZ sorts scheduling hints according to the number of memory accesses that will be reordered. This is to prioritize scheduling hints that reorder as many as memory accesses possible, according to the search heuristic mentioned above.

### 4.4 Concurrent Test to Identify OoO Bugs

The last step of OZZ is to run reordered and interleaved memory accesses according to scheduling hints calculated in §4.3.

**Constructing MTIs.** After calculating scheduling hints, OZZ translates STIs to multi-threaded inputs (MTIs) to identify OOO bugs. Each STI is translated into multiple MTIs each of which consists of the same set of system calls with the STI. In addition, MTIs are annotated with a pair of system calls to run concurrently, as well as a scheduling hint between the two system calls. As shown in Algorithm 1, each hint contains a scheduling point (*i.e.*, `h.sched`), and a set of memory accesses to reorder (*i.e.*, `h.reorder`).

**Running MTIs.** After constructing MTIs, OZZ runs MTIs to monitor whether the results of MITs are kernel OOO bugs or not. Specifically, OZZ sends `h.sched` as an input to the custom scheduler (which is explained in §4.4.1), and `h.reorder` as an input to OEMU. As shown in Figure 5, these two mechanisms take control of out-of-order execution and thread interleaving respectively, and collaborate together to trigger an OOO bug. During the runtime of each MTI, OZZ leverages various bug-detecting oracles such as lockdep [68], KASAN [66], or manually inserted assertions. If an OOO bug is detected, OZZ files up a report of memory accesses that were reordered as well as the hypothetical memory barrier that was used to construct the scheduling hint. Developers can utilize the report to comprehend in what order memory accesses were executed in causing the OOO bug.

#### 4.4.1 Custom Scheduler.

In addition to OEMU, OZZ needs a mechanism to deterministically control thread interleaving. Thus, OZZ borrows a mechanism from previous studies [20, 22, 39, 41], which we call a *custom scheduler*. The custom scheduler is implemented in the hypervisor layer to override the guest kernel's scheduler, and controls thread interleaving. It accepts an input as scheduling points (*i.e.*, `h.sched`) through hypercall interfaces. As the custom scheduler is not a primary focus of this paper, we leave details of the custom scheduler in the Appendix (§10.3).

### 4.5 Discussions

**Applicability to Rust.** Linux recently adopted Rust as the second official programming language [84, 102]. Interestingly, OOO *bugs can occur in kernel modules written in Rust, and OEMU and* OZZ *is applicable to them.* Figure 10 in the Appendix (§10.4) shows a *synthetic* example of an OOO bug written in Rust. We confirm that OEMU can trigger the OOO bug in the example. Note that OEMU is implemented through an LLVM pass, and it is applicable to all languages, including Rust, that are translated to LLVM IR. Although the mainline branch of the Linux kernel does not include modules written in Rust at this point, we expect that Linux will increasingly incorporate such modules, and OZZ and OEMU will be helpful in identifying OOO bugs in them.

**Concurrent accesses with hardware.** We observe that OOO bugs can also occur between a kernel thread and hardware. As an example, we found a patch to fix an OOO bug in the RDMA module [85]. It addressed an issue caused by reordering of two load operations on values *written by hardware*. In this case, OEMU can emulate load-load reordering in the device driver. So, we believe that if we run the device driver with a proper hardware, we can trigger the OOO bug with OEMU. To this end, we may need further research, as a fuzzer needs to know which instructions are shared with hardware and are subjects of out-of-order execution.

**Limitations.** While OZZ is capable of detecting concurrency bugs related to missing memory barriers, it has some limitations. First, OZZ is limited in detecting performance bugs caused by redundant memory barriers, as its primary focus is on identifying missing barriers. Since these performance bugs are also critical in highly concurrent implementations, we leave this direction for future work.

Second, due to the OZZ 's design, which tests a single hypothetical memory barrier at a time, it is not effective in detecting OOO bugs that manifest only when memory accesses are reordered across two or more threads. However, we have

not encountered any bugs of this nature in practice, and we believe that such cases are rare.

Lastly, OZZ and OEMU are limited in exposing OOO bugs caused by the reordering of a prior load operation and a subsequent store operation. While this type of OOO bugs may happen theoretically, a recent study [58] shows that load-store reordering is rarely implemented, making these bugs unlikely to occur in practice.

## 5 Implementation

**OEMU.** OEMU consists of two parts, a compiler pass and callback functions. We implement the compiler pass based on the LLVM compiler 12.0.1 [70] with 834 LoC[6] in C++. The callback functions reside in the Linux kernel source tree with 1797 LoC in C. These two parts automatically transplant OEMU into the kernel binary during the kernel compilation.

**OZZ.** We implement OZZ based on SYZKALLER [25], a state-of-the-art kernel fuzzer developed by Google. OZZ is implemented with 7694 LoC in Go and 913 LoC in C++. While the custom scheduler is not a major part of this paper, it is implemented in QEMU [83] with 2167 LoC in C.

## 6 Evaluation

In order to verify the effectiveness of OZZ, we conduct several evaluation. (1) We run OZZ to discover unknown kernel OOO bugs (§6.1), (2) verify that OEMU can reproduce previously reported OOO bugs (§6.2), (3) evaluate whether OZZ provides reasonable runtime overheads (§6.3), and (4) compare OZZ to related work (§6.4).

### 6.1 Finding Real-world OOO bug

**Experiment setup.** We use a two-socket machine equipped with Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz (32 physical cores) and 512 GB of RAM. The host operating system is Ubuntu Server 20.04.4 LTS on Linux 5.4.143 with the modified KVM module to allow hypercalls from the userspace. We configure OZZ to launch 32 virtual machines (VMs) where each VM is equipped with four virtual CPUs and 8 GB memory. Fuzzing performance can vary depending on the seed used and the target being fuzzed [56]. In this evaluation, we use Linux kernel configurations used by SYZKALLER [25] so that SYZKALLER and OZZ explore the same kernel subsystems. We run OZZ in recent kernel versions ranging from 6.5-rc6 to 6.8 over a 6-week period, and we use seeds provided by SYZKALLER [26].

**Newly found OOO bugs.** During our evaluation, OZZ discovers 61 unique crashes titles, including crashes that `Syzkaller` also found. After OZZ discovers crashes, we manually identified ones that are caused by out-of-order execution. Consequently, 11 are identified as ones caused by out-of-order execution as demonstrated in Table 3.

