Diagnosing Kernel Concurrency Failures with AITIA

Dae R. Jeong
KAIST

Minkyu Jung
KAIST

Yoochan Lee
Seoul National University

Byoungyoung Lee
Seoul National University

Insik Shin
KAIST

Youngjin Kwon
KAIST

Abstract

Kernel concurrency failures are notoriously difficult to identify and diagnose their fundamental reason, the root cause. Kernel concurrency bugs frequently involve challenging patterns such as multi-variable races, data races with asynchronous kernel threads, and pervasive benign races. We perform an in-depth study of real-world kernel concurrency bugs and elicit three requirements: comprehensiveness, pattern-agnostic, and conciseness.

To fulfill the requirements, this paper defines the root cause as a chained sequence of data races, called a causality chain. A causality chain is presented as a *comprehensive* form to explain how a failure eventually happens in the presence of multi-variable races rather than simply pointing out a few instructions related to the root cause. To build a causality chain, this work proposes two practical approaches: Least Interleaving First Search to reproduce a concurrency failure, and Causality Analysis to identify the root cause. Causality Analysis runs the kernel to confirm what data races contribute to the failure among all detected data races. The approach is *pattern-agnostic* because it dynamically tests data races without counting on pre-defined patterns. While testing data races, Causality Analysis rules out failure-irrelevant data races such as benign races, producing a *concise* causality chain.

AITIA is a system implementing the two approaches. By evaluating AITIA with 22 real-world concurrency failures, we show that AITIA can successfully build their causality chain. With AITIA, we found the root causes of six unfixed bugs; three bugs were concurrently fixed, the root causes of three bugs were confirmed by kernel developers.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; **Concurrency control**.

Keywords: Failure diagnosis, Operating system, Concurrency bug, Debugging

EuroSys '23, May 9–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567486

ACM Reference Format:
Dae R. Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. Diagnosing Kernel Concurrency Failures with AITIA. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3567486

1 Introduction

Correcting concurrency bugs is a perennial task in operating systems. Modern operating systems have employed a number of advanced concurrency techniques, such as read-copy-update [63], the pervasive use of reference counting [14], lock-free data structures [102], and deferred works [51] to eschew scalability bottlenecks and make good use of multiple cores. These techniques have played their own roles well in improving performance, but when combined with complex kernel concurrency patterns, they become a major source of concurrency bugs. Concurrency bugs in operating systems are notoriously difficult to fix due to their non-deterministic nature and the tricky reasoning of their parallel executions. Consequently, kernel developers sometimes write incorrect fixes [76, 109] or leave reported bugs unfixed for a long time [18, 32].

There have been many approaches to identify the root causes of concurrency failures in user applications. They determine the root cause using statistical correlation [7, 8, 34, 37, 38, 83, 84], by combining static and dynamic analyses [58, 60, 100], and by comparing common instruction sequences [117] from successful and failed executions of a program. These approaches are meaningful in their relevant domains, but applying them to the kernel is limited due to the following requirements inherent to kernel concurrency bugs.

Comprehensive. Kernel concurrency failures often involve data races of multiple variables, called a *multi-variable race*. In the Linux kernel, multi-variable races exhibit complex interactions. Often, a multi-variable race invokes a non-deterministic control flow, called a *race-steered control flow*. A race-steered control flow is a control flow that can be changed by how preceding data-racing instructions are executed, and thus previously un-executed code can be executed, which triggers another data race, causing a failure eventually. We find that 16 bugs involve race-steered control flows out of 22 real-world concurrency bugs. Therefore, to help developers in their reasoning about kernel concurrency bugs, a root cause diagnosis

	AITIA	Kairux [117]	MUVI [58]	Cooperative Bug Localization			Failure Reproduction	
				Snorlax [38]	Gist [37]	CCI [34]	REPT [16]	RR [78]
Comprehensive (§2.1)	✓	-	△	△	△	△	✓	✓
Pattern-agnostic (§2.2)	✓	✓	-	-	-	-	✓	✓
Concise (§2.3)	✓	✓	✓	✓	✓	✓	-	-

Table 1. Three root cause diagnosis requirements and whether each work satisfies the requirements. ✓means satisfied, and – mean not satisfied respectively. △ means a requirement is conditionally satisfied only when the root cause meets assumptions of each work.

