# SEGFUZZ: Segmentizing Thread Interleaving to Discover Kernel Concurrency Bugs through Fuzzing

Dae R. Jeong† Byoungyoung Lee‡ Insik Shin† Youngjin Kwon†
† *School of Computing, KAIST*,
‡ *Department of Electrical and Computer Engineering, Seoul National University*

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179398

***Abstract*—Discovering kernel concurrency bugs through fuzzing is challenging. Identifying kernel concurrency bugs, as opposed to non-concurrency bugs, necessitates an analysis of possible interleavings between two or more threads. However, because the search space of thread interleaving is vast, it is impractical to investigate all conceivable thread interleavings. To explore the vast search space, most previous approaches perform random or simple heuristic searches without having coverage for thread interleaving or with an insufficient form of coverage. As a result, they either conduct wasteful searches with redundant executions or overlook concurrent bugs that their coverage cannot address.**

**To overcome such limitations, we propose SEGFUZZ, a fuzzing framework for kernel concurrency bugs. When exploring the search space of thread interleavings, SEGFUZZ decomposes an entire thread interleaving into a set of segments, each of which represents an interleaving of the small number of instructions, and utilizes individual segments as interleaving coverage, called interleaving segment coverage. When searching for thread interleavings, SEGFUZZ mutates interleavings in explored interleaving segments to construct new thread interleavings that have not yet been explored. With SEGFUZZ, we discover new 21 concurrency bugs in Linux kernels, and demonstrate the efficiency of SEGFUZZ by showing that SEGFUZZ can identify known bugs on average 4.1 times quickly than the state-of-the-art approaches.**

## I. INTRODUCTION

Concurrency bugs are common in modern kernels due to the pervasive adoption of efficient but difficult-to-reason parallelization techniques. The consequence of kernel concurrency bugs is disastrous. They crash the entire system, breaking availability [16, 57] or causing data loss [48]. Even worse, attackers exploit concurrency bugs to mount a privilege escalation attack. These threats are becoming more serious in recent years, as the number of concurrency bugs has been steadily increasing in Linux kernels [26], and a recent study [38] demonstrates that a user-level attacker reliably exploits concurrency bugs. Furthermore, we observe that many zero-day attacks in Linux kernels [21, 23, 31, 32, 39] use concurrency bugs by exploiting malicious thread interleavings.

However, discovering kernel concurrency bugs is much more difficult than finding non-concurrency bugs. In contrast to non-concurrency bugs which are discovered by sequential testing of single-thread execution, kernel concurrency bugs are typically caused by the concurrent execution of two or more threads. Kernel concurrency bugs only emerge in a particular pattern of interleavings between threads, which happens only when a certain timing condition is met. Therefore, to discover kernel concurrency bugs, a fuzzer must consider two aspects: how to explore execution paths (*i.e.*, sequential aspects), and how to navigate thread interleavings (*i.e.*, concurrent aspects).

For decades, coverage-guided fuzzing has been extensively studied to effectively explore the search space of execution paths [8, 11, 20, 22, 35, 36, 70, 75, 89]. However, little progress has been made in exploring thread interleavings. Specifically, little is known about how to define the coverage metric for *thread interleavings* (shortly, interleaving coverage metric), and how to utilize interleaving coverage metric in exploring thread interleavings. For example, Razzer [28] and Snowboard [17] use their own heuristics combined by static or dynamic analysis, but they are coverage-oblivious. They cannot prune redundant executions of thread interleavings that produce identical coverage. KRACE [88] proposes alias coverage, which tracks concurrently-executed *two* instructions, but alias coverage suffers from effectively summarizing the behavior of concurrency bugs. This is because most concurrency bugs are caused by interleavings of *more* instructions [49]. Furthermore, KRACE randomly schedules instructions during runtime without considering what thread interleavings are explored before.

To overcome the shortcoming of existing techniques, this paper proposes SEGFUZZ, a fuzzing framework for kernel concurrency bugs. At its core, SEGFUZZ aims to systematically explore the vast search space of thread interleavings to quickly identify interleavings leading to concurrency bugs. The central idea behind SEGFUZZ is i) defining a new, effective interleaving coverage metric that reflects the common nature of concurrency bugs, and ii) using that metric to drive an efficient search strategy for discovering previously untested thread interleavings. Hence, we demonstrate that SEGFUZZ significantly saves CPU costs in kernel fuzzing by finding concurrent bugs quickly. SEGFUZZ achieves the goal with the following two key ideas: segmentizing thread interleaving and the mutation-based thread interleaving exploration.

**Segmentizing thread interleaving.** The key idea is built upon the discovery in the previous study [49] which highlights that most concurrency bugs occur when at most four memory accesses referring to shared memory objects are interleaved. Based on this finding, SEGFUZZ decomposes the large thread interleaving space into tractable small sub-spaces, called *interleaving segment*. The number of instructions in an interleaving segment is small (at most four by default), and each interleaving segment represents an interleaving of those instructions.


DOI 10.1109/SP46215.2023.00161

Using interleaving segments, we propose novel interleaving coverage metric, *interleaving segment coverage*, which is defined as a set of detected interleaving segments. Using interleaving segment coverage provides two benefits to fuzzers. First, a fuzzer can consider interactions of multiple instructions, which increases accuracy of discovering concurrency bugs in comparison to previous approaches [30, 88]. Second, a fuzzer can test the unexplored thread interleavings guided by interleaving segment coverage while avoiding redundant testing.

**Mutation-based thread interleavings exploration.** To effectively test unexplored thread interleavings, SEGFUZZ proposes a mutation approach on observed interleaving segments that are recorded in interleaving segment coverage. This is achieved by changing the interleaving orders of instructions in each segment. This approach allows SEGFUZZ to proactively infer which instruction orders are possible before running them, enabling it to determine the next interleavings to be tested. Furthermore, SEGFUZZ mutates as many segments as possible to generate unexplored thread interleavings from the mutated segments, which helps to navigate the search space of thread interleavings more quickly

We implement SEGFUZZ across various software layers. We run SEGFUZZ against the latest version of the Linux kernel (ranging from 5.19-rc2 to 6.2), and find new 21 concurrency bugs all of which exhibits harmful behaviors such as memory corruption, task hangs, or assertion violations. We emphasize that all concurrency bugs found by SEGFUZZ were lurking in subsystems where Syzkaller [20] has been testing for several years, which demonstrates the usefulness of SEGFUZZ in discovering concurrency bugs. To quantify the efficiency of SEGFUZZ, we experimentally compare SEGFUZZ against the state-of-the-art techniques, Snowboard and KRACE, and demonstrate that SEGFUZZ discovers concurrency bugs more quickly (4.1× on average) than previous works.

We describe our contributions in three folds:

- **Interleaving segment coverage:** Based on the idea of segmentizing thread interleaving, we propose interleaving segment coverage, which encodes interleavings of multiple instructions, and provides solid clues for a fuzzer to track the progress of exploring thread interleavings.
- **Mutation-based interleaving exploration:** Unlike the random or simple heuristic exploration of thread interleavings (as in most previous approaches), we propose the mutation-based interleaving exploration, which strategically and quickly explores the search space of thread interleavings.
- **Practical impact:** We discovered new 21 harmful concurrency bugs in recent Linux kernels, which have been extensively tested before, demonstrating the practicality of SEGFUZZ.

## II. BACKGROUND

### A. (Conventional) Kernel Fuzzing

Throughout decades, a number of fuzzing approaches [20, 22, 33, 35, 36, 68, 75, 89] have been proposed to discover vulnerabilities in the kernel. These approaches, specifically coverage-guided fuzzing, execute a large number of random inputs (*i.e.*, sequences of system calls) while a code coverage metric (*e.g.*, branch coverage) tracks the execution path that each input explores. Although conventional fuzzing techniques have proven useful in large system software, they primarily focus on the sequential aspect of execution, which relates to the execution path of a single thread and do not adequately consider the concurrent aspect, which involves thread interleaving among threads. As a consequence, conventional fuzzing is limited in discovering concurrency bugs such as race conditions, data races, and deadlocks [12, 77].

### B. Concurrency Fuzzing

Recently, several approaches [10, 17, 28, 30, 88] have been proposed to consider both sequential and *concurrent* aspects of the kernel execution into fuzzing. Concurrency fuzzers repeatedly execute a multi-thread input (*e.g.*, a multi-threaded system call sequence) while controlling thread scheduling and tracking unique behaviors of thread interleavings.

**Thread interleaving exploration.** Unlike conventional fuzzers, concurrency fuzzers consider thread interleaving is another input domain that a fuzzer needs to explore. To explore diverse thread interleavings, concurrency fuzzers adopt various thread interleaving exploration schemes. For instance, at every iteration, Razzer [28] and Snowboard [17] selects a single pair of instructions, and determines and enforces their execution order. Whereas, KRACE [88] randomly schedules instructions, and Conzzer [30] designates two functions and run them concurrently. During runtime, concurrency fuzzing overrides the kernel's scheduler, and controls thread scheduling as desired.

**Interleaving coverage metric.** As KRACE [88] reveals, code coverage metrics (*e.g.*, branch coverage) are not enough to capture unique behaviors of thread interleavings, because code coverage metrics only track behaviors in a single thread without paying attention to thread interleavings. To capture unique behaviors of thread interleavings, a few concurrency fuzzers [30, 88] come up with interleaving coverage metrics, which track the execution order of concurrent events between threads, ranging from a finer-granularity (*e.g.*, instruction [88]) to a coarser-granularity (*e.g.*, function [30]). Concurrency fuzzers utilize interleaving coverage when determining if thread interleavings of a multi-thread input needs to be further tested.

## III. MOTIVATION

We first study how concurrency bugs manifest depending on thread interleaving through a real-world bug example. Then, we identify design objectives to discover kernel concurrency bugs efficiently and discuss limitations in prior work.

**Manifestation of concurrency bugs.** In Figure 1, an uninitialized access bug may manifest when two system calls are executed concurrently: `sendmsg()`, and `setsockopt()`. Let us assume `inet->hdrincl` is initially 1. During the execution of `sendmsg()`, thread A reads a value of `inet->hdrincl` twice at `A2` and `A4`, which may be intervened in the middle by another thread (*i.e.*, thread B). In that case, if `B1` is executed between

**Initial value:** `inet->hdrincl = 1;`

| | Thread A - sendmsg() | | Thread B - setsockopt() |
|---|---|---|---|
| | /* In raw_sendmsg() */ | | /* In do_ip_setsockopt() */ |
| A1 | struct raw_frag_vec **rfv**; | | |
| A2 | if (**inet->hdrincl** == 0) | | |
| A3 | initialize_rfv(**&rfv**); | B1 | **inet->hdrincl** = 0; |
| A4 | if (**inet->hdrincl** == 0) | | |
| A5 | ip_append_data(..., **&rfv**, ...); | | |
| A6 | sk->owned = 1; | B2 | sk->owned = 0; |

**Fig. 1:** Simplified code snippet of CVE-2017-17712. If `B1` is executed between `A2` and `A4`, concurrent accesses on `inet->hdrincl` leads to uninitialized stack pointer usage on `rfv`, and an attacker may gain root privileges through a dedicated attack technique [47].

`A2` and `A4`, thread A reads different values of `inet->hdrincl` at `A2` and `A4`, causing uninitialized access to `rfv` (*e.g.*, `A5`).

**Observation 1: Interleaving of multiple memory accesses.** This example demonstrates that *a specific interleaving of multiple instructions is necessary to cause a concurrency bug*. In the example of Figure 1, three memory accesses are interleaved, which eventually causes the uninitialized access bug. First, `A2` should be executed before `B1` (*i.e.*, `A2` $\Rightarrow$ `B1`[1]) to make thread A not initialize `rfv`. Second, `B1` should be executed before `A4` (*i.e.*, `B1` $\Rightarrow$ `A4`) to make thread A dereference uninitialized `rfv` while other memory accesses do not contribute to the bug. Therefore, to trigger the kernel concurrency bug, a fuzzer must consider interleavings of the three memory accesses, *e.g.*, (`A2` $\Rightarrow$ `B1` $\Rightarrow$ `A4`).