[6]We use cloc [16] to measure LoC of various languages.

```
1  /******* Thread A *******/
2  /* net/tls/tls_main.c */
3  int tls_init() {
4    ctx = kzalloc();
5    sk->data = ctx;
6    ctx->sk_proto =
7      READ_ONCE(sk->sk_prot);
8  + smp_wmb();
9    WRITE_ONCE(sk->sk_prot,
10             &tls_prots);
11 }
12
13 struct proto_ops tls_prots = {
14  .setsockopt = tls_setsockopt,
15 };
```

```
16 /******* Thread B *******/
17 /* net/core/socket.c */
18 int sock_common_setsockopt() {
19   struct sock *sk = sock->sk;
20   return READ_ONCE(sk->sk_prot)
21            ->setsockopt(sk);
22 }
23
24 /* net/tls/tls_main.c */
25 int tls_setsockopt() {
26   struct tls_context *ctx =
27     sk->data;
28   return ctx->sk_proto
29            ->setsockopt(sk);
30 }
```

**Figure 7.** Simplified code snippet of the OOO bug found in the TLS subsystem (Bug #9). Developers missed the possibility of reordering between accesses on `ctx->sk_proto` (#6) and `sk->sk_prot` (#9). Therefore, a NULL pointer dereference bug may occur at #28 if two memory accesses on #6 and #9 are reordered.

It is worth noting that these bugs are found in popular subsystems such as the TLS network subsystem (Bug #5 and #9), the general notification mechanism (Bug #2), and the BPF subsystem (Bug #6). Furthermore, we emphasize that it is impractical to identify these bugs with SYZKALLER unless one utilizes plenty of ARM machines. In practice, Google invests significant computing power in continuously running SYZKALLER with x86-64 machines. However, all these bugs do not manifest on x86-64 machines because the x86-64 architecture does not reorder store-store and load-load operations (refer to the `Type` column). Google attempts to identify bugs that manifest on ARM machines with the TCG instruction emulation. However, this approach is insufficient because TCG does not reorder memory accesses.

**Case study 1: Improper adoption of memory barriers.** During the evaluation, we found a case that developers recognized a buggy concurrent execution, but wrote an incorrect patch afterwards (Bug #9). Figure 7 shows the case. In this case, Thread A initializes the TLS context for a socket (*i.e.*, calling `tls_init()`), while Thread B runs the `setsockopt()` system call on the socket. During the initialization, Thread A allocates `ctx` and initializes it (#4 ~ #7). Then, Thread A replaces function pointers of the socket (*i.e.*, `sk->sk_prot`) to ones for TLS (#9). Thus, if Thread B sees the replaced function pointers (#20), then `tls_setsockopt()` is expected to perceive initialized `ctx->sk_proto` at #28. However, because a store memory barrier was missing at #8, Thread A might be replace `sk->sk_prot` (#9) before initializing `ctx->sk_proto` (#6). As a result, `tls_setsockopt()` may see uninitialized `ctx->sk_proto` (#28), causing the OOO bug with the execution order of (#9 → #20 → #28 → #6).

Interestingly, developers once recognized a data race on `sk->sk_prot` [36, 89]. However, they did not catch the possibility of out-of-order execution, and incorrectly fixed the data race. Specifically, developers annotate memory accesses at #9 and #20 with `WRITE_ONCE()` and `READ_ONCE()`, which do not prevent reordering with other memory accesses, but

| ID | Kernel version | Subsystem | Summary | Status |
|---|---|---|---|---|
| Bug #1 | v6.7-rc8 | RDS | KASAN: slab-out-of-bounds Read in rds_loop_xmit | Fixed |
| Bug #2 | v6.5-rc6 | watchqueue | BUG: unable to handle kernel NULL pointer dereference in _find_first_bit | Reported |
| Bug #3 | v6.5-rc6 | VMCI | general protection fault in add_wait_queue | Reported |
| Bug #4 | v6.6-rc2 | XDP | BUG: unable to handle kernel NULL pointer dereference in xsk_poll | Fixed |
| Bug #5 | v6.6-rc2 | TLS | BUG: unable to handle kernel NULL pointer dereference in tls_getsockopt | Fixed |
| Bug #6 | v6.7-rc8 | BPF | BUG: unable to handle kernel NULL pointer dereference in sk_psock_verdict_data_ready | Fixed |
| Bug #7 | v6.5-rc7 | XDP | BUG: unable to handle kernel NULL pointer dereference in xsk_generic_xmit | Fixed |
| Bug #8 | v6.7-rc8 | SMC | BUG: unable to handle kernel NULL pointer dereference in connect | Confirmed |
| Bug #9 | v6.7-rc2 | TLS | BUG: unable to handle kernel NULL pointer dereference in tls_setsockopt | Fixed |
| Bug #10 | v6.8-rc1 | SMC | KASAN: null-ptr-deref Write in fput | Confirmed |
| Bug #11 | v6.8 | GSM | BUG: unable to handle kernel NULL pointer dereference in gsm_dlci_config | Confirmed |

**Table 3.** List of concurrency bugs newly discovered by OZZ.

suppress a data race detector (*i.e.*, KCSAN) from reporting on the data race. Consequently, this OOO bug has not been fixed even after the patch is applied. This case underscores that comprehending the possibility of out-of-order execution is truly challenging and error-prone, while OZZ offers a great potential to enhance their understanding of it.

**Case study 2: Incorrect customized lock.** We also found a customized lock implementation that misses a memory barrier, breaking the mutual exclusion (Bug #1). Figure 8 shows the incorrect lock implementation. In this example, `acquire_in_xmit()` acts as a try-lock function, so a caller checks the return value of this function to determine that it has successfully acquired the lock. On the other hand, calling `release_in_xmit()` releases the lock. These two functions are implemented by using atomic bit operations.

`clear_bit()`, however, does not prevent a reordering between the bit modification of `cp_flags` and instructions issued before `clear_bit()`. Consequently, when a thread is releasing a lock by calling `release_in_xmit()`, it is possible that memory accesses inside the critical section are reordered to be executed after clearing the bit, breaking the mutual exclusion. The correct implementation of the lock is using `clear_bit_unlock()` in `release_in_xmit()`. This variation of a bit operation ensures that all memory accesses issued before `clear_bit_unlock()` are finished before modifying the bit. We figure out that this bug is hardly detectable using existing approaches. For example, data race detectors suffer from detecting it as the lock implementation does not contain a data race. In contrast, OZZ can figure out the bug by actually reordering memory accesses inside the critical section and the bit modification on `cp_flags`.

```
1  /* net/rds/send.c */
2  int acquire_in_xmit()
3  {
4    int acquired =
5      !test_and_set_bit
6        (IN_XMIT, &cp_flags);
7    return acquired;
8  }
```

```
9  /* net/rds/send.c */
10 void release_in_xmit()
11 {
12 - clear_bit(IN_XMIT, &cp_flags);
13 + clear_bit_unlock(IN_XMIT,
14                    &cp_flags);
15 }
16
```

**Figure 8.** Incorrect customized lock implementation found by OZZ (Bug #1). In `release_in_xmit()`, `clear_bit_unlock()` should be used instead `clear_bit()` to prevent reordering between the bit modification and instructions inside the critical section.