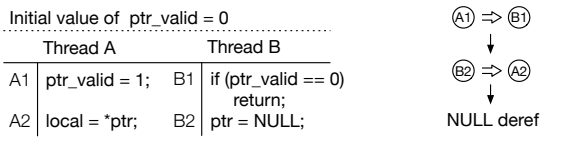
tool must be able to consider the complex interactions of multi-variable races involving race-steered control flows.

Pattern-agnostic. Previous systems rely on predefined bug patterns to identify the root cause. However, it is challenging to express multi-variable races and race-steered control flows as generic patterns due to their complex interactions. Moreover, when they are combined with asynchronous events (e.g., background kernel thread), the problem becomes worse. Thus, a root cause diagnosis tool must not rely on predefined patterns and must reflect the dynamic behaviors of the kernel.

Concise. The Linux kernel contains *benign races* which do not contribute to a failure. Kernel developers intentionally use benign races to improve performance (e.g., statistic counters) [107]. Existing kernel data race detectors such as DataCollider [21] are known to use a significant number of benign races (104 data races out of 113 detected races were benign). Such benign races are a major source of false positives in root cause diagnosis systems, demanding significant manual efforts from developers to rule out benign races. Therefore, a root cause diagnosis tool must not contain failure-irrelevant information such as benign races.

Unfortunately, existing systems have limitations with regard to their ability to meet the three aforementioned goals, as summarized in Table 1.

AITIA is a root cause diagnosis system for kernel concurrency bugs. AITIA satisfies the three requirements. To capture comprehensive information, AITIA defines the root cause as a chained sequence of data races, called a *causality chain*, rather than a simple pattern [37, 38, 83]. A causality chain is presented in a comprehensive form to explain how a failure eventually occurs. Figure 1 shows an example of a failure (*i.e.*, NULL pointer dereference) and its causality chain. Figure 1 shows multi-variable races involving two semantically correlated variables, `ptr_valid` and `ptr` (i.e., a non-zero value of `ptr_valid` indicates that `ptr` contains a valid pointer). The code has a race-steered control flow introduced by the pair (A1) and (B1). To be specific, if (A1) ⇒[1] (B1), (B2) ⇒ (A2) would lead to the NULL pointer dereference. However, if the execution order changes to (B1) ⇒ (A1), the control flow changes as well—now thread B simply returns and it does not incur a failure. As a result, the following data race at (A2) and (B2) does not occur. From the causality chain, developers know the following: (1) two data races exist in the instructions of ((A1); (B1)) and ((A2), (B2)); (2) the failure-causing instruction sequence; (3) a race-steered control flow (B2) ⇒ (A2), which is induced by (A1) ⇒ (B1); and (4) most importantly, how to fix the bug. The causality chain explains, *"If a fix does not allow one of the interleaving orders in the chain, it does not incur a failure."* For example, if a developer patches the code so that it does not execute (A1) ⇒ (B1), the failure will not occur. Likewise, a developer can avoid the failure by not allowing (B2) ⇒ (A2).



(a) A buggy code example (b) Causality chain

Figure 1. An abstract example of a concurrency failure.

To construct a causality chain, AITIA performs two steps: *Least Interleaving First Search (LIFS)* and *Causality Analysis*. From a bug-finding system, AITIA initially requires the inputs, system call traces, and failure information. LIFS aims to reproduce a kernel concurrency failure from the input. LIFS explores different interleavings of kernel instructions to find the instruction sequence that causes the concurrency failure. To reduce the search space, LIFS detects instructions accessing the same memory location (i.e., conflicting instructions), a situation which may cause data races, and considers the interleavings of the conflicting instructions. LIFS adapts the idea of dynamic partial order reduction (DPOR) [22, 113] to prune unnecessary search steps. At the end of the searches, LIFS produces an instruction sequence that deterministically causes a concurrency failure; e.g., (A1) ⇒ (B1) ⇒ (B2) ⇒ (A2) in Figure 1.