**Design goal 1: Informative interleaving coverage.** The observation gives an insight of how to define interleaving coverage metric. When discovering the uninitialized access bug, an interleaving coverage metric should distinguish different interleavings of multiple memory accesses. For example, (`B1` $\Rightarrow$ `A2` $\Rightarrow$ `A4`) (*i.e.*, Figure 2-(a)) and (`A2` $\Rightarrow$ `B1` $\Rightarrow$ `A4`) (*i.e.*, Figure 2-(b)) should be distinguished, and thus, interleaving coverage should not be saturated until (`A2` $\Rightarrow$ `B1` $\Rightarrow$ `A4`) is executed. Otherwise, a fuzzer may think that there is no more interesting thread interleaving in the multi-thread input after it executes Figure 2-(a), and stop searching for new interleavings in the input, missing the opportunity to find the concurrency bug. However, it is crucial that how many instructions the coverage metric consider to distinguish different interleavings. If interleaving coverage metric tracks interleavings of too few instructions (*e.g.*, 2), it may miss interleavings that should be tested. On the other hand, tracking interleavings of too many instructions (*e.g.*, thousands) cause high search complexity due to its large coverage space. Therefore, this work seeks a balancing point in the bug-finding capability and the search complexity when defining an interleaving coverage metric.

**Observation 2: Feedback from explored executions.** We find that even if an explored execution does not cause a concurrency bug, it provides useful feedback to guide what interleavings should be further explored. In the example of Figure 1, let us assume a fuzzer executes the two system calls *sequentially* such that thread B executes all instructions followed by the execution

[1]In this paper, `X` $\Rightarrow$ `Y` denotes that `X` is executed before `Y`

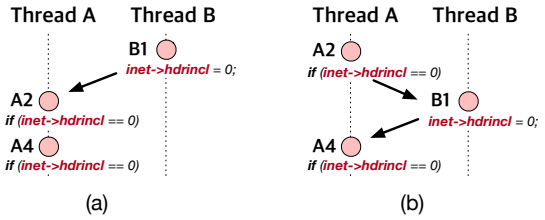


**Fig. 2:** Two thread interleavings between thread A and thread B described in Figure 1. The uninitialized access bug manifests only in (b). We omit bug-irrelevant memory accesses (*i.e.*, `A6` and `B2`).

of thread A (*i.e.*, Figure 2-(a)). In the explored execution, a fuzzer observes the three instructions, `B1`, `A2`, and `A4`, are executed in the order of (`B1` $\Rightarrow$ `A2` $\Rightarrow$ `A4`) and they access the same memory object (*i.e.*, `inet->hdrincl`). From this execution, one can easily imagine a new interleaving of these three instructions by changing the execution order of `B1` and `A2`, resulting in Figure 2-(b). The speculative interleaving is what exactly we are looking for; it is the offending interleaving causing the uninitialized access bug, *e.g.*, (`A2` $\Rightarrow$ `B1` $\Rightarrow$ `A4`), and, if executed, the interleaving triggers the bug.

**Design goal 2: Speculative interleaving exploration.** The observation gives a direction that how a fuzzer should explore the search space using a coverage metric. If interleaving coverage tracks that (`B1` $\Rightarrow$ `A2` $\Rightarrow$ `A4`) is *explored* before, a fuzzer can utilize interleaving coverage as feedback to infer *unexplored* thread interleavings (*e.g.*, `A2` $\Rightarrow$ `B1` $\Rightarrow$ `A4`). This allows a systematic way to explore the search space rather than performing randomized or heuristic-based exploration pervasively used in previous approaches [7, 10, 14, 88]. Considering concurrency bugs often manifest with a thread interleaving that rarely happen [38], the systematic interleaving exploration boosts up the concurrency bug discovery, as a fuzzer directly executes unexplored thread interleavings instead of executing random interleavings thousands of times.

### A. Limitation of prior approaches

Although prior approaches achieve their own successes, we find that their interleaving coverage metrics and interleaving exploration methods do not satisfy **Design goal 1** and **2**.

**Less informative interleaving coverage.** We find that previously proposed interleaving coverage metrics are limited in distinguishing two interleavings in Figure 2. Thus, they do not satisfy **Design goal 1**. For example, let us suppose we adopt alias coverage [88], which tracks interleaving orders of *two* instructions. Alias coverage determines an interleaved execution `X` exposes a new behavior if `X` contains an unexplored directed-instruction pair $I_W \rightarrow I_R$, where $I_R$ reads a value written by $I_W$. Assuming Figure 2-(a) is executed first, alias coverage identifies Figure 2-(a) exposes new behaviors when it sees two unexplored directed-instruction pair: (`B1` $\rightarrow$ `A2`) and (`B1` $\rightarrow$ `A4`). However, according to alias coverage, Figure 2-(b), which causes the uninitialized access bug, does not exhibit any new coverage because (`B1` $\rightarrow$ `A4`) is already explored in Figure 2-(a). Therefore, alias coverage may make the wrong decision

about whether a fuzzer needs to run these two system calls more, misleading a fuzzer to de-prioritize a multi-thread input in which a concurrency bug resides. In evaluation, we quantify the limitation in Table III. Likewise, concurrent call pair coverage [30] also suffers from distinguishing these two thread interleavings; These two interleavings take place in the same functions (*i.e.*, `raw_sendmsg()` and `do_ip_setsockopt()`), while concurrent call pair tracks a concurrently-executed function pair without being aware of fine-grained interleavings of instructions within functions.

**Ineffective interleaving exploration.** Stemming from the aforementioned limitations in coverage, existing thread interleaving exploration methods do not efficiently explore the search space of thread interleaving. Specifically, Razzer [28] and Snowboard [17] are coverage-oblivious. They do not make use of any interleaving coverage and perform their heuristic-based interleaving exploration. KRACE [88] randomly schedules instructions at every iteration without considering what thread interleavings are explored before. KRACE utilizes its own interleaving coverage only when deciding whether it runs a given multi-thread input more. Whereas Conzzer [30] designates two functions and run them concurrently, but it still leaves instructions' interleaving uncontrolled and scheduled randomly. In summary, existing approaches do not systematically search for thread interleavings, and execute redundant thread interleavings.

## IV. Exploring Thread Interleaving Space

This work focuses on how to efficiently explore the vast search space of thread interleavings when a multi-thread input (*e.g.*, `sendmsg()` and `setsockopt()` in Figure 1) is given. Our approach utilizes an *explored* thread interleaving as a seed interleaving (*e.g.*, Figure 3-(a)). With the seed interleaving, a fuzzer explores thread interleavings (*e.g.*, Figure 6-(c)) of the given input in the following three steps:

1) ***Decomposing*** the explored thread interleaving into segments containing a small number of instructions.
2) ***Mutating*** interleavings within segments to generate unexplored interleavings for each segment.
3) ***Recomposing*** mutated segments into whole interleaving to determine how to schedule instructions in future.

In this section, we first explain our key intuition behind decomposing explored interleavings into *interleaving segments* (§IV-A). Then, we propose a novel interleaving coverage metric based on decomposed interleaving segments (*i.e.*, *step 1*) to satisfy **Design goal 1** (§IV-B). Lastly, we explain how to mutate (*i.e.*, *step 2*) and recompose (*i.e.*, *step 3*) interleaving segments to explore thread interleavings to satisfy **Design goal 2** (§IV-C).

### A. Key Idea: Segmentizing thread interleaving

When fuzzing concurrency bugs with an interleaving coverage metric, the fundamental challenge is the large search space, because testing multiple memory accesses increases the search space exponentially. Therefore, this work seeks to strike the balance in the trade-off between the bug-finding capability and the search complexity.

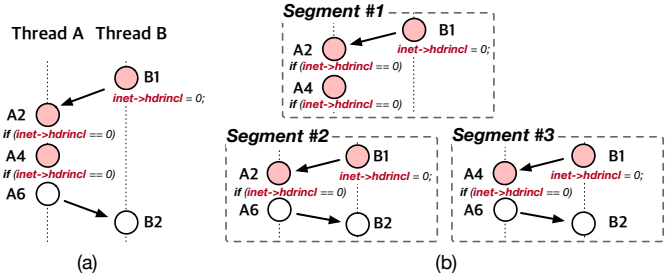


**Fig. 3:** (a) A thread interleaving example of Figure 1, and (b) interleaving segments contained in (a). Note that red circles correspond to instructions involved in triggering the uninitialized access bug.

**Segmentizing thread interleaving.** To reduce the search complexity, we take the classical wisdom of problem decomposition, where the complexity of a problem exponentially decreases as the problem size decreases. Our key strategy is *decomposing* the search space into small sub-spaces, called *interleaving segments*. Interleaving segments represent interleavings of a small number of instructions accessing shared memory objects. To define the size (*i.e.*, the number of instructions) of an interleaving segment, we use the observation from an extensive survey [49] in which 92.4% (97 out of 105) of concurrency bugs manifest due to the execution order of *at most* four memory accesses referring to shared memory objects. We confirm that the observation is still valid in recent Linux kernels. We collect 15 recent patches to fix kernel concurrency bugs, and check how many memory accesses are involved when triggering concurrency bugs fixed by these patches. Unsurprisingly, 14 concurrency bugs among them were caused by at most four memory accesses, while only 6 were caused by at most two memory accesses.

This finding implies that all other memory accesses beyond four accesses and their execution orders do not meaningfully affect manifestation of a concurrency bug. It is also applied to the example of Figure 1. The uninitialized access bug is triggered by the execution order of three memory accesses (*e.g.*, `A2`, `A4` and `B1`), while others (*e.g.*, `A6` and `B2`) are irrelevant to the manifestation of the concurrency bug. Following the observation, we confine the size of an interleaving segment to contain at most four memory-accessing instructions, which makes the problem of defining a coverage metric tractable while maintaining a strong bug-finding capability.

**Example of interleaving segments.** Figure 3 illustrates interleaving segments from the single execution of two system calls, where the execution scenario is represented in Figure 3-(a). Then, interleaving segments described in Figure 3-(b) represent a part of its thread interleaving, comprised by at most four instructions. For example, `Segment #1` represents an interleaving of three instructions `B1`, `A2`, and `A4`, stating that `B1` is executed before `A2` and `A4`. Similarly, `Segment #2` and `Segment #3` describe interleavings of (`B1`, `A2`, `A6`, and `B2`), and (`B1`, `A4`, `A6`, and `B2`) respectively. How to build segments is described in §IV-B.

**Benefits of segmentizing thread interleaving.** Segmentizing thread interleavings provides two benefits to a fuzzer. First, when defining an interleaving coverage metric, tracking explored interleavings in each segment gives a befitting guidance to determine whether a fuzzer further explores thread interleavings or not. Because most concurrency bugs manifest depending on interleavings of at most four memory accesses, if a fuzzer explores all possible interleavings for each detected segment, it is unlikely that a fuzzer misses a concurrency bug. Accordingly, *interleavings in each segment can act as an interleaving coverage metric, satisfying* ***Design goal 1***.

Second, explored interleaving segments can be used to systematically search for interleavings in next iterations. Since each interleaving segment contains only a small number of memory accesses, a fuzzer can speculatively creates new interleavings from explored interleavings. Taking an example of `Segment #1` in Figure 3, besides execution order (`B1` ⇒ `A2` ⇒ `A4`) (represented in `Segment #1`), a fuzzer easily rearranges the three instructions to speculate unexplored execution orders of these instructions such as (`A2` ⇒ `B1` ⇒ `A4`) which causes the uninitialized access bug if executed. Therefore, instead of doing a randomized search (as in most prior approaches), if a fuzzer figures out which of the enumerated interleavings have not been explored, *a fuzzer can strategically explore the search space with speculative interleavings, satisfying* ***Design goal 2***.

### *B. Interleaving Segment Coverage*

As discussed, we **decompose** explored thread interleavings into interleaving segments. We first represent an explored thread interleaving as a directed acyclic graph (DAG). Then, we compute subgraphs of the DAG called *segment graphs*, each of which represents an interleaving segment. Using a set of segment graphs, we propose a novel interleaving coverage metric called *interleaving segment coverage*. Interleaving segment coverage is a collection of explored segment graphs, designed to track interleavings within interleaving segments.

**Graph representation of thread interleaving.** In the representation of DAG, a vertex represents a memory-accessing instruction, and an edge represents an execution ordering between two memory-accessing instructions. There are two types of edges, program-order edges and interleaving-order edges. A program-order edge indicates an execution order between two memory-accessing instruction in a single thread. Whereas, an interleaving-order edge indicates the execution order between two memory-accessing instructions that 1) access the same shared data, 2) at least one of them is a write operation, and 3) are executed by different threads.