## 6.2 Reproducing Known OOO bugs

We evaluate whether OEMU has the strong capability in controlling out-of-order execution by reproducing previously reported OOO bugs.

**Method.** There does not exist a publicly available benchmark for this evaluation, we adopt our best-effort approach in building a benchmark of OOO bugs. We first collect patches that addressed OOO bugs from the git history of Linux. From the git history, we collect patches as follows: 1) we first filter patches by keyword matching with a few keywords including *"missing.*barrier"*, *"reordering"*, *"out of order"*, and *"out-of-order"*, and then 2) we manually inspect that the patches are written to fix OOO bugs by checking their descriptions and whether they added memory barriers.

After collecting patches, we build inputs (*i.e.*, sets of system calls) that can potentially trigger OOO bugs if the collected patches are reverted. To this end, we exploit a rich set of inputs that has been collected by SYZKALLER for a few years. From the dashboard of SYZKALLER [27], we check that SYZKALLER has explored locations of memory barrier that the collected patches added for fixing OOO bugs. If the memory barrier is reachable from SYZKALLER, we extract inputs from the dashboard to figure out system calls that can reach the locations of added memory barriers, and revert patches to introduce OOO bugs. Note that this collection step has a similar effect with running OZZ for a long time, as the first step (§4.2) is just a slight modification of SYZKALLER. Lastly, by providing collected inputs as single-threaded inputs, we checked whether OZZ can trigger OOO bugs or not.

**Result.** The Table 4 shows the result. As shown in this table, OEMU can reproduce eight among nine OOO bugs. Among eight OOO bugs that OEMU can reproduce, five OOO bugs can be reproduced by store-store reordering, while three OOO bugs can be reproduced by load-load reordering. In addition, given single-threaded inputs, OZZ reproduce those bugs by running within tens of test runs on average, showing that the OEMU capability of controlling out-of-order execution is strong enough to reproduce most of reported OOO bugs.

| ID | Subsystem | Version | Reproduced? | # of tests | Type |
|---|---|---|---|---|---|
| #1 [120] | vlan | 5.12-rc7 | ✓ | 342 | S-S |
| #2 [31] | watchqueue | 5.17-rc7 | ✓ | 23 | S-S |
| #3 [103] | xsk | 4.17-rc4 | ✓ | 47 | S-S |
| #4 [101] | xsk | 5.3-rc3 | ✓ | 12 | S-S |
| #5 [30] | fs | 6.1-rc1 | ✓ | 17 | L-L |
| #6 [60] | sbitmap | 5.1-rc1 | × | - | S-S |
| #7 [78] | nbd | 6.7-rc1 | ✓ | 17 | L-L |
| #8 [50] | tls | 6.7-rc1 | ✓* | 42 | S-S |
| #9 [106] | unix | 5.0-rc7 | ✓ | 23 | L-L |

**Table 4.** List of previously-reported OoO bugs. In the `Reproduced?` column, ✓* (*i.e.*, #8) means that OZZ can run the buggy reordering case, but the symptom is not a system crash. (*i.e.*, its consequence is returning a wrong value to a system call).

Among all bugs, however, we faced one bug that OEMU cannot reproduce, while it caused a kernel crash in a deployed kernel [60]. We analyzed the reason of the reproduction failure, and figured out that this is because of thread migration, another kernel behavior that we do not consider. Specifically, this bug resided in the multi-queue (MQ) block layer [6]. This bug is tricky to trigger because it is a concurrency bug on a *per-cpu variable*. In our understanding, triggering this bug requires a few conditions: 1) Initially, two threads run on the same CPU, 2) the two threads get an address of a per-cpu variable on the CPU, 3) one thread is migrated to another CPU, causing the two threads to run concurrently on different CPUs, and lastly, 4) two threads run memory accesses in a specific order. After that, the OoO bug manifests. However, OZZ pins concurrent threads on specific CPUs before executing system calls, limited in satisfying the above conditions. To verify the above analysis, we slightly modified the kernel code to satisfy the above condition. Specifically, we enforce specific two threads to get an address of the per-cpu variable from the same CPU even if they are running on different CPUs, and checked whether OZZ can trigger this bug with the manual information. Consequently, we confirmed that OZZ can reproduce the bug successfully.

### 6.3 Measuring Performance Overhead of OEMU

OEMU provides an ability to control out-of-order execution at the cost of runtime overheads. To evaluate these overheads, we conduct two evaluations, overheads of OZZ in microbenchmarks (§6.3.1), and throughput degradation of OZZ (§6.3.2).

**6.3.1 Microbenchmark.** We measure runtime overheads using LMBench [74], a benchmark suite designed for evaluating various OS operations. We conduct tests on two Linux kernels, one with OEMU instrumentation enabled and one without, taking the average of five measurement runs for each.

Table 5 demonstrates results of measurements. As shown in the table, OEMU incurs relatively high overheads, ranging from 3.0× to 59.0×, mainly because of the heavy instrumentation. These high latencies directly impact on the OZZ's performance, as OZZ frequently invokes system calls. Nonetheless,

| Tests | Linux (µs) | Linux w/ OEMU (µs) | Overhead |
|---|---|---|---|
| null | 1.74 | 43.3 | 24.9× |
| stat | 75.64 | 859.6 | 11.4× |
| open/close | 128 | 1369.2 | 10.7× |
| File create | 403.3 | 5623.5 | 13.9× |
| File delete | 207.8 | 3363 | 16.2× |
| ctxsw 2p/0k | 23.8 | 71.5 | 3.0× |
| pipe | 59.3 | 610.1 | 10.3× |
| unix | 173.8 | 2567.6 | 14.8× |
| fork | 7590 | 145.6k | 19.2× |
| mmap | 133.8k | 7896.1k | 59.0× |

**Table 5.** LMBench microbenchmark results.

as shown in §6.1, OZZ has enough capability to find real-world OoO bugs. We further argue that we can reduce the runtime overheads with developers's expertise. Since OoO bugs mostly occur in lockless implementations (unless lock APIs are incorrectly implemented), developers may opt to *selectively* enable the OEMU instrumentation for submodules that heavily rely on lockless implementations. This will result in a small set of submodules in which OEMU is enabled, and accordingly, reduced runtime overheads.

**6.3.2 Throughput Measurement.** To further comprehend the impact of runtime overheads shown in §6.3.1, we measure the fuzzing throughput of OZZ and SYZKALLER, which is a baseline of the OZZ implementation. Because OEMU is compiled into the kernel, we compile another kernel binary without OEMU for measuring the throughput of SYZKALLER. As a result, we confirm that the throughput of OZZ is 0.92 tests/s, while the throughput of SYZKALLER is 7.33 tests/s; OZZ shows 7.9× lower throughput compared to SYZKALLER. It is because OZZ incurs extra jobs in several layers, such as memory copies in the userspace, and VM operations in the hypervisor layer. Note that the runtime overheads in Table 5 only apply to the kernel under test.

This decrease in throughput can be understood from two perspectives. First, at the expense of low throughput, OZZ gains the ability to control out-of-order execution. Since OoO bugs manifest only with a specific and subtle order of memory accesses, blindly running system calls is undesirable when identifying OoO bugs. Second, OZZ allows out-of-order execution fuzzing with machines of stronger memory models (*e.g.*, x86-64 machines) to identify OoO bugs that manifest under weaker memory models (*e.g.*, OoO bugs that manifest in ARM machines). Without OZZ, developers seeking for OoO bugs in ARM machines would need to purchase a fleet of ARM machines. Considering a huge computing power is investigated to identify bugs, they may need to pay the high cost. In that case, developers can take advantages of OZZ instead to save the cost of buying new machines.

### 6.4 Comparing with OFence [61]

To the best of our knowledge, OFence [61] is the state-of-the-art work to discover OoO bugs in the kernel. OFence predefines likely-buggy patterns based on an observation

that memory barriers are usually used in pair. (*e.g.*, in Figure 1, `smp_wmb()` and `smp_rmb()`). It finds out potential OoO bugs through *static* pattern matching analysis.

We first evaluate whether OZZ can detect OoO bugs found by OFence. To this end, we conduct a similar evaluation done in §6.2. In particular, from the git history of Linux, we collected patches that are annotated by the `Reported-by` tag with the authors' names of the OFence paper. We also asked the authors for patches we missed. For those patches, we evaluate if OZZ identifies these bugs or not. Unfortunately, we figure out that all patches addressed submodules in which OZZ (and SYZKALLER) is limited in generating inputs to test. One notable reason is that a submodule requires specific hardware to run, inhibiting dynamic testing in such submodules. In addition, we also evaluate whether OFence can identify OoO bugs found by OZZ. Since we could not access the source code of OFence, we count the number of OoO bugs in Table 3 that do not fall into predefined patterns of OFence. As a result, 8 out of 11 are hardly detectable by OFence.

This comparison highlights distinct advantages and disadvantages of OZZ and OFence. OZZ does not rely on predefined buggy patterns, providing general capability in identifying OoO bugs. However, it may face limitations when testing device drivers that require specific hardware to run. Conversely, OFence is a static analysis tool, which is capable of detecting OoO bugs without the need to execute a target implementation. Nevertheless, it relies on predefined patterns to avoid excessive false positives, limiting its scope of bug detection to the specific patterns.

## 7 Related work

**In-Vitro Out-of-Order Execution Testing.** It is hard to control out-of-order execution at runtime. To workaround the difficulty, previous approaches [10, 12, 19, 21, 32, 33, 35, 37, 49, 57, 71, 80] adopt in-vitro testing approaches, which analyzes behaviors caused by out-of-order execution offline. However, when applying them to the kernel, they cannot consider the kernel runtime context, limiting scope of identifying OoO bugs such as double-free bug. On the other hand, OZZ is designed to monitor the kernel behavior according out-of-order execution, showing significantly better capability in identifying kernel OoO bugs.

**Concurrency-aware Fuzzing.** Fuzzing [28, 29, 38, 51–55, 72, 90, 92, 97, 98, 108, 114, 117] has gained the popularity due to its excellent bug-finding capability in large system software. While conventional fuzzers focus on exploring execution paths, recent approaches [13, 22, 23, 39, 41–43, 112, 113], called concurrency fuzzing, have been proposed to identify concurrency bugs with the idea that thread interleaving is a subject of exploration and is controllable [9, 20, 40, 64, 76, 77, 99]. Unfortunately, controlling thread interleaving only is not sufficient to discover OoO bugs and they often hide effects of memory access reordering by using the breakpoint mechanism. OEMU extends previous work to identify OoO bugs. It enables control knobs of memory access reordering, while preserving the capability of controlling thread interleaving.

**Data Race Detector.** A large body of work [7, 11, 18, 65, 73, 79, 81, 87, 105, 118, 119] has been proposed to detect data races, which are a subset of concurrency bugs [2, 94]. While they might be helpful in identifying OoO bugs, OEMU provides stronger benefits. For example, most of data race detectors fall short in comprehending what memory accesses should not be reordered and what will be the result of reordering, while OZZ controls what memory accesses are reordered to explore the search space of kernel OoO bugs.

In the case of the Linux kernel, KCSAN [65] partially models the behavior of weak memory models. It samples memory accesses, rather than actually reordering them, and determines if a data race could occur under weak memory models. However, OZZ and OEMU offer notable advantages 1) KCSAN is limited to analyzing the delay of a single memory access not annotated with an API in Table 1, whereas OZZ can reorder multiple loads, stores, and atomic operations, regardless of annotations, showing the stronger reordering capability; 2) OZZ can reorder accesses across function boundaries, while KCSAN could not (e.g., Bug #5 in Table 3, and #3, #6 in Table 4); and 3) KCSAN's behavior is non-deterministic, while OZZ and OEMU provide deterministic control over out-of-order execution. Thus, OZZ can explicitly test and reproduce reordered instruction sequences as needed, enabling more efficient search and testing.

## 8 Conclusion

We believe this work marks the beginning of the journey into fuzzing for OoO bugs. As cloud and mobile systems increasingly adopt ARM-based architectures, hardening systems against OoO bugs will become increasingly important. In response to this trend, our work presents a new and practical fuzzing approach for discovering OoO bugs through in-vivo emulation of out-of-order execution. In evaluation, OZZ identifies 11 previously unknown OoO bugs. In addition, we demonstrate that OEMU is effective in reproducing previously reported OoO bugs, which are hardly detectable with previous approaches.