**Generating segment graphs.** Figure 4 illustrates how a fuzzer computes segment graphs from the execution example of Figure 3-(a). First, the given thread interleaving is represented as a DAG. Given that the execution in Figure 3-(a) involves five memory accesses, a fuzzer generates five vertices, each of which corresponds to a memory-accessing instruction as shown in Figure 4-(a). Between these vertices, a fuzzer generates edges to represent execution orderings. Specifically, program-order edges represented as dotted edges (*e.g.*, (`A2` ⇒ `A4`)) are drawn to represent orderings in a single thread, such as "`A2` is executed before `A4` in thread A". Similarly, a fuzzer generates interleaving-order edges represented as solid edges, expressing interleaving orders between threads. For example, (`B1` ⇒ `A2`) represents interleaving between `B1` and `A2` that access the same memory object (*i.e.*, `inet->hdrincl`). As a result, a fuzzer constructs a DAG to describe the whole thread interleaving. We provide an algorithm of the above process in Appendix A1.

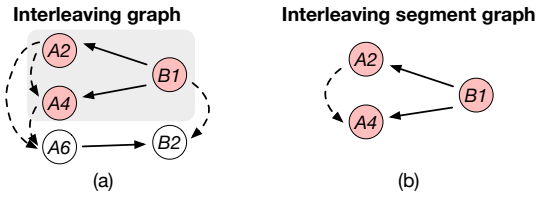


**Fig. 4:** (a) Graph representation of Figure 3-(a), and (b) segment graph corresponding to `Segment #1` in Figure 3-(b) (extracted from the gray part of (a)). In each graph, dotted arrows represent program-order edges, and solid arrows represent interleaving-order edges.

Afterwards, a fuzzer derives subgraphs, called *segment graphs*, from the generated DAG, each of which includes at most four vertices, which reflects the finding in the previous survey [49], To compute a segment graph $g$, a fuzzer selects two interleaving-order edges and vertices connected by these two edges. In the example, assume a fuzzer selects (`B1` ⇒ `A2`) and (`B1` ⇒ `A4`) from Figure 4-(a). As three vertices `A2`, `A4`, and `B1` are connected by these two edges, a segment graph $g$ contains the three vertices. Then, a fuzzer extracts all edges that connect these vertices (*i.e.*, the two interleaving-order edges and remaining (`A2` ⇒ `A4`)), and adds the edges into $g$, resulting in Figure 4-(b). A detailed algorithm is given in Appendix A2.

Lastly, a fuzzer gathers segment graphs into a set $G = \{g_1, g_2, ..., g_n\}$ (*e.g.*, in Figure 3, three segment graphs corresponding to `Segment #1`, `#2`, and `#3`), and uses $G$ to track interleaving segment coverage and to search unexplored thread interleavings.

**Tracking interleaving segment coverage.** Our interleaving coverage called *interleaving segment coverage* tracks segment graphs. When a new segment graph is detected, it is added to interleaving segment coverage. If new segment graphs are continuously discovered in a multi-thread input, it indicates that a fuzzer should invest more computing power to further explore thread interleavings in the input. Otherwise, a fuzzer can conclude that the input no longer reveals interesting interleavings. In practice, however, interleaving segment coverage contains a large number of segment graphs, which consumes a large amount of memory even though the size of individual segment graphs is small. To reduce memory consumption, we hash each segment graph, so interleaving segment coverage is stored as a universal hash table of segment graphs. We adopt Merkel hashing [25, 51]. The key property of Merkel hashing is reflecting directions of edges, so that a fuzzer can distinguish different interleavings of the same vertices. For example, four segment graphs described in Figure 5 are hashed into different values. Due to space constraints, we

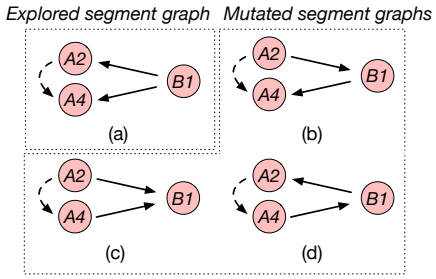


**Fig. 5:** Example of interleaving segment mutation. In this example, (a) represents an interleaving segment graph of Figure 4 while (b) and (c) represent *valid* mutated segments of (a). Whereas (d) is an *invalid* mutated segment of (a) because it contains a loop.

present how our graph hashing works in distinguishing different interleavings of Figure 5 in Appendix B.

### C. Mutation-based Interleaving Exploration

To explore the interleaving search space, we use detected segment graphs. Specifically, a fuzzer **mutates** *explored* interleaving segments into *unexplored* interleaving segments, where mutated segments contain new interleavings to be tested. Afterwards, a fuzzer **recomposes** mutated segments to obtain a speculative interleaving, and generates scheduling points from the recomposed interleaving.

**Mutating interleaving segment.** Suppose a set of *explored* segment graphs $G = \{g_1, g_2, ..., g_n\}$ (*e.g.*, segment graphs corresponding to Figure 3-(b)) is given. For each $g_i \in G$, a fuzzer mutates $g_i$ by flipping the directions of its interleaving-order edges, which implies changing the interleaving order of an instruction pair that access the same memory object. Figure 5 illustrates how a fuzzer mutates a segment graph. Given an explored segment graph described in Figure 5-(a), flipping (`B1` $\Rightarrow$ `A2`) produces a segment graph described in Figure 5-(b). Likewise, flipping both (`B1` $\Rightarrow$ `A2`) and (`B1` $\Rightarrow$ `A4`) in Figure 5-(a) generates a different segment graph of Figure 5-(c), which also represents another interleaving. Note that flipping only (`B1` $\Rightarrow$ `A4`) of Figure 5-(a), however, generates an *invalid* segment graph as shown in Figure 5-(d) because it creates a loop `B1` $\Rightarrow$ `A2` $\Rightarrow$ `A4` $\Rightarrow$ `B1`. When a mutated segment graph contains a loop, a fuzzer discards the segment. Note that a fuzzer drops a mutated segment graph if it is already explored, eliminating redundant executions. A fuzzer identifies such case when the hash value of a newly mutated segment graph is recorded in interleaving segment coverage. In such a way, a fuzzer forms a set of *unexplored* mutated segment graphs $G_{mutated}$ (*e.g.*, in Figure 3, mutated segment graphs derived from `Segment #1`, `#2`, and `#3`), and recomposes them for the next search.

**Recomposing mutated segment graphs.** One could take up a single mutated segment graph in $G_{mutated}$ for the next search. However, it might require many search iterations because the size of $G_{mutated}$ could be large. Instead, we select multiple mutated segment graphs, and *recompose* them to form a large graph to test multiple mutated segments at one execution.

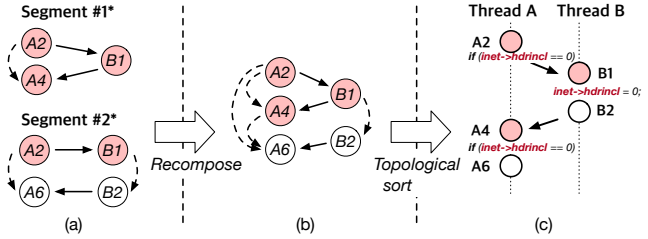


**Fig. 6:** Example of recomposing mutated segment graphs into a speculative thread interleaving, where (a) represents mutated segment graphs, (b) represents a graph combining the two mutated segment graphs, and (c) represents an instruction sequence to test `Segment #1*` and `#2*`

Figure 6 demonstrates a sketch of recomposing steps. Suppose, from $G_{mutated}$, we select two mutated segment graphs called `Segment #1*` and `#2*` (*i.e.*, Figure 6-(a)), which are derived from `Segment #1` and `#2` in Figure 3-(b) respectively. We can then merge the two segment graphs, resulting in a larger graph containing all vertices and all edges from `Segment #1*` and `#2*` as described in Figure 6-(b). This large graph describes all interleavings from selected mutated segment graphs, and later, is used to schedule instructions to test both `Segment #1*` and `#2*` at once (*i.e.*, Figure 6-(c)).

When recomposing mutated segment graphs, one important constraint is that a loop should not be formed (*i.e.*, Figure 6-(b) should not contain a loop). To select multiple mutated segment graphs without making a loop, a fuzzer starts with an empty set of $G^*_{mutated}$. Then, a fuzzer iterates over all mutated segment graphs $g \in G_{mutated}$. From $g$, a fuzzer tries to add edges in $g$ into $G^*_{mutated}$ one by one and check it causes a loop or not. If it does not form a loop, a fuzzer adds the segment graph $g$ into $G^*_{mutated}$. After iterating all mutated segment graphs in $G_{mutated}$, a fuzzer obtains $G^*_{mutated}$ which is a subset of $G_{mutated}$ without containing a loop. Then, a fuzzer excludes selected mutated segment graphs from the set of all mutated segment graphs (*i.e.*, $G_{mutated} = G_{mutated} \setminus G^*_{mutated}$), and generates scheduling points to test mutated interleaving segments contained in $G^*_{mutated}$. We show this process in Algorithm 3 in Appendix A3.

**Generating scheduling points.** As a final step, once $G^*_{mutated}$ (*i.e.*, Figure 6-(b)) is obtained, a fuzzer creates scheduling points, where each scheduling point describes an instruction on which preemption should happen (including the end of each system call) as well as the next thread to run. A fuzzer generates scheduling points by conducting a topological sort [34] to $G^*_{mutated}$, which generates an instruction sequence. Figure 6-(c) shows the result of conducting the topological sort to Figure 6-(b). As shown in the figure, the resulting instruction sequence can be used to test merged mutated segments, *i.e.*, `Segment #1*` and `#2*`. In this instruction sequence, a fuzzer selects scheduling points as `A2`, `B2` and `A6`, and uses scheduling points when enforcing thread scheduling as detailed in §V-C.

## V. Design of SegFuzz

SegFuzz is a kernel concurrency fuzzer adopting interleaving segment coverage (§IV-B) and the mutation-based

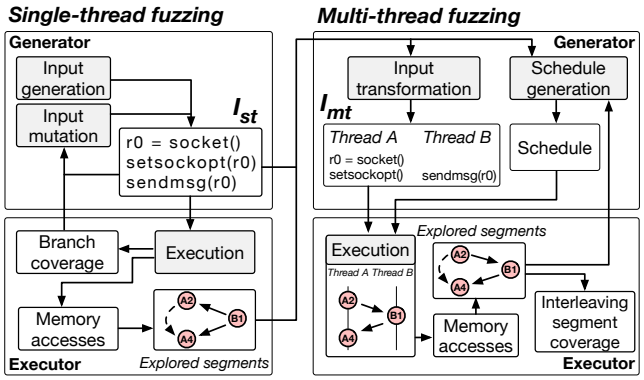


**Fig. 7:** An overview of SEGFUZZ's architecture.

interleaving exploration (§IV-C). As shown in Figure 7, SEGFUZZ consists of two stages of fuzzing. The first stage is a *single-thread fuzzing*, which searches for execution paths using branch coverage, and the second stage is a *multi-thread fuzzing* to explore thread interleavings using interleaving segment coverage. In both stages, SEGFUZZ traces timestamp-annotated memory accesses executed by each system call, and the second stage adopts a mechanism to control thread scheduling.

In the following, we provide the overall design of SEGFUZZ (§V-A). Then, we describe the kernel instrumentation (§V-B) to trace timestamp-annotated memory accesses, and the execution engine (§V-C) to control thread scheduling. Lastly, we explain the implementation detail of SEGFUZZ (§V-D).

### A. Two-stage Fuzzing

SEGFUZZ's single-thread fuzzing (§V-A1) and multi-thread fuzzing (§V-A2) consist of two components, an input generator and an input executor. We explain each details in the following.

*1) Single-thread fuzzing:* In a single-thread fuzzing stage, the *single-thread generator* produces a single-thread input (referred as $I_{ST}$) in the form of a sequence of random system calls, $(S_1, S_2, ..., S_n)$. And then, the *single-thread executor* runs $I_{ST}$ to identify two system calls in $I_{ST}$ that potentially exhibit new interleaving segment coverage. If identified, $I_{ST}$ will be passed to the next stage, the multi-thread fuzzing.

**Single-thread generator.** Similar with a conventional fuzzing, the single-thread generator constructs a single-threaded input $I_{ST}$ with two strategies, generation and mutation. When using the generation strategy, SEGFUZZ randomly generates a system call sequence based on well-formed system call description grammar `Syzlang` [18], which describes templates of available system calls including types of arguments, a range of feasible values of each argument, and the type of a return value. With `Syzlang`, SEGFUZZ produces a single-thread input by repeatedly selecting a random system call and providing reasonable arguments of the system call. The mutation strategy is an alternative of the generation strategy. When using a mutation strategy, SEGFUZZ picks up an already-generated single-thread input, and modifies the single-thread input by appending additional system calls, removing existing system calls, or changing values of arguments of existing system calls.

**Single-thread executor.** Given $I_{ST}$ from the single-thread generator, the single-thread executor runs $I_{ST}$ while tracing basic blocks and timestamp-annotated memory accesses executed by each system call. After the execution is finished, the single-thread executor computes branch coverage, and if $I_{ST}$ exposes new branch coverage that has not been explored, SEGFUZZ keeps $I_{ST}$, and feeds it back to the single-thread generator so that the single-thread generator further mutates $I_{ST}$ to find more branch coverage.

Moreover, the single-thread executor identifies a pair of system calls in $I_{ST}$ that *potentially* exposes new interleaving segment coverage if executed concurrently. Specifically, for each system call pair $(S_i, S_j)$ in $I_{ST}$, the single-thread executor computes a set of explored segment graphs $G$ with memory accessed executed by $S_i$ and $S_j$ (as described in §IV-B), and checks if $G$ contains new segment graphs that have not been explored. If so, the single thread executor passes $I_{ST}$ as well as $(S_i, S_j)$ and $G$ to the multi-thread fuzzing. The multi-thread fuzzing will split $I_{ST}$ into two system call sequences with the purpose of running $S_i$ and $S_j$ concurrently.

**CVE-2017-17712 in single-thread fuzzing.** To discover CVE-2017-17712 demonstrated in Figure 1, let us assume the single-thread generator produces $I_{ST}$ as a sequence of three system calls `r0 = socket()`, `setsocktopt(r0)`, `sendmsg(r0)` as described in Figure 7. Then, the single-thread executor runs this $I_{ST}$, and collects memory accesses executed by two system calls `setsockopt(r0)` and `sendmsg(r0)`. Since these memory accesses are annotated with timestamps, the single-thread executor identifies that all memory accesses executed by `setsockopt(r0)` are followed by memory accesses executed by `sendmsg(r0)`, and computes a set of segment graphs $G$, which includes (`B1` ⇒ `A2` ⇒ `A4`) (*i.e.*, Figure 2-(a)). Assuming (`B1` ⇒ `A2` ⇒ `A4`) has been unseen before, SEGFUZZ determines that it is worth exploring interleavings between the two system calls. Thus, the single-thread executor passes $I_{ST}$ as well as the two system calls and $G$ to the next stage.

*2) Multi-thread fuzzing:* After $I_{ST}$ is passed with $(S_i, S_j)$ and $G$, the *multi-thread generator* transforms $I_{ST}$ to a multi-thread input $I_{MT}$. In addition, the multi-thread generator produces *schedules*, according to the mutation-based interleaving exploration (§IV-C). The *multi-thread executor* then tests each schedule of $I_{MT}$ with a support of the execution engine (§V-C), and collects interleaving segment coverage (§IV-B).

**Multi-thread generator.** Given $I_{ST}$ and $(S_i, S_j)$, the multi-thread generator first produces $I_{MT}$, which preserves all system calls in $I_{ST}$, but executes $S_i$ and $S_j$ concurrently. Specifically, assuming $i < j$, the multi-thread generator splits $I_{ST}$, which is a single sequcne of system calls $(S_1, S_2, ..., S_i, ..., S_j, ..., S_{n-1}, S_n)$, into two sequences of system calls, $(S_1, S_2, ..., S_i)$ and $(S_{i+1}, ...S_j, ..., S_n)$. Then, the multi-thread executor runs the two system call sequences in two different threads, while runs $S_i$ and $S_j$ concurrently. A detailed example is provided in Appendix C.

Also, the multi-thread generator repeatedly produces various schedules. To generate schedules, the multi-thread generator

implements the mutation-based interleaving exploration method detailed in §IV-C. Briefly explaining, given a set of explored segment graphs $G$, the multi-thread generator mutates segment graphs in $G$ to derive a set of unexplored mutated segment graphs $G_{mutated}$, and then selects a subset of $G_{mutated}$ called $G^*_{mutated}$ to generate a schedule. The multi-thread executor then runs $I_{MT}$ while enforcing the generated schedule.

**Multi-thread executor.** The multi-thread executor runs $I_{MT}$ while enforcing a schedule generated by the multi-thread generator. During the execution of $I_{MT}$, the multi-thread executor checks if the running thread interleaving causes a harmful behavior (*e.g.*, memory corruption or deadlock) with a support from devloper tools such as lockdep [44], kernel watchdog [46], and sanitizers [24, 41, 45, 73]. If these developer tools detect that an abnormal behavior occurs in the kernel, the multi-thread executor writes a report of the abnormal behavior as well as $I_{MT}$ and the executed thread interleaving.

Otherwise, with memory accesses executed by $S_i$ and $S_j$, the multi-thread executor computes a set of executed segment graphs $G'$, and probes for interleaving segment coverage as described in §IV-B. In addition, the multi-thread executor feeds $G'$ to the multi-thread generator, allowing the multi-thread generator to further explore thread interleavings.

**CVE-2017-17712 in multi-thread fuzzing.** Once receiving $I_{ST}$, the multi-thread generator transforms $I_{ST}$ into $I_{MT}$. In this example, thread A executes two system calls, `r0 = socket` and `setsockopt(r0)`, and thread B executes one system call `sendmsg(r0)`, while two system calls `setsockopt(r0)` and `sendmsg(r0)` will be executed concurrently (as described as $I_{MT}$ in Figure 7). Furthermore, the multi-thread generator produces a schedule according to the mutation-basd interleaving exploration. Given $G$ including (`B1` $\Rightarrow$ `A2` $\Rightarrow$ `A4`), the multi-thread generator produces a mutated segment graph corresponding to (`A2` $\Rightarrow$ `B2` $\Rightarrow$ `A4`) (refer Figure 5). Then, the multi-thread generator generates a schedule to test (`A2` $\Rightarrow$ `B1` $\Rightarrow$ `A4`) (refer Figure 6). Lastly, when the multi-thread executor runs $I_{MT}$ with the generated schedule, the uninitialized access bug is triggered.

### B. Kernel Instrumentation

SEGFUZZ requires to trace basic blocks (for computing code coverage) and timestamp-annotated memory accesses (for computing interleaving coverage) executed by each system call. For this, SEGFUZZ incorporates a compiler pass that inserts callback function calls 1) at the entry of each basic block, and 2) before each instruction that accesses globally-visible memory objects. At the entry of a basic block, the callback function records the starting address of the basic block into a per-thread memory region, which is shared between the user space and the kernel space through mmap. Thus after a thread executes a system call, the thread can identify executed basic blocks by reading the memory region. Similar to the case of basic block, at instructions that access globally-visible memory objects, the callback function records five tuples (*i.e.*, the address of the memory object, the instruction address, the size and the type of the memory access, and the timestamp) into another per-thread memory region, allowing a thread to identifying memory accesses executed by a system call.

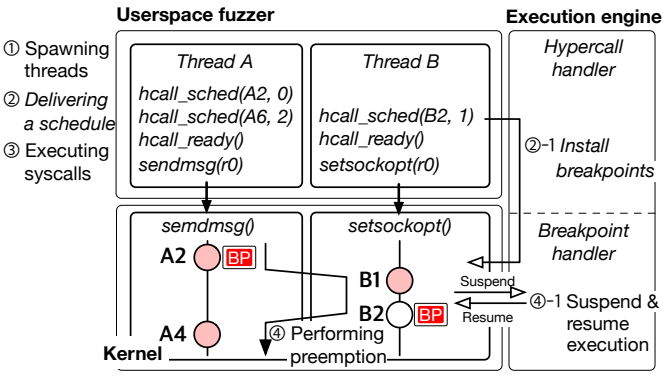


**Fig. 8:** The workflow of the execution engine.

### C. Execution Engine

The execution engine grants an ability to the fuzzer process that it can control thread scheduling. It is implemented in the hypervisor layer to be non-intrusive to the kernel execution.

**Enforcing a schedule.** To request the execution engine to enforce a schedule, the fuzzer process sends a schedule through hypercall interfaces before executing system calls. During the execution, the execution engine suspends and resumes the execution of system calls as described in the given schedule.

Figure 8 shows a workflow of the execution engine. At the beginning, the fuzzer process spawns threads, where each thread is assigned a system call and scheduling points (① in Figure 8). Then, each thread invokes hypercalls (*i.e.*, `hcall_sched()`) to deliver scheduling points to the execution engine (②). When the execution engine receives a scheduling point, the execution engine installs a breakpoint on an instruction on which the scheduling point refers (②-1). It is worth noting that `hcall_sched()` takes the order of scheduling points as a second paramter. In this example, preemption will occur first on `A2`, followed by `B2` and `A6` in order during the execution. After delivering scheduling points, each thread calls `hcall_ready()`, which sleeps until all threads call are ready, and after all threads call `hcall_ready()`, the execution engine wakes them up, and the threads start executing system calls (③). While executing system calls, the execution engine keeps only one thread to run. When the running thread reaches a scheduling point, the execution engine performs preemption (④) by suspending the running thread and resuming the next thread to run as specified in the given schedule (④-1).

**Performing preemption.** In order to perform preemption, SEGFUZZ leverages the hardware breakpoint feature [27] shipped in modern Intel CPU chipsets. During the execution, the execution engine recognizes that the execution reaches a scheduling point when a breakpoint is hit. When a thread hits a breakpoint, the execution engine stores context information (*e.g.*, register values) of the running thread in the hypervisor's memory, and changes the program counter of the thread to an infinite loop called a trampoline. In the trampoline, the

thread keeps calling a kernel API `cond_resched()` which yields a CPU. As a consequence, the thread is suspended in the trampoline without making a progress. To resume the suspended thread, the execution engine restores registers including a program counter with the values stored when the thread was suspended, and then the thread can continue its execution.

**Restriction of hardware breakpoint.** While SEGFUZZ often needs more than four scheduling points, the number of hardware breakpoints that can be installed simultaneously is limited (*e.g.*, four in Intel CPU chipsets). SEGFUZZ overcomes this restriction by leveraging the fact that scheduling points are ordered. Specifically, the execution engine installs breakpoints on first four scheduling points, and when an installed breakpoint is hit, the execution engine moves a breakpoint onto the next scheduling point.

**Handling missing scheduling points.** During the execution, instructions on which scheduling points install might not be executed, for example, if the control flow changes due to kernel internal states. In such case, the execution engine does not mess up the order of scheduling points. For example, when executing a schedule of Figure 8, it is possible that thread A may skip `A2`, and hit a breakpoint installed on `A6`. If so, the execution engine ignores all scheduling points before `A6` (*i.e.*, `A2` and `B2`), and keeps enforcing a schedule after `A6`.

**Virtual Machine Instrospection (VMI).** The execution engine introspects the target kernel for two reasons. First, because a breakpoint does not distinguish which thread hits the breakpoint, the execution engine needs to determine whether the breakpoint is hit by a thread of the fuzzer process, or by an irrelevant thread. Second, when a running thread tries to acquire a lock, the execution engine inspects whether the lock has been held by a suspended thread, which may cause the unexpected block. While the VMI is crucial to properly control thread scheduling, we leave details of our VMI in Appendix D.

### D. Implementation

We implement SEGFUZZ in various software layers as follows: The SEGFUZZ's two-stage fuzzing (§V-A) is implemented based on `Syzkaller` [20], with 3334 [2] LoC in GoLang and 341 LoC in C++. The kernel instrumentation (§V-B) is implemented in two parts, a compiler pass and callback functions. The compiler pass is implemented on the LLVM compiler suite 12.0.1 [69] with 323 LoC in C++, and, the callback functions are implemented in the Linux kernel source tree with 265 LoC in C. Lastly, the execution engine (§V-C) is implemented on QEMU 6.0.0 with 1662 LoC in C, and leverages KVM (Kernel-based Virtual Machine) to take advantage of hardware acceleration. We have open-sourced the implementation of SEGFUZZ in https://github.com/casys-kaist/segfuzz.

## VI. EVALUATION

We demonstrate the usefulness of SEGFUZZ by 1) providing newly found concurrency bugs in the recent Linux kernels (§VI-A), 2) quantitatively comparing SEGFUZZ against prior concurrency fuzzing techniques (§VI-B), and 3) analyzing performance characteristics of SEGFUZZ (§VI-C).

[2]We use scc [6] and sloccount [85] to measure LoC of GO and C, C++ respectively.