## 9 Acknowledgment

We greatly appreciate the anonymous reviewers and our shepherd, Aurojit Panda, for their constructive comments and feedback. This work was supported in part by NRF (RS-2024-00347516 and RS-2024-00359979), IITP (RS-2023-00232728), TIPA (TIPS 00262147), K-Startup (20144069) and Samsung Electronics. This work also was supported by the Korea Institute of Science and Technology Information (KISTI) in 2024 (No.(KISTI) K24L4M2C5), aimed at developing KONI (KISTI Open Natural Intelligence), a large language model specialized in science and technology.

## References


[1] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2): 1–74, 2014.

[2] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[3] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 43 (2):1–54, 2021.

[4] S. Andrew Waterman, Krste Asanović. The RISC-V Instruction Set Manual, 2017. https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf.

[5] ARM Holdings. Arm® Architecture Reference Manual for A-profile architecture, 2022. https://developer.arm.com/documentation/ddi0487/latest.

[6] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, 2013.

[7] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[8] N. Borisov. btrfs: Fix deadlock caused by missing memory barrier, 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6e7ca09b583de4be6c27d9d4b06e8c5dd46a58fa.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2010.

[10] J. Burnim, K. Sen, and C. Stergiou. Testing concurrent programs on relaxed memory models. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Toronto, Canada, July 2011.

[11] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle, WA, Nov. 2016.

[12] M. Cao, J. Roemer, A. Sengupta, and M. D. Bond. Prescient memory: exposing weak memory model behavior by looking into the future. *ACM SIGPLAN Notices*, 51(11):99–110, 2016.

[13] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, Aug. 2020.

[14] T. W. Christopher. Reference count garbage collection. *Software: Practice and Experience*, 14(6):503–507, 1984.

[15] Compaq Computer Corporation. Alpha Architecture Reference Manual, 2002. https://download.majix.org/dec/alpha_arch_ref.pdf.

[16] A. Danial. cloc, 2020. https://github.com/AlDanial/cloc.

[17] E. Dumazet. tcp: add a missing barrier in tcp_tasklet_func(), 2016. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0a9648f1293966c838dc570da73c15a76f4c89d6.

[18] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective Data-Race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[19] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[20] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[21] M. Gao, S. Chakraborty, and B. K. Ozkan. Probabilistic concurrency testing for weak memory programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, Canada, Mar. 2023.

[22] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[23] S. Gong, D. Peng, D. Altınbüken, P. Fonseca, and P. Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, Oct. 2023.

[24] Google. Syscall description language, 2016. https://github.com/llvm-mirror/llvm/blob/master/lib/Transforms/Instrumentation/SanitizerCoverage.cpp.

[25] Google. Syzkaller - kernel fuzzer, 2022. https://github.com/google/syzkaller.

[26] Google. Syzkaller's initial seeds, 2023. https://github.com/google/syzkaller/tree/master/sys/linux.

[27] Google. Syzbot, 2024. https://syzkaller.appspot.com/upstream.

[28] H. Han and S. K. Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Nov. 2017.

[29] H. Han, D. Oh, and S. K. Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[30] J. Horn. fs: use acquire ordering in __fget_light(), 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7ee47dcfff1835ff75a794d1075b6b5f5462cfed.

[31] D. Howells. watch_queue: Fix lack of barrier/sync/lock between post and read, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=2ed147f015af2b48f41c6f0b6746aa9ea85c19f3.

[32] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, June 2013.

[33] S. Huang, B. Cai, and J. Huang. Towards Production-Run heisenbugs reproduction on commercial hardware. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[34] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer Manuals, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[35] M. M. Islam and A. Muzahid. Bugaroo: Exposing memory model bugs in many-core systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 178–188. IEEE, 2018.

[36] K. Iwashima. ipv6: Fix data races around sk->sk_prot., 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=364f997b5cfe1db0d63a390fe7c801fa2b3115f6.

[37] R. Jain, R. Purandare, and S. Sharma. Bird: Race detection in software binaries under relaxed memory models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–29, 2022.

[38] J. Jang, M. Kang, and D. Song. Reusb: Replay-guided usb driver fuzzing. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[39] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[40] D. R. Jeong, M. Jung, Y. Lee, B. Lee, I. Shin, and Y. Kwon. Diagnosing kernel concurrency failures with aitia. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, Rome, Italy, May 2023.

[41] D. R. Jeong, B. Lee, I. Shin, and Y. Kwon. SegFuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2023.

[42] Z. Jiang, M. Wen, Y. Yang, C. Peng, P. Yang, and H. Jin. Effective concurrency testing for go via directional primitive-constrained interleaving exploration. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Kirchberg, Luxembourg, Sept. 2023.

[43] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Apr. 2022.

[44] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the 2010 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Reno, NV, Oct. 2010.

[45] J. Jung, J. Lee, J. Choi, J. Kim, S. Park, and J. Kang. Modular verification of safe memory reclamation in concurrent separation logic. In *Proceedings of the 2023 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Cascais, Portugal, Oct. 2023.

[46] J. Jung, J. Lee, J. Kim, and J. Kang. Applying hazard pointers to more concurrent data structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, 2023.

[47] J. Kang and J. Jung. A marriage of pointer-and epoch-based reclamation. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Virtual, June 2020.

[48] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[49] B. Kasikci, C. Zamfir, and G. Candea. Automated classification of data races under both strong and weak memory models. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37 (3):1–44, 2015.

[50] J. Kicinski. tls: improve lockless access safety of tls_err_abort(), 2023. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8a0d57df8938e9fd2e99d47a85b7f37d86f91097.

[51] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. HFL: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[52] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. Butler, A. Bianchi, and D. J. Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[53] K. Kim, S. Kim, K. R. Butler, A. Bianchi, R. Kennell, and D. J. Tian. Fuzz The Power: Dual-role state guided black-box fuzzing for USB power delivery. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[54] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.

[55] T. E. Kim, J. Choi, K. Heo, and S. K. Cha. DAFL: Directed grey-box fuzzing guided by data dependency. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[56] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[57] M. Kokologiannakis and V. Vafeiadis. Genmc: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.

[58] S.-H. Lee, M. Cho, R. Margalit, C.-K. Hur, and O. Lahav. Putting weak memory in order via a promising intermediate representation. In *Proceedings of the 2023 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Orlando, FL, June 2023.