### A. Finding Real-world Concurrency Bugs

**Experimental setup.** We run SEGFUZZ on a two-socket machine equipped with Intel(R) Xeon(R) CPU E5-2683 v4 @ 2.10GHz (32 physical cores) and 512 GB of RAM. We run Ubuntu Server 20.04.4 LTS on Linux 5.4.143 as a host operating system. During our experiments, we launch 32 virtual machines (VMs) where each VM is equipped with four vCPUs and 8 GB memory. We use a Linux kernel configuration used by `Syzkaller` [20] so that `Syzkaller` and SEGFUZZ search the same kernel modules/subsystems. The kernel versions we run on SEGFUZZ ranges from 5.19-rc2 to 6.2.

**Newly found concurrency bugs.** During our evaluation period, SEGFUZZ discovers 83 unique crash titles including ones that `Syzkaller` also finds. Among them, 21 are newly identified as harmful concurrency bugs, and three (*i.e.*, `#2`, `#3`, and `#5`) were reported by `Syzkaller` after few month later. The result is summarized in Table I. This result shows that SEGFUZZ is able to find bugs across the entire kernel layers from specific device drivers (*e.g.*, `#1`, and `#3`) to various network subsystems (*e.g.*, `#8`, `#12`, and `#21`). Unlike KRACE, which focuses on kernel file systems, SEGFUZZ is not tailored to specific subsystems. SEGFUZZ discovers not only less-harmful bugs such as warnings (*e.g.*, `#12`) but also critical bugs such as memory corruptions (*e.g.*, `#2`, `#10`, and `#13`). Interestingly, based on the root cause confirmation by kernel developers, we find that some bugs has been in the kernel for a long time, but are never identified by other fuzzing systems. `#10` (*i.e.*, `KASAN: use-after-free Read in slip_ioctl`) was present in the kernel since 2013, and `#11` (*i.e.*, `general protection fault in add_wait_queue`) was present since 2011. These cases demonstrate that SEGFUZZ is capable of discovering hard-to-find concurrency bugs.

### B. Comparison with prior approaches

We compare SEGFUZZ against prior approaches to answer the following questions: 1) is interleaving segment coverage (§IV-B) informative and helpful in discovering concurrency bugs? (**Design goal 1** in §III) (§VI-B1), and 2) how efficient is the mutation-based interleaving exploration (§IV-C) in exploring the search space of thread interleavings? (**Design goal 2**) (§VI-B2). To evaluate the points, we collect well-known concurrency bugs as a test set, and then compare SEGFUZZ and prior approaches on the test set.

**Bug selection.** Table II shows concurrency bugs for the comparison study. Our concurrency bug selection criteria are 1) concurrency bugs are studied in previous studies [17, 28, 38, 68, 86], and 2) we can find patches fixing concurrency bugs so that we can inject them into a kernel. We exclude two Android-specific concurrency bugs (*i.e.*, CVE-2019-1999 [59] and CVE-2019-2025 [60]) evaluated in ExpRace [38]. To run tests on a consistent environment, we use the same kernel of

| ID | Kernel Version | Subsystem | Crash Type | Crash Summary | Status |
|---|---|---|---|---|---|
| #1 | 5.19-rc2 | drivers/misc/vmw_vmci | general protection fault | general protection fault in vmci_host_poll | being fixed |
| #2 | 5.19-rc2 | net/caif | use-after-free access | KASAN: use-after-free Read in cfusbl_device_notify | fixed |
| #3 | 5.19-rc2 | drivers/net/can/slcan | use-after-free access | KASAN: use-after-free Read in slcan_receive_buf | fixed |
| #4 | 5.19-rc2 | net/netfilter | general protection fault | general protection fault in cttimeout_net_exit | fixed |
| #5 | 5.19-rc2 | kernel | use-after-free access | KASAN: use-after-free Read in raw_notifier_call_chain | fixed |
| #6 | 5.19-rc3 | kernel/trace | task hung | INFO: task hung in blk_trace_remove | confirmed |
| #7 | 5.19-rc3 | kernel/trace | task hung | INFO: task hung in blk_trace_setup | confirmed |
| #8 | 5.19-rc3 | net/key | assertion violation | kernel BUG in pfkey_send_acquire | reported |
| #9 | 6.0-rc7 | kernel/sched | general protection fault | general protection fault in add_wait_queue_exclusive | being fixed |
| #10 | 6.0-rc7 | drivers/net/slip | use-after-free access | KASAN: use-after-free Read in slip_ioctl | confirmed |
| #11 | 6.0-rc7 | kernel/sched | general protection fault | general protection fault in add_wait_queue | being fixed |
| #12 | 6.0-rc7 | net/can | warning | WARNING in isotp_tx_timer_handler | being fixed |
| #13 | 6.0-rc7 | sound/core/oss | use-after-free access | KASAN: use-after-free Read in snd_pcm_plug_read_transfer | reported |
| #14 | 6.0-rc7 | mm | assertion violation | Kernel BUG in find_lock_entries | reported |
| #15 | 6.0-rc7 | net/ipv4 | use-after-free access | KASAN: use-after-free Read in tcp_write_timer_handler | confirmed |
| #16 | 6.2-rc1 | kernel/events | use-after-free access | KASAN: use-after-free Read in event_sched_out | fixed |
| #17 | 6.2-rc1 | drivers/video/fbdev | general protection fault | general protection fault in soft_cursor | reported |
| #18 | 6.2-rc1 | kernel/events | use-after-free access | KASAN: use-after-free Read in perf_event_groups_insert | fixed |
| #19 | 6.2-rc7 | drivers/usb/core | invalid page fault | BUG: unable to handle kernel paging request in usb_start_wait_urb | reported |
| #20 | 6.2-rc7 | fs/kernfs | invalid page fault | BUG: unable to handle kernel paging request in __kernfs_new_node | reported |
| #21 | 6.2 | net/ipv4 | general protection fault | general protection fault in raw_seq_start | being fixed |

**TABLE I:** List of concurrency bugs newly discovered by SEGFUZZ. In the `Status` column, "being fixed" means that a patch is submitted but has not been merged into the mainline kernel at the time of writing.

| ID | Vulnerability | Subsystem | Crash Type | Reference |
|---|---|---|---|---|
| Vul #1 | CVE-2016-8655 [52] | net/packet | use-after-free access | [28, 38] |
| Vul #2 | CVE-2017-2636 [55] | drivers/tty | double-free | [28, 38] |
| Vul #3 | CVE-2017-7533 [56] | fs/notify | slab-out-of-bound acc. | [38, 68] |
| Vul #4 | CVE-2017-17712 [54] | net/ipv4 | uninitialized access | [28, 38] |
| Vul #5 | CVE-2017-15649 [53] | net/packet | use-after-free access | [86] |
| Vul #6 | CVE-2018-12232 [57] | net | NULL dereference | [68] |
| Vul #7 | CVE-2019-6974 [61] | virt/kvm | use-after-free access | [38] |
| Vul #8 | CVE-2019-11486 [58] | drivers/tty | use-after-free access | [38] |
| Vul #9 | 69e16d01d1de [16] | net/l2tp | NULL dereference | [17] |

**TABLE II:** Known concurrency bugs that are studied in previous works, MoonShine [68], Razzer [28], ExpRace [38], FUZE [86], and Snowboard [17].

| ID | Trials #0 | #1 | #2 | #3 | #4 | #5 | #6 | #7 | #8 | #9 | Avg. exec. |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Vul #1 [52] | ✓ | ✓ | - | ✓ | - | - | ✓ | - | ✓ | ✓ | 9.4 |
| Vul #2 [55] | - | - | - | - | - | - | - | - | - | - | 8.4 |
| Vul #3 [56] | - | - | - | - | - | - | - | - | - | - | 32 |
| Vul #4 [54] | - | - | - | - | - | - | - | - | - | - | 21.5 |
| Vul #5 [53] | ✓ | - | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | - | 12 |
| Vul #6 [57] | - | - | - | - | - | - | - | - | - | - | 16.0 |
| Vul #7 [61] | - | - | - | - | - | - | - | - | - | - | 10.4 |
| Vul #8 [58] | - | - | - | - | - | - | - | - | - | - | 6 |
| Vul #9 [16] | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | 9.3 |

**TABLE III:** Trade-off between the bug-finding capability and the search complexity. Out of 10 trials using the less-informative interleaving coverage metric, ✓ indicates that a bug is triggered, while - indicates that a bug is not triggered. The `Avg. exec.` column denotes that the average number of execution until the saturation of interleaving coverage.

version v6.0-rc7 and make the bugs available by rolling back patches that addressed concurrency bugs.

**Comparison method.** We measure the number of executions and the elapsed time required to discover each concurrency bug. However, these metrics heavily depend on initial seeds and the process of mutating multi-threaded input seeds, which may vary across evaluations and disturb a fair fuzzing evaluation [37]. To confine such noisy randomness, we manually provide a multi-threaded input as well as a pair of system calls that triggers each concurrency bug. Therefore, we evenly compare how accurately and quickly fuzzers can discover concurrency bugs when the same bug-triggering inputs are given. We limit the number of executions to 10000 for each measurement, which is large enough for our evaluation.

*1) Comparison of interleaving coverage metrics:* We implement a different coverage metric in SEGFUZZ and compare the version with SEGFUZZ using interleaving segment coverage.

**Comparison target.** We choose alias coverage as a comparison target since it is an interleaving coverage metric most relevant to interleaving segment coverage (*i.e.*, both describe interleavings of instructions). However, alias coverage is implemented only for file systems, so we emulate alias coverage in SEGFUZZ by limiting the number of vertices of each segment graph to two. Hence, like alias coverage, interleaving segments of size 2 contains a single interleaving between two instructions where at least one of them is a write instruction. Using the emulated alias coverage, we run SEGFUZZ to see whether each concurrency bug is triggered until the emulated alias coverage is saturated.

**Result.** Table III shows the result. When using emulated alias coverage, *many (six ouf of nine) concurrency bugs are not discovered even if emulated alias coverage is completely saturated*. Whereas using interleaving segment coverage, SEGFUZZ can discover *all* listed concurrency bugs before the saturation of interleaving segment coverage in *every* trial. The example of CVE-2017-17712 described in §III (*i.e.*, Figure 1) explains why such results come out; emulated alias coverage considers only interleavings between two instructions, which is *not sufficient* to execute the offending interleaving containing multiple instructions. We manually identify that all listed concurrency bugs are caused by thread interleavings of three or four instructions, which supports our design choice on the size of interleaving segments (§IV-A). On the other hand, alias coverage is saturated quickly. With our mutation-based interleaving exploration, it is saturated after 13.9 executions on average, ranging from 6 to 32.

**Bug-finding capability and search complexity trade-off.** The result indicates the trade-off between the bug-finding capability and the search complexity; Alias coverage shows the *lower*

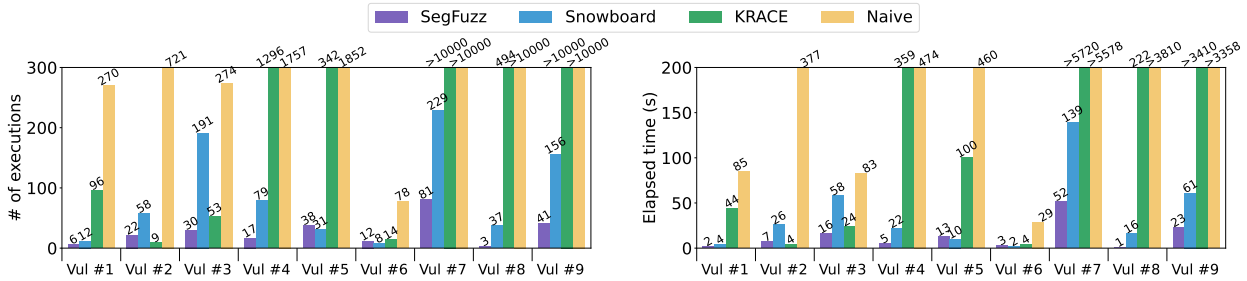


**Fig. 9:** The number of executions and the elapsed time (secs) required for each fuzzer to discover given concurrency bugs. The `Naive` column indicates the kernel's scheduler (*i.e.*, no thread scheduling control applied).

search complexity while it also shows the *weaker* bug-finding capability. Whereas, interleaving segment coverage shows the *stronger* bug-finding capability but it sets the larger search space than the alias coverage, causing higher search complexity. Nevertheless, we claim that its search complexity is still tractable to perform the mutation-based interleaving exploration, and we validate our claim in the next evaluation (§VI-B2).

*2) Efficiency of mutation-based interleaving exploration:* One could argue that even if interleaving segment coverage is informative to describe offending thread interleavings, its search space is too large to explore. However, it is tractable to run the mutation-based interleaving exploration using interleaving segment coverage. To demonstrate this, we compare the mutation-based interleaving exploration (§IV-C) against various interleaving exploration methods proposed in prior approaches.