[59] Y. Lee, C. Min, and B. Lee. ExpRace: Exploiting kernel races through raising interrupts. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[60] M. Lei. sbitmap: order READ/WRITE freed instance and setting clear bit, 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e6d1fa584e0dd9bfebaf345e9feea588cf75ead2.

[61] B. Lepers, J. Giet, W. Zwaenepoel, and J. Lawall. Ofence: Pairing barriers to find concurrency bugs in the linux kernel. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, Rome, Italy, May 2023.

[62] B. Li. mm/filemap: avoid buffered read/write race to read inconsistent data, 2023. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e2c27b803bb664748e090d99042ac128b3f88d92.

[63] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.

[64] C. Lidbury and A. F. Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Phoenix, AZ, June 2019.

[65] Linux. The kernel concurrency sanitizer (KCSAN), 2020. https://docs.kernel.org/dev-tools/kcsan.html.

[66] Linux. The Kernel Address Sanitizer (KASAN), 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/dev-tools/kasan.rst.

[67] Linux. kcov: code coverage for fuzzing, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/dev-tools/kcov.rst.

[68] Linux. Runtime locking correctness validator, 2022. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[69] Linux. The Undefined Behavior Sanitizer - UBSAN, 2022. https://www.kernel.org/doc/Documentation/dev-tools/ubsan.rst.

[70] LLVM Project. The LLVM Compiler Infrastructure, 2021. https://llvm.org/.

[71] W. Luo and B. Demsky. C11tester: a race detector for c/c++ atomics. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual, Apr. 2021.

[72] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[73] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[74] L. W. McVoy, C. Staelin, et al. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, Jan. 1996.

[75] J. Min, D. Yu, S. Jeong, D. Song, and Y. Jeon. Erasan: Efficient rust address sanitizer. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2024.

[76] S. Mukherjee, P. Deligiannis, A. Biswas, and A. Lal. Learning-based controlled concurrency testing. In *Proceedings of the 2020 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Virtual, Sept. 2020.

[77] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[78] L. Nan. nbd: fix null-ptr-dereference while accessing 'nbd->config', 2023. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c2da049f419417808466c529999170f5c3ef7d3d.

[79] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[80] B. Norris and B. Demsky. Cdschecker: checking concurrent data structures written with c/c++ atomics. In *Proceedings of the 24th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[81] R. O'callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, San Diego, CA, June 2003.

[82] N. Piggin. [PATCH] buffer: memorder fix, 2007. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=72ed3d035855841ad611ee48b20909e9619d4a79.

[83] QEMU. QEMU: A generic and open source machine emulator and virtualizer, 2021. https://www.qemu.org/.

[84] Rust for Linux. Rust for Linux, 2022. https://rust-for-linux.com/#rust-for-linux.

[85] S. Saleem. RDMA/irdma: Add missing read barriers, 2023. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4984eb51453ff7eddee9e5ce816145be39c0ec5c.

[86] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[87] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[88] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[89] J. Sitnicki. net/tls: Annotate access to sk_prot with READ_ONCE/WRITE_ONCE, 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d5bee7374b68de3c44586d46e9e61ffc97a1e886.

[90] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[91] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. Sok: Sanitizing for security. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[92] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, Aug. 2020.

[93] E. Stepanov and K. Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[94] A. Stern. Explanation of the Linux-Kernel Memory Consistency Model, 2017. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/memory-model/Documentation/explanation.txt.

[95] A. Stern. USB: core: Fix hang in usb_kill_urb by adding memory barriers, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=26fbe9772b8c459687930511444ce443011f86bf.

[96] B. A. Stoica, S. Lu, M. Musuvathi, and S. Nath. WAFFLE: Exposing memory ordering bugs efficiently with active delay injection. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, Rome, Italy, May 2023.

[97] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[98] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang. KSG: Augmenting kernel fuzzing with system call specification generation. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*, Carlsbad, CA, July 2022.

[99] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Orlando, FL, Feb. 2014.

[100] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[101] B. Töpel. xsk: use state member for socket synchronization, 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=42fddcc7c64b723a867c7b2f5f7505e244212f13.

[102] L. Torvalds. Merge tag 'rust-v6.1-rc1' of https://github.com/rust-for-linux/linux, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b.

[103] B. Töpel. xsk: add missing write- and data-dependency barrier, 2018. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=37b076933a8e38e72ffd3c40d3eeb5949f38baf3.

[104] J. D. Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*. Citeseer, 1994.

[105] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[106] A. Viro. missing barriers in some of unix_sock ->addr and ->path accesses, 2019. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ae3b564179bfd06f32d051b9e5d72ce4b2a07c37.

[107] B. Wang. Arm-based PCs to Nearly Double Market Share by 2027, 2023. https://www.counterpointresearch.com/insights/arm-based-pcs-to-nearly-double-market-share-by-2027/.

[108] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[109] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the 7th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Nuevo Leone, Mexico, Jan. 2001.

[110] P. H. Wang, H. Wang, J. D. Collins, E. Grochowski, R.-M. Kling, and J. P. Shen. Memory latency-tolerance approaches for itanium processors: out-of-order execution vs. speculative precomputation. In *Proceedings of the 8th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Boston, MA, Feb. 2002.

[111] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, Pittsburgh, PA, May 2022.

[112] D. Wolff, S. Zheng, G. Duck, U. Mathur, and A. Roychoudhury. Grey-box fuzzing for concurrency testing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Diego, CA, Apr.–May 2024.

[113] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[114] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[115] Z. Yejian. ring-buffer: Fix race while reader and writer are on the same page, 2023. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6455b6163d8c680366663cdb8c679514d55fc30c.

[116] M. Yuan, B. Zhao, P. Li, J. Liang, X. Han, X. Luo, and C. Zhang. DDRace: finding concurrency uaf vulnerabilities in linux drivers with directed fuzzing. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[117] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[118] T. Zhang, D. Lee, and C. Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[119] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[120] D. Zhu. net: fix a data race when get vlan device, 2021. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c1102e9d49eb36c0be18cb3e16f6e46ffb717964.

| Dependency | Description | Applied to |
|---|---|---|
| **Data** | A load and a store operation are linked by a data dependency if the value obtained by the value stored by the store operation. | load-store reordering |
| **Address** | A load operation and another memory access are linked by an address dependency if the value obtained by the load operation affects the location accessed by the other event. The second event can be either a load operation or a store operation. | load-load & load-store reordering |
| **Control** | A load operation `X` and a store operation `Y` are linked by a control dependency if `Y` syntactically lies within an arm of an if statement and `X` affects the evaluation of the if condition via a data or address dependency. | load-store reordering |

**Table 6.** Three types of dependencies defined in the LKMM.

# 10 Appendix

## 10.1 Details of How OEMU Complies with the LKMM

As explained in §3, OEMU reorders memory accesses through a delayed store operation and a versioned load operation. These two operations should not reorder memory accesses in a way that none of architectures that the Linux kernel supports would do.

However, different architectures define different rules of out-of-order execution. Thus, instead of implementing OEMU for all architectures, we implement OEMU to comply with the Linux Kernel Memory Model [2, 94]. The LKMM is a unified memory model to cope with different architectures, and it defines reordering cases that would not occur in any architectures that the Linux kernel supports. In other words, if concurrent execution of the Linux kernel correctly behaves according to the LKMM, it will not cause an OOO bug in any architecture that the Linux kernel supports. This work aims to comply with the LKMM, making the kernel behave correctly even on an exotic architecture.

In the LKMM, reordering cases that should not happen are reflected in the ppo relationship, which defines a relationship between two instructions that a CPU is obliged to execute them in the sequential order. The LKMM defines five cases that a memory barrier prevents reordering of two memory accesses `x` and `y`, where each case corresponds to a barrier presented in Table 1 (§10.1.1). On the other hand, the LKMM defines two cases that dependencies from a former load operation to subsequent memory accesses. Among the two cases, one describes dependencies between a former load operation and a later load operation (*i.e.*, **Case 6**), and the other case describes dependencies between a former load operation and a later store operation (*i.e.*, **Case 7**) (§10.1.2).

**10.1.1 Reordering Cases Prohibited by Memory Barriers. Case 1** describes that memory accesses of two instructions `X` and `Y` should not be reordered if a strong barrier (*e.g.*, `smp_mb()`) exists, regardless of the types of memory accesses. OEMU obeys this case by performing the memory access of `X` prior to the memory barrier, and the memory access of `Y` after the memory barrier. In particular, if `X` performs a load operation, it will never be performed after the memory barrier, as OEMU never delays a load operation. If `X` performs a store operation, it will also never be performed after the memory barrier, as OEMU commits the value of the store operation at least right before the memory barrier. Similarly, if `Y` performs a store operation, it will not be performed before the memory barrier, as OEMU only delays a store operation. In the case that `Y` performs a load operation, the versioning window restricts the load operation on `Y` to read a value written after the execution of the memory barrier, ensuring that the load operation is not performed before the memory barrier.

**Case 2** describes a case that there is a store barrier (*e.g.*, `smp_wmb()`) between two store operations `X` and `Y`. In this case, OEMU should finish the first store operation `X` before executing the second store operation `Y`. During the execution, since a delayed store operation ensures that the value of `X` has been committed right before a store barrier is executed, `X` is always finished before executing `Y`.

**Case 3** describes a case that there is a load barrier (*e.g.*, `smp_rmb()`) between two load operations `X` and `Y`. In this case, suppose `smp_rmb()` is executed at time $t_0$. Then, a load operation on `X` finishes reading a value before $t_0$ (regardless of whether it reads an old version from the store history or the latest version from memory). Also, OEMU guarantees that the value read by the load operation on `Y` is written after $t_0$ (refer to the versioning window in §3.2). Consequently, OEMU ensures that a load operation on `X` is executed before the memory barrier, while a load operation on `Y` reads a value written after $t_0$, ensuring that it is not executed before the memory barrier.

**Case 4** is a case that the former memory access is annotated with an acquire memory barrier (*e.g.*, `smp_load_acquire(&X)`). In this case, the load operation on `X` should be finished before all subsequent memory accesses. To this end, OEMU behaves as if there is a load barrier right after the load operation on `&X`. In other words, if any subsequent memory access `Y` is a load operation, OEMU prohibits reordering of load operations `X` and `Y` as done by **Case 3**. If any subsequent memory access `Y` is a store operation, OEMU ensures that the load operation on `&Y` is executed before `&Y`, as it does not emulate load-store reordering.

**Case 5** is a case that the latter memory access is annotated with a release memory barrier (*e.g.*, `smp_store_release(&Y)`). In this case, all precedent memory accesses should be finished before the store operation on `Y`. To faithfully emulate this, OEMU acts as there is a store barrier (*i.e.*, `smp_wmb()`) right before the store operation on `&Y`. Accordingly, if any precedent memory access `X` is a store operation, it ensures that the value of the store operation on `X` is committed before `Y` because OEMU acts as if there is a store barrier between them. Also, if any precedent memory access `X` is a load operation, OEMU ensures that the load operation on `X` is executed before `Y`, as it does not emulate load-store reordering.

**Algorithm 2:** Algorithm of the $filter\_out()$ function

**Input** : $S_i, S_j$: Sequences of memory accesses and memory barriers executed by two system calls.
**Output** : $S'_i, S'_j$: Sequences of memory accesses and memory barriers in which irrelevant memory accesses are filtered.

```
1  shared_mem = ∅
2  for (a_i, a_j) ∈ S_i × S_j do
3      if either a_i or a_j is not a memory access then
4          continue
5      o = shared_memory_location(a_i, a_j)
6      if o ≠ ∅ then
7          shared_mem = shared_mem ∪ {o}
8  for k ∈ {i, j} do
9      for a ∈ S_k do
10         if a is not a memory access then
11             continue
12         if a.addr ∉ shared_mem then
13             S_k = S_k \ {a}
14 S'_i, S'_j = S_i, S_j
15 return S'_i, S'_j
```

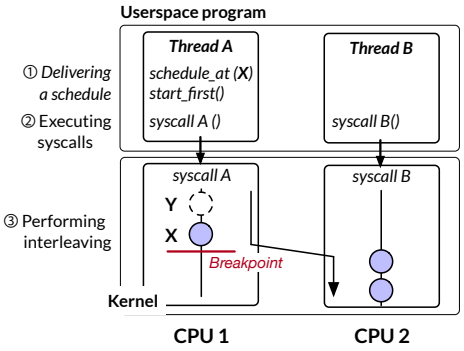



**Figure 9.** Workflow of the custom scheduler

#### 10.1.2 Reordering Cases Prohibited by Dependencies.

In addition to cases regarding memory barriers, the LKMM defines reordering cases that are prohibited by dependencies. Specifically, the LKMM defines three types of dependencies, such as address, data, and control dependencies as shown in Table 6. As shown in the table, all three types of dependencies restrict reordering of a load operation and a store operation (*i.e.*, **Case 7**), while only one dependency (*i.e.*, address dependency) can restrict reordering between two load operations (*i.e.*, **Case 6**).