**Comparison target.** We compare SEGFUZZ to state-of-the-art kernel concurrency fuzzers, Snowboard [17], and KRACE [88]. Unfortunately, we cannot directly run Snowboard and KRACE for the comparison study; KRACE is implemented only for file systems, and Snowboard runs based on the binary translation (*i.e.*, TCG) of QEMU without utilizing the hardware acceleration (*i.e.*, KVM), and thus, its original implementation is not suitable for the elapsed time comparison with SEGFUZZ. Therefore, we implement their approaches on the multi-thread fuzzing phase (§V-A2) of SEGFUZZ by applying 1) the random delay injection scheme of KRACE, and 2) the enforcing-single-interleaving-order scheme used by Snowboard. In addition to them, we also compare with the kernel scheduler (*i.e.*, no thread scheduling control applied) as the naive baseline.

**Discovering concurrency bugs.** Figure 9 shows the comparison result when discovering concurrency bugs. In these figures, `SegFuzz`, `Snowboard`, and `KRACE` display the number of executions and the elapsed time taken and `Naive` corresponds to the kernel scheduler without any thread scheduling control enabled. In `Naive`, we identify the difficulty of discovering varies from a bug to a bug. For example, CVE-2018-12232 appears as the easiest concurrency bug to discover since it can be discovered within 78 executions even with the naive kernel scheduler. On the other hand, the kernel scheduler fails to discover three concurrency bugs, CVE-2019-6974, CVE-2019-11486, and `69e16d01d1de` within 10K times of executions. Regardless of the varying difficulty, SEGFUZZ can discover all of concurrency bugs within a short time. SEGFUZZ can discover given concurrency bugs within just 26.8 runs (ranging from 3 to 81) and 13 seconds (ranging from 1 to 52) on average. Whereas, Snowboard discovers concurrency bugs within 89 runs (ranging from 8 to 229 runs) and 37.5 seconds (ranging from 2 to 139) on average, and KRACE discovers them, if successful, within 329.1 runs (ranging from 9 to 1296) and 107 seconds (ranging from 4 to 359) on average. Moreover, KRACE even suffers from discovering CVE-2019-6974 and `69e16d01d1de`.

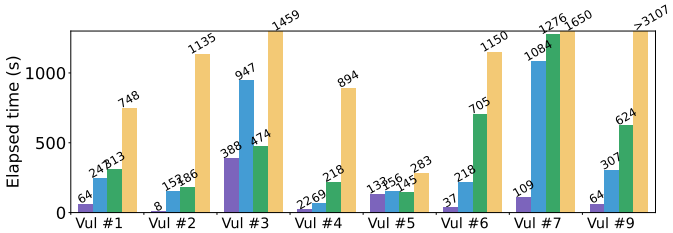


**Fig. 10:** Elapsed time to exhaustively test given multi-threaded inputs after applying patches that fix concurrency bugs.

**Exhaustively testing multi-threaded inputs.** While Figure 9 shows that SEGFUZZ outperforms previous approaches when discovering concurrency bugs within given multi-threaded inputs, we wonder whether SEGFUZZ is also efficient when exhaustively testing multi-threaded inputs until interleaving segment coverage is saturated, but the testing does not cause concurrency bugs. These are common cases when a fuzzer tests inputs. To evaluate this, we re-apply all patches to fix concurrency bugs in Table II and measure how much time is required until saturating interleaving segment coverage of given multi-threaded inputs. In this experiment, `Vul #8` is excluded because its patch simply disables the vulnerable subsystem.

Figure 10 shows the result. As shown in the figure, SEGFUZZ demonstrates elapsed time that is 7.1 times faster than Snowboard and 11.1 times faster than KRACE. While this result compares each approach when testing a single multi-threaded input, it is worth noting that the performance benefit of SEGFUZZ is *cumulative* across long-running fuzz testing. During fuzzing, a fuzzer needs to run a significant number of multi-threaded inputs. For example, during our evaluation, SEGFUZZ generates more than 60,000 inputs. As shown in Figure 10, SEGFUZZ can save 298 seconds for testing a single multi-threaded input on average compared to Snowboard, and this save keeps accumulated as a fuzzer keeps running. In this perspective, our evaluation demonstrates that SEGFUZZ has remarkable benefits over other approaches.

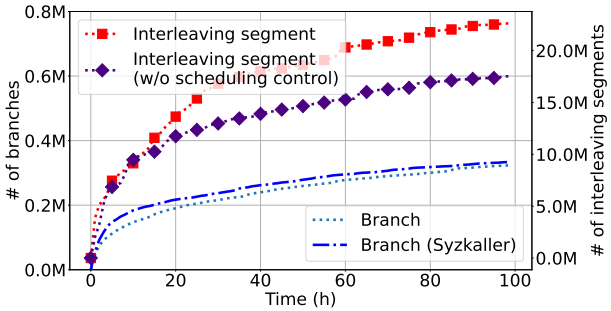


**Fig. 11:** Coverage growth of SEGFUZZ.

In summary, these analysis results confirm that the mutation-based interleaving exploration exhibits superior performance by quickly navigating the search space even if interleaving segment coverage constructs more complex search space than alias coverage.

### C. Performance characteristics of SEGFUZZ

This section provides performance characteristics of SEGFUZZ in coverage growth, fuzzing throughput, and per-input overhead.

**Coverage growth.** Since coverage metrics are the paramount performance metric of fuzzing, we measure the coverage growth for both code coverage (*i.e.*, the number of taken branches) and interleaving coverage (*i.e.*, the number of observed interleaving segments) during 100 hours of fuzzing. To see how much SEGFUZZ improves thread interleaving exploration, we disable the thread scheduling control in the multi-thread fuzzing phase of SEGFUZZ (*i.e.*, random thread schduling), and measure the interleaving coverage. The result is described in a line denoted by `Interleaving segment w/o scheduling control` in Figure 11. With the thread scheduling control disabled, SEGFUZZ finds 29.1% less interleaving segment coverage during the same period. To confirm that this improvement comes from actual performance benefits, we repeat the same experiment with 24 hours duration, and measure the *p*-value using Mann-Whitney U test [3, 37, 67]. The *p*-value is 0.03, which is lower than a conventional threshold of 0.05, indicating that the observed improvement is likely caused by a the performance benefit of SEGFUZZ rather than randomness. As a consequence, we can conclude that our design choices significantly improve in exploring thread interleavings.

In addition, since SEGFUZZ invests the computing power to repeatedly execute a multi-threaded input, we expect that SEGFUZZ might explore code coverage less than the baseline `Syzkaller`. To see the difference of the code coverage exploration in SEGFUZZ and `Syzkaller`, we measure code coverage of `Syzkaller` and illustrate it as a line denoted by `Branch (Syzkaller)` in Figure 11. As a result, SEGFUZZ finds 3.2% less code coverage compared to the baseline `Syzkaller`. This is a definitely downside of SEGFUZZ. However, considering the huge benefit in exploring thread interleavings, we still believe that this is marginal.

| SEGFUZZ | Syzkaller | Syzkaller-memtrace |
|---|---|---|
| 4.55 | 8.40 | 4.74 |

**TABLE IV:** Fuzzing throughput (# of exec/s) of SEGFUZZ and `Syzkaller`. `Syzkaller-memtrace` indicates throughput of `Syzkaller` with memory access tracing enabled.

| | | Comp. overhead (§IV) | | Runtime overhead (§V) | |
|---|---|---|---|---|---|
| Total | Exec. syscall | Tracking coverage (§IV-B) | Interleaving search (§IV-C) | Tracing accesses (§V-B) | Thread scheduling (§V-C) |
| 267.2 | 107.6 | 8.9 | 17.2 | 90.7 | 42.8 |

**TABLE V:** Elapsed time (ms) for executing one multi-thread input. We decompose the elapsed time into the system call execution (`Exec. syscall`), SEGFUZZ's computational overheads (`Comp. overhead`) and runtime overhead (`Runtime overhead`).

**Fuzzing throughput.** All SEGFUZZ's mechanisms provide benefits in finding concurrency bugs with a cost of additional overheads and throughput degradation. To comprehend the trade-off, we measure the fuzzing throughput of SEGFUZZ and compare it with the `Syzkaller`'s throughput. In order to experiment in the same environment, we measure throughput with an empty set of seed. Because both `Syzkaller` and SEGFUZZ restart VMs after an hour of fuzzing, we measure throughput in an hour of execution in order to eliminate noises caused by, for example, VM rebooting or kernel crashes.

Table IV shows the result. SEGFUZZ shows the lower throughput than `Syzkaller`. In particular, the SEGFUZZ's throughput is about 54% of the `Syzkaller`'s throughput. To further understand why the SEGFUZZ's throughput is degraded, we additionally measure throughput of `Syzkaller` while tracing memory accesses (through instrumentation described in §V-B), but not making use of it. As shown in the `Syzkaller-memtrace` column in Table IV, it shows the throughput similar to that of SEGFUZZ; the `Syzkaller-memtrace`'s throughput is just 4.1% higher than the throughput of SEGFUZZ. These results indicate that the throughput of SEGFUZZ is mainly degraded by the heavy instrumentation to trace memory accesses. However, as KRACE [88] states, it can be understandable at the cost for the high input quality. In the fuzzer's perspective, while tracking memory accesses has negative impacts on throughput, it is important to have a higher randomness for a fuzzer to execute more interesting inputs (*i.e.*, unexplored thread interleavings) to discover concurrency bugs. The effectiveness of high input quality is more pronounced in §VI-B2, showing SEGFUZZ can discover concurrency bugs very quickly.

**Per-input overhead.** In SEGFUZZ, there are two types of overheads for executing a multi-thread input, *i.e.*, computational overheads and runtime overheads. Specifically, computation overheads are caused by tracking interleaving segment coverage after executing the input (§IV-B), and calculating scheduling points before executing the input (§IV-C). On the other hand, runtime overheads are caused by tracing memory accesses (§V-B) and controlling thread scheduling (§V-C). To closely examine these overheads, we measure the elapsed time

for executing a single multi-thread input, and break down the elapsed time into each overhead. For this measurement, we run 10 thousands times and take an average.

Table V shows the result. When executing a single input, the total elapsed time is 267.2ms. During the execution, the part that took the longest time is executing system calls; it takes 107.6ms. However, overheads incurred by SEGFUZZ is not negligible. Tracing memory accesses (§V-B) takes 90.7ms, and controlling thread scheduling (§V-C) takes 42.8ms. These two runtime overheads almost double the execution time, and are the main cause of degrading the throughput of fuzzing as shown in the above. In contrast, the total amount of time for computation is 26.1 (= 8.9 + 17.2)ms, and occupies approximately 9% of the total elapsed time. Accordingly, we can see that the computational overhead is relatively small.

## VII. DISCUSSSION

**Larger interleaving segment.** While we restrict the size of each interleaving segment to four, a small fraction of concurrency bugs (*e.g.*, 8 out of 105 from the survey study [49]) are triggered with interleaving segments with size of larger than four. Thus, interleaving segment coverage falls short in tracking offending interleavings of this kind of concurrency bugs. Nonetheless, the mutation-based interleaving exploration is still able to trigger them via recomposing multiple interleaving segments. Since this kind of bugs also may severely affect the security of the kernel, we leave how to track their offending interleavings using interleaving coverage as future work.

**Kernel background threads.** Some system calls can spawn a kernel background thread (*e.g.*, `kworkerd`), and cause a concurrency bug with it. However, SEGFUZZ does not utilize interleaving segment coverage for kernel background threads. Thus, it may be inefficient for SEGFUZZ to find concurrency bugs in kernel background threads. Nonetheless, we emphasize that this limitation is not stemmed from our design choices but from the lack of a mechanism to trace basic blocks and memory accesses in kernel background threads. We expect that SEGFUZZ can be further improved if we devise the tracing mechanism, and leave this part as future work.

## VIII. RELATED WORK

In this section, we discuss prior efforts to automatically discover concurrency bugs in the kernel.

**Kernel fuzzing.** To discover vulnerabilities in the kernel, fuzzing, specifically coverage-guided fuzzing [20, 22, 33, 35, 36, 68, 70, 74–76, 80, 89], has proven to be practical and is widely used in industrial fields. Then, to further improve a kernel fuzzing, a number of attempts have been made to incorporate advanced techniques [35, 68, 75], or to expand the input space beyond syscalls [36, 74, 89]. Although they all achieve meaningful successes, they are limited in exploring thread interleavings, which raises a demand for discovering concurrency bugs in the kernel.