**Case 6** describes a case that there is an address dependency between two load operations `X` and `Y` (*e.g.*, `X` and `Y` can be reading `i` and reading `arr[i]`), and `X` is annotated with `READ_ONCE()` or an atomic operation (*e.g.*, `atomic_read()`). Interestingly, due to the Alpha architecture, the LKMM prohibits reordering of `X` and `Y` only if `X` is annotated with `READ_ONCE()` or an atomic operation. In other words, without using such APIs, *the LKMM allows load-load reordering even if there is an address dependency* (please refer to "18. THE PRESERVED PROGRAM ORDER RELATION: ppo" and "19. AND THEN THERE WAS ALPHA" in [94]). Thus, OZZ is required to respect the address dependency only if the load operation on `X` is annotated with `READ_ONCE` or an atomic operation. This is the reason why OEMU considers `READ_ONCE()` as a load memory barrier (as explained in §3.2). In other words, if `X` is annotated with `READ_ONCE()`, OEMU thinks that there is a load memory barrier right after `X`. As a consequence, `Y` cannot read an old value written before `X`, preventing reordering of `X` and `Y`.

**Case 7** describes a case with a dependency between a precedent load operation and a subsequent store operation. According to the LKMM, all three dependencies can be applied in this case. Thus, if there is a dependency between a load and a store operation, OEMU should not reorder them. However, as we mentioned in §3, OEMU just does not reorder a former load operation and a latter store operation regardless of dependencies, as it is out-of-scope in this paper.

### 10.2 Algorithm to Filter Out Memory Accesses Irrelevant to OOO bugs

The OZZ's first step to calculate scheduling hints (§4.3) is to filter out memory accesses that cannot contribute to the manifestation of OOO bugs. As an OOO bug is a kind of concurrency bug, OZZ excludes memory accesses that do not access shared memory locations.

Algorithm 2 represents an algorithm of how to filter out such memory accesses. It takes, as an input, sets of memory accesses and memory barriers that are executed by two system calls $S_i$ and $S_j$. Then, OZZ finds out memory locations shared between the two system calls (#2 ~ #7). For each pair of memory accesses executed by the two system calls (#2), OZZ calculates an overlapped memory location on which at least one of the memory accesses writes a value. OZZ collects all such locations into a set *shared_mem*. If a memory access does not access a memory location contained in *shared_mem*, the memory access cannot contribute to the manifestation of an OOO bug.

After that, for each of memory accesses of each system call (#8 ~ #9), OZZ excludes memory accesses that do not access a memory location contained in *shared_mem* (#12 ~ #13). After filtering out memory accesses, OZZ returns filtered memory accesses as well as memory barriers (#14 ~ #15) to the next step (`Step 2 and 3` in §4.3).

### 10.3 Custom Scheduler

In addition to OEMU, OZZ requires a mechanism to deterministically control thread interleaving. To this end, we adopt a mechanism from previous works [20, 22, 23, 39], which we call a custom scheduler. The custom scheduler allows the fuzzer to specify a scheduling point on which an interleaving is performed. The custom scheduler is implemented in the hypervisor layer to override the scheduler of the guest kernel, and exposes hypercall interfaces to accept a scheduling

```
/* Initially, x = 0, y = 0 */
// In thread 1
thread_1.x.store(1, Ordering::Relaxed);
thread_1.y.load(Ordering::Relaxed)
// In thread 2
thread_2.y.store(1, Ordering::Relaxed);
thread_2.x.load(Ordering::Relaxed)
// In another thread running after thread 1 and thread 2
assert!(x == 1 || y == 1);
```

**Figure 10.** A *synthetic* OOO bug example written in Rust. Even in this example, OEMU can be used to identify the assertion violation (*i.e.*, the values of `x` and `y` are both `0`).

point as an input from a guest userspace program. During the kernel execution, the custom scheduler suspends and resumes the execution of virtual CPUs in other to control thread interleaving.

Figure 9 shows a workflow of the custom scheduler. In this example, suppose there are two threads to run concurrently, and they are pinned on different CPUs. If Thread A wants to instruct the custom scheduler to perform at an interleaving at `Y`, Thread A invokes a hypercall `schedule_at(Y)` to deliver a scheduling point to the custom scheduler (①). The custom scheduler, then, installs a breakpoint on the instruction. After delivering a scheduling point, each thread starts executing system calls (②). When running system calls, the custom scheduler keeps only one thread to run while the other thread is suspended. This is done by suspending a virtual CPU on which the suspending thread runs. In this example, Thread B is suspended at first (see `start_first()` in Thread A). During the execution, when the running thread (*i.e.*, Thread A) hits the scheduling point (*i.e.*, the breakpoint), the custom scheduler performs an interleaving by suspending the virtual CPU on which the thread is running, and resuminmg the suspended virtual CPU (③). It is worth noting that when suspending a virtual CPU, OEMU may reorder memory accesses. In this figure, OEMU performs a delayed store operation on `X`, so the value of the store operation has not been committed to memory when suspending a virtual CPU (see the circle on `X` is empty). Lastly, when the running thread finishes its execution, the custom scheduler resumes the suspended virtual CPU to complete the execution of two system calls.

### 10.4 OoO Bug Example in Rust

Since out-of-order execution is a behavior of a processor, OOO bugs occur in any language, including Rust, that implements low-level system software. Recently, the Linux kernel adopted Rust as the second official programming language, and we check that OEMU and OZZ are effective in Rust.

Figure 10 shows an example of a *synthetic* OOO bug written in Rust. Rust requires all concurrent memory accesses to be annotated with an ordering semantic. In this example, accesses to `x` and `y` are annotated with `Ordering::Relaxed`, which does not enforce any ordering between accesses. Thus, when `thread 1` and `thread 2` run concurrently, out-of-order execution may run `#4` before `#3`, and `#7` before `#6`. In that case, `assert!()` will fail, meaning that both `x` and `y` are `0`.

In this example, we confirm that OEMU can trigger the OOO bug by reordering memory accesses between either `#3` and `#4`, or `#6` and `#7`. Consequently, the assertion violation `#9` can be triggered. As we said in the discussion section, we expect that Rust will be used more in the near future, and we also expect that OZZ and OEMU will help developers for kernel modules written in Rust.