**Controlled concurrency testing (CCT).** CCT [1, 7, 9, 14, 29, 40, 62–64, 78] introduces an idea of overriding the kernel scheduler and methodically testing thread interleavings of a *given* input. To the best of our knowledge, concurrency fuzzing stems from CCT with an idea of merging *the test case generation* (*i.e.*, fuzzing) and *the thread interleaving exploration* (*i.e.*, CCT). Specifically, Razzer [28] states that it is inspired by SKI [14], a CCT technique aiming the kernel. SEGFUZZ is also affected by CCT techniques, and improves the concurrency bug-finding capability by adopting interleaving segment coverage and mutation-based interlaving exploration.

**Data race detection.** A large volume of work [5, 13, 42, 50, 65, 66, 72, 79, 90, 91] are proposed to detect data races, a *subset* of concurrency bugs [2, 43], while there are also non-data race concurrency bugs. For example, during our evaluation, SEGFUZZ can easily trigger non-data race concurrency bugs (*e.g.*, CVE-2019-6974, `69e16d01d1de`) that cannot be detected using a data race detector, while data race detectors can detect non-memory corruption bugs such as semantic bugs (detailed in Appendix E). In addition, data race detectors are orthogonal to the design of SEGFUZZ, and thus, the SEGFUZZ's concurrency-bug detection capability can be agumented if deployed together with a data race detector.

**Double-fetch bugs.** Several work [4, 71, 81, 83, 84, 87] are proposed to discover a certain type of concurrency bugs, called double-fetch bugs [82]. Double-fetch bugs manifest when one thread tries to fetch a *single* data twice while another thread modifies the data in the middle of fetches. While SEGFUZZ is able to identify other concurrency bugs that are not double-fetch bugs, SEGFUZZ may not be effective in detecting certain double-fetch bugs, such as ones that shared data resides in *userspace* memory (as described in Appendix F). Therefore, we believe SEGFUZZ and previous work to discover double-fetch bugs complement each other.

## IX. CONCLUSION

In this paper, we propose SEGFUZZ, a novel and effective kernel concurrency fuzzer. Its key improvements reside in 1) adopting informative interleaving coverage called interleaving segment coverage, and 2) mutation-based interleaving exploration which utilizes explored thread interleavings. As a result, SEGFUZZ has discovered 21 previously-undisclosed concurrency bugs, and a comparison study demonstrates that SEGFUZZ outperforms state-of-the-art concurrency fuzzers.

## X. ACKNOWLEDGMENT

We sincerely appreciate anonymous reviewers and our anonymous shepherd for their constructive and valuable comments. This work was supported in part by ERC (NRF-2018R1A5A1059921) funded by the Korea government(MSIT), the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2021-0-00871), and Samsung Electronics.

## REFERENCES


[1] M. Abdelrasoul. Promoting secondary orders of event pairs in randomized scheduling using a randomized stride. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana-Champaign, IL, Oct.–Nov. 2017.

[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] A. Arcuri and L. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33th International Conference on Software Engineering (ICSE)*, Honolulu, HI, May 2007.

[4] A. Bhattacharyya, U. Tesic, and M. Payer. Midas: Systematic kernel TOCTTOU protection. In *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[5] 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.

[6] B. Boyter. Sloc Cloc and Code (scc), 2020. https://github.com/boyter/scc/.

[7] 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.

[8] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[9] Y. Cai and Z. Yang. Radius aware probabilistic testing of deadlocks with guarantees. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Singapore, Sept. 2016.

[10] 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.

[11] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Newport Beach, CA, Mar. 2011.

[12] A. Choudhary, S. Lu, and M. Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, May 2017.

[13] 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.

[14] 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.

[15] G. Fowler, L. C. Noll, and K.-P. Vo. FNV Hash, 2022. http://isthe.com/chongo/tech/comp/fnv/index.html.

[16] S. Gong. net: fix a concurrency bug in l2tp_tunnel_register(), 2012. https://github.com/torvalds/linux/commit/69e16d01d1de4f1249869de342915f608feb55d5.

[17] 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.

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

[19] Google. The Go Programming Language, 2022. https://pkg.go.dev/hash/fnv.

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

[21] R. Guo and J. Zeng. Trace Me if You Can: Bypassing Linux Syscall Tracing, 2022. https://www.blackhat.com/us-22/briefings/schedule/index.html#trace-me-if-you-can-bypassing-linux-syscall-tracing-26427.

[22] 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.

[23] H. Han, R. Jian, X. Wang, and P. Zhou. Android Universal Root: Exploiting Mobile GPU / Command Queue Drivers, 2022. https://www.blackhat.com/us-21/briefings/schedule/index.html#typhoon-mangkhut-one-click-remote-universal-root-formed-with-two-vulnerabilities-22946.

[24] W. Han, B. Joe, B. Lee, C. Song, and I. Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[25] C. Helbling. Directed graph hashing. *CoRR*, abs/2002.06653, 2020. URL https://arxiv.org/abs/2002.06653.

[26] Z. Huang, S. Guo, M. Wu, and C. Wang. Understanding concurrency vulnerabilities in linux kernel, 2022.

[27] Intel Corporation. Hardware and Software Breakpoints, 2020. https://software.intel.com/en-us/node/676419.

[28] 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.

[29] 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.

[30] 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.

[31] X. Jin, R. Neal, C. Resell, and C. Lecigne. Monitoring Surveillance Vendors: A Deep Dive into In-the-Wild Android Full Chains in 2021, 2021. https://www.blackhat.com/us-22/briefings/schedule/index.html#monitoring-surveillance-vendors-a-deep-dive-into-in-the-wild-android-full-chains-in--26629.

[32] X. Jin, R. Neal, and J. Bottarini. Android Universal Root: Exploiting Mobile GPU / Command Queue Drivers, 2022. https://www.blackhat.com/us-22/briefings/schedule/index.html#android-universal-root-exploiting-mobile-gpu--command-queue-drivers-27239.

[33] D. Jones. Trinity: Linux system call fuzzer., 2012. https://github.com/kernelslacker/trinity.

[34] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Y. Lee, B. Lee, and C. Min. Exploiting Kernel Races through Taming Thread Interleaving, 2022. https://www.blackhat.com/us-20/briefings/schedule/index.html#exploiting-kernel-races-through-taming-thread-interleaving-20223.

[40] 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.

[41] 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.

[42] Linux. The kernel concurrency sanitizer (kcsan), 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/dev-tools/kcsan.rst.

[43] Linux. Explanation of the Linux-Kernel Memory Consistency Model, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/memory-model/Documentation/explanation.txt.

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

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

[46] Linux. Linux Watchdog Support, 2022. https://www.kernel.org/doc/html/latest/watchdog/index.html.

[47] K. Lu, M.-T. Walter, D. Pfaff, S. Nümberger, W. Lee, and M. Backes. Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[48] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST) (FAST 13)*, San Jose, CA, Feb. 2013.

[49] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[50] 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.

[51] R. C. Merkle. A digital signature based on a conventional encryption function. In *Proceedings of the 6th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Amsterdam, The Netherlands, Apr. 1987.

[52] MITRE Corporation. Cve-2016-8655, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655.

[53] MITRE Corporation. Cve-2017-15649, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15649.

[54] MITRE Corporation. Cve-2017-17712, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17712.

[55] MITRE Corporation. Cve-2017-2636, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2636.

[56] MITRE Corporation. Cve-2017-7533, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533.

[57] MITRE Corporation. Cve-2018-12232, 2018. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12232.

[58] MITRE Corporation. Cve-2019-11486, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11486.

[59] MITRE Corporation. Cve-2019-1999, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1999.

[60] MITRE Corporation. Cve-2019-2025, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-2025.

[61] MITRE Corporation. Cve-2019-6974, 2019. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6974.

[62] 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.

[63] 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.

[64] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Beijing, China, June 2012.

[65] 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.

[66] 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.

[67] R. L. Ott and M. T. Longnecker. *An introduction to statistical methods and data analysis*. Cengage Learning, 2015.

[68] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

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

[70] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[71] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features. In *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Incheon, Korea, June 2018.

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

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

[74] 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.

[75] 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.

[76] 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.

[77] V. Terragni and M. Pezzè. Effectiveness and challenges in generating concurrent tests for thread-safe classes. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Gothenburg, Sweden, May–June 2018.

[78] 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.

[79] 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.

[80] 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.

[81] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[82] P. Wang, K. Lu, G. Li, and X. Zhou. A survey of the double-fetch

vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6):e4345, 2018.

[83] P. Wang, K. Lu, G. Li, and X. Zhou. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. *Frontiers of Computer Science*, 13: 247–263, 2019.

[84] W. Wang, K. Lu, and P.-C. Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[85] D. A. Wheeler. SLOCCount, 2020. https://dwheeler.com/sloccount/.

[86] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou. FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[87] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

## APPENDIX

### A. Algorithms to Decompose and Recompoase Thread Interleavings

**Algorithm 1:** Generating a DAG $(V, E)$

**Input** : $S_i, S_j$: sequences of memory access executed by two system calls
**Output** : $(V, E)$: a directed-acyclic graph

$V, E_{po}, E_{io} = \emptyset, \emptyset, \emptyset$
**for** $S \in \{S_i, S_j\}$ **do**
  **for** $a \in S$ **do**
    $V = V \cup \{vertex(a)\}$
  **end**
  ▷ `Generate program-order edges`
  **for** $a_1, a_2 \in S \times S$ **do**
    **if** $a_1.timestamp < a_2.timestamp$ **then**
      $E_{po} = E_{po} \cup (vertex(a_1), vertex(a_2))$
  **end**
**end**
▷ `Generate interleaving-order edges`
**for** $a_1, a_2 \in (S_i \times S_j) \cup (S_j \times S_i)$ **do**
  **if** $memory\_access\_overlapped(a1, a2)$ **then**
    **if** $a1.timestamp < a2.timestamp$ **then**
      $E_{io} = E_{io} \cup (vertex(a_1), vertex(a_2))$
    **else**
      $E_{io} = E_{io} \cup (vertex(a_2), vertex(a_1))$
    **end**
**end**
$E = E_{po} \cup E_{io}$

*1) Algorithm to generate a DAG:* Algorithm 1 describes an algorithm for generating a DAG for two system calls $(S_i, S_j)$. It first generates vertices corresponding memory accesses (`line #3`∼`#5`), and program-order edges for each system call (`line #6`∼`#9`). Then, to describe a thread interleaving, it generates an interleaving-order edge $(a_1, a_2)$ if they are overlapped; $a_1$ and $a_2$ 1) are executed different threads (`line #11`), 2) access the same data where at least one of them is write (`line #12`). Lastly, it returns a DAG $(V, E)$ where $E$ is an union of program-order edges and interleaving-order edges.

**Algorithm 2:** Decomposing a DAG $(V, E)$ into $G$

**Input** : $(V, E)$: a directed-acyclic graph
**Output** : $G$: a set of segment graphs

$G = \emptyset$
**for** $e_1, e_2 \in E \times E$ **do**
  ▷ `Pick two interleaving-order edges and compose a subgraph` $g$
  **if** $is\_io\_edge(e_1) \wedge is\_io\_edge(e_2) \wedge e_1 \neq e_2$ **then**
    $v = vertices(e_1) \cup vertices(e_2)$
    $e = \emptyset$
    **for** $v1, v2 \in v \times v$ **do**
      **if** $(v1, v2) \in E$ **then**
        $e = e \cup \{(v1, v2)\}$
      **end**
    **end**
    $G = G \cup \{(v, e)\}$
**end**

*2) Algorithm to decompose a DAG:* Algorithm 2 describes an algorithm to decompose a given DAG $(V, E)$ to a set of segment graphs. To this end, it selects two edges $e_1, e_2 \in E$ (`line #2`). If these two edges are both interleaving-order edges and they are different (`line #3`), it generates a segment graph $(v, e)$, where $v$ is a set of all vertices connected by $e_1$ and $e_2$ (`line #4`), and $e$ is a set of all edges connecting vertices in $v$ (`line #5`∼`#8`). To check an edge $e$ is an interleaivng-order edge in (`line #3`), `is_io_edge()` returns true if two vertices connected by $e$ are executed by different threads. Lastly, it gathers all segment graphs into $G$ (`line #10`) and returns $G$.

*3) Algorithm to recompose segment graphs:* Algorithm 3 describes our greedy algorithm to recompose segment graphs. Specifically, it iterates over $G_{mutated}$ (`line #2`), and for each mutated segment graph $g_{mutated}$, it checks whether adding $g_{mutated}$ into a recomposed DAG $(V, E)$ forms a loop or not. To this end, it checks that adding each edge $e$ from a selected mutated graph $g_{mutated}$ to a recomposed DAG $(V, E)$ forms a loop or not (`line #4`∼`#7`). Although it can be further optimized, we use BFS to check a loop is formed in `loop_detected()` (`line #5`). If it is confirmed that adding $g_{mutated}$ does not form a loop in $(V, E)$, it combines $g_{mutated}$ into $(V, E)$ (`line #11`) and remove $g_{mutated}$ from $G_{mutated}$ (`line #12`). After all process is finished, SEGFUZZ conducts a topological sort on $(V, E)$ to generate scheduling points.

### B. Hashing Segment Graph

In order to distinguish different interleavings of the same vertices, we adopt Merkel hashing [25, 51]. In particular, given

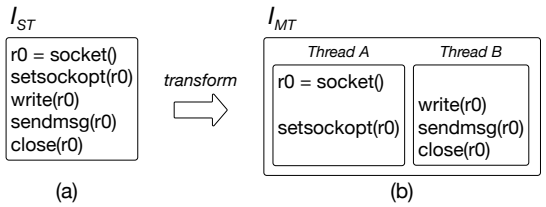


**Fig. 13:** Example of the input transformation conducted by the userspace fuzzer (§V-A). $I_{MT}$ is a multi-thread input to trigger an uninitialized access bug described in Figure 1.

**Algorithm 3:** Recomposing segment graphs

**Input** : $G_{mutated}$: a set of mutated segment graphs
**Output** : $(V, E)$: a recomposed DAG

$V, E = \emptyset, \emptyset$
**for** $g_{mutated} \in G_{mutated}$ **do**
  $conflict = false$
  ▷ Check whether adding $g_{mutated}$ into $(V, E)$ forms a loop
  **for** $e \in edges(g_{mutated})$ **do**
    **if** $loop\_detected(V \cup vertices(e), E \cup \{e\})$ **then**
      $conflict = true$
      break
  **end**
  **if** $\neg conflict$ **then**
    ▷ Add $g_{mutated}$ into $(V, E)$
    **for** $e \in edges(g_{mutated})$ **do**
      $V, E = V \cup vertices(e), E \cup \{e\}$
      $G_{mutated} = G_{mutated} \setminus \{g_{mutated}\}$
    **end**
**end**

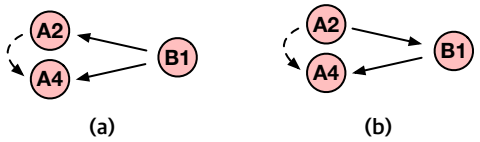


**Fig. 12:** Two different interleaving segment graphs representing different interleavings of `A2`, `A4`, and `B1` from Figure 1.

a graph $G = (V, E)$, for all vertices $v \in V$, we first calculate a hash value of $v$, $hash(v)$, which reflects its out-going edges. Let us denote $o_1, ..., o_m$ as vertices that is connected by an edge $v \rightarrow o_x$. Then, $hash(v)$ is calculated as follow:

$$hash(v) = \mathcal{H}(v.label{+}{+}o_1.label{+}{+}...{+}{+}o_m.label)$$

where $\mathcal{H}$ denotes the non-cryptographic FNV hash function [15, 19], and $++$ denotes the label-concatenation operation. Then, $hash(G)$ is defined as follows:

$$hash(G) = \bigoplus_{v \in V} hash(v)$$

where $\oplus$ denotes the XOR operation.

Applying $hash(G)$ to Figure 12, $hash(\texttt{B1})$ is calculated as $\mathcal{H}(\texttt{B1}++\texttt{A2}++\texttt{A4})$ in Figure 12-(a), whereas, $hash(\texttt{B1})$ is calculated as $\mathcal{H}(\texttt{B1}++\texttt{A4})$ in Figure 12-(b). Thus, hash values of even vertices labeled same (*i.e.*, corresponding to the same instruction) are calculated differently according to edges connected to the vertex. Likewise, values of $hash(\texttt{A2})$ are also calculated differently in the two graphs as the direction of edges connecting `A2` and `B1` is different in the two graphs. As a consequence, values of $hash(G)$ (which are calculated as $hash(\texttt{B1}) \oplus hash(\texttt{A2}) \oplus hash(\texttt{A4})$) of Figure 12-(a) and (b) are different, and SEGFUZZ can distinguish the two graphs in Figure 12 using hash values of the two graphs.

### C. Input Transformation

Figure 13 demonstrates an example of the input transformation. Let us assume $I_{ST}$ is generated as a sequence of system calls as described in Figure 13-(a), where `setsockopt(r0)` and `sendmsg(r0)` executes source codes in Figure 1. When $I_{ST}$ is executed, the single-thread executor identifies that `setsockopt(r0)` executed `B1`, and `sendmsg(r0)` executed `A2` and `A4`, and their execution order was (`B1` $\Rightarrow$ `A2` $\Rightarrow$ `A4`). Then, the single-thread executor identifies that the segment graph has not been explored, and consequently, $I_{ST}$ is passed to the multi-thread fuzzing phase.

When the multi-thread generator receives $I_{ST}$ and the two system calls `setsockopt(r0)` and `sendmsg(r0)`, the multi-thread generator simply splits $I_{ST}$ into two parts, where the first part conatins system calls from the first system call until `setsockopt(r0)`, and the second part contains the rest of the system calls. Then, the multi-thread generator assigns the first part to `Thread A` and the second part to `Thread B` (Figure 13-(b)). During runtime, all system calls are executed in the same order as in $I_{ST}$ except `setsockopt(r0)` and `sendmsg(r0)`. Specifically, `thread A` executes `r0 = socket()` first as it is the first system call in $I_{ST}$. And then, when `thread A` sees the next system call is `setsockopt(r0)`, it defers the execution of `setsockopt(r0)` to execute it with `sendmsg(r0)`. As the next system call in $I_{ST}$ is `write(r0)`, `thread B` executes it. When the next system call in $I_{ST}$ is `sendmsg(r0)`, two threads execute two system calls `setsockopt(r0)` and `sendmsg(r0)` concurrently. Lastly, `thread B` executes `close(r0)` which is the last system call in $I_{ST}$. In this way, SEGFUZZ continuously transforms $I_{ST}$ to $I_{MT}$, and the multi-thread fuzzing stage keeps exploring thread interleavings of $I_{MT}$.

### D. Virtual Machine Introspection

The execution engine introspects the target kernel for two reasons: 1) the execution engine determines whether the breakpoint is hit by a thread of the userspace fuzzer, or by an irrelevant thread, and 2) when a running thread tries to acquire a lock, the execution engine inspects whether the lock is held by a suspended thread.

As a hardware breakpoint does not distinguish the running context of a kernel, if the context switch happens, a breakpoint may be hit by another thread or an interrupt handler, making the execution out of expectation. The execution engine recognizes a running context using `task_struct` which holds the thread description, and the per-cpu `preempt_count` variable indicating what context the thread is in (*e.g.*, a task context for running a syscall, or a hardIRQ context to handle hardware

| | Thread A | | Thread B |
|---|---|---|---|
| | /* In kvm_ioctl_create_device() */ | | /* In __close_fd() */ |
| A1 | smp_store_release(fd[fd], file); | | |
| | | B1 | file = smp_load_acquire (fd[fd]); |
| | | B2 | if (!file) |
| | | B3 | return; |
| | | B4 | if (dec_and_test (obj->refcnt)) |
| | | B5 | kfree(obj); |
| A2 | atomic_inc(obj->refcnt); | | |

**Fig. 14:** Example of non-data race concurrency bug (CVE-2019-6974) found in the KVM submodule.

interrupts). If a breakpoint is hit by a context other than the fuzzer-controlled thread, the execution engine ignores it and keeps the breakpoint.

In addition, when the suspended thread already acquires a lock while the running thread wants to hold the same lock, the whole execution cannot make a progress, because the execution engine forces the lock-holding thread to suspend. Therefore, the execution engine inspects whether the running thread is going to be blocked due to the lock contention, and if it is, the execution engine takes control from the running thread to the suspended thread. Inspecting the lock contention is conducted by hooking lockdep functions (*i.e.*, `lock_acquire()` and `lock_release()` in `kernel/locking/lockdep.c`) [44] that are commonly called from synchronization primitives. When the lockdep functions are called, the execution engine determines whether the running thread can make a progress through various information such as the address of the synchronization primitive, and operation type (*i.e.*, lock, unlock, and trylock). If the running thread cannot make a progress, the execution engine perform preemption to prevent unexpected block.

### *E. Non-data race concurrency bug*

Data races are a subset of concurrency bugs as described in the follow.

**Data race.** In this paper, we employ the definition of data race from Linux Kernel Memory Model (LKMM) [43]. According to LKMM, a data race occurs when there are two memory accesses such that 1) they access the same memory location, 2) at least one of them is a store, 3) at least one of them is not annotated with special APIs such as `WRITE_ONCE()`, or `smp_load_acquire()`, 4) they occur on different CPUs (or in different threads on the same CPU), and 5) they execute concurrently.

**Example of non-data race concurrency bug.** However, there are many non-data race concurrency bugs. Figure 14 shows the examples of non-data race concurrency bug (*i.e.*, CVE-2019-6974). In Figure 14, there is no data race because reading from and writing to `fdt->fd[fd]` and `obj->refcnt` is annotated with dedicated APIs such as `smp_load_acquire()` and `dec_and_test()`. In the perspective of data race detectors, all concurrent accesses are out of the definition of data race, thus, data race detectors report nothing with this example. However, depending on interleaving of the two functions `kvm_ioctl_create_device()` and `__close_fd()`, a use-after-free bug may occur as described in the figure.

**Scope of this paper.** SEGFUZZ mostly focuses on finding concurrency bugs (including data races) that exhibit harmful behaviors (*e.g.*, memory corruptions). Thus, SEGFUZZ can easily detect Figure 14, while data race detectors cannot.

### *F. Detailed discussion of double-fetch bugs with examples*

In Figure 15, the two threads share the data through userspace memory (pointed by &ucmsg->len). Specifically, thread A fetches a length value pointed by `&ucmsg->len` twice from the userspace. Since thread A checks the sanity of the value only on the first read, if thread B modifies the value in the middle of fetches, a buffer-overflow bug manifests when executing `copy_from_user()`. Regarding this kind of double-fetch bugs, SEGFUZZ may be ineffective to discover. This is because SEGFUZZ does not trace memory accesses taken place in userspace, and SEGFUZZ does not track interleaving segments across kernel memory and userspace memory.

| | Thread A - kernel space | | Thread B - user space |
|---|---|---|---|
| | // check the sanity of ucmsg->len | | |
| A1 | get_user(ucmlen, &ucmsg->len); | | |
| A2 | if (!sanity_check(ucmlen)) | | |
| A3 | return -EINVAL; | | |
| | | | // modify the value pointed |
| | | | // by ucmsg->len |
| | | B1 | ucmsg->len = 0xffffffff; |
| | // Re-read the value | | |
| A4 | get_user(ucmlen, &ucmsg->len); | | |
| A5 | copy_from_user(buf, uptr, ucmlen); | | |

**Fig. 15:** A code snippet of a double-fetch bug from [81] that SEGFUZZ is ineffective in finding. `&ucmsg->len` points to a memory object in userspace.

**Initially *vmci* is not intiaizlied**

| | Thread A | | Thread B |
|---|---|---|---|
| | // Read uninitilized pointer | | |
| A1 | ptr = vmci->context; | | |
| | | | // Initialize vmci |
| | | B1 | vmci->context = ctx_create(); |
| | | B2 | vmci->ct_type = CONTEXT; |
| A2 | if (vmci->ct_type == CONTEXT) | | |
| A3 | ptr->host_context; | | |

**Fig. 16:** A code snippet of a concurrency bug (#1 in Table I) that is found by SEGFUZZ but is not a double-fetch bug.

On the other hand, Figure 16 shows a concurrency bug that was found by SEGFUZZ and is not a double-fetch bug. In this example, thread A should read vmci->context *after* checking the value of `vmci->ct_type` in order to guarantee that the value read through `vmci->context` is initialized when dereferencing. However in the buggy scenario, thread A incorrectly reads an uninitialized value of `vmci->context`, causing a general-protection-fault. This example is not a double-fetch bug, and is out of focus of [4, 81, 87]. SEGFUZZ can be helpful in discovering this kind of non-double-fetch bugs.