Causality Analysis builds a causality chain from the output of LIFS. The intuition is as follows: if an interleaving order of a data race is flipped, one can test whether the data race contributes to the failure or not. For example, in Figure 1, if (A1) ⇒ (B1) is flipped to (B1) ⇒ (A1), a failure will not occur; therefore, Causality Analysis inserts (A1) ⇒ (B1) into the causality chain. For the same reason, (B2) ⇒ (A2) is inserted as well. Causality Analysis systematically flips interleaving orders of data races one at a time to identify all data races related to the failure. To identify whether a data race contributes to the failure, AITIA *executes kernel* following an instruction sequence where a single data race is flipped and tests if kernel fails or not.

[1] In this paper, $X \Rightarrow Y$ denotes an execution order between two instructions such that X is executed before Y.

AITIA dynamically tests data races by running the kernel with different instruction sequences. AITIA does not rely on predefined patterns of bugs (**Pattern-agnostic**). Instead, AITIA relies on concrete evidence of the root cause—the runtime behavior leading to a failure. While testing data races at runtime, AITIA identifies the race-steered control flows and relationships of multiple data races in the form of a causality chain (**Comprehensive**). AITIA tests all identified data races to identify races that actually contribute to a failure, thereby effectively excluding benign races from the root cause (**Concise**).

AITIA is implemented by 6,705 lines of code (LoC) in GO and 3,851 LoC in C in the KVM hypervisor and QEMU. We applied AITIA to 22 real-world kernel concurrency bugs, which were collected from the CVE vulnerability database [99] since 2016 and to a well-known Linux kernel bug-finding system, Syzkaller [26]. AITIA finds the root causes of *six unfixed kernel concurrency bugs*. Patches for three of them were submitted by developers before we reported [50, 53, 54]. We reported the root causes of the remaining three bugs, and they were confirmed by Linux kernel developers.

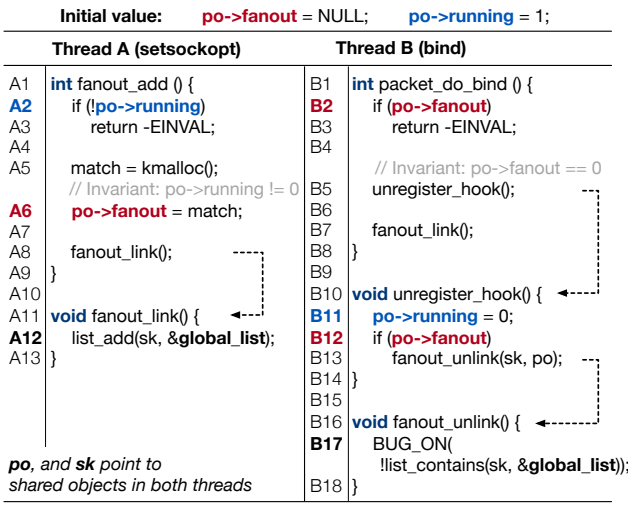


Figure 2. Simplified code snippet of CVE-2017-15649. Dashed lines represent function calls when the failure manifests. Data-racing instructions: (A2, B11), (A6, B2), (A6, B12), (A12, B17). A failure-causing instruction sequence: A2 ⇒ A5 ⇒ B2 ⇒ B11 ⇒ A6 ⇒ B12 ⇒ B17 (BUG_ON()).

2 Concurrency Bugs in the Kernel

We study real-world kernel concurrency bugs to motivate this work. We find that it is challenging to identify the root causes of kernel concurrency bugs due to the following three types of races: 1) multi-variable races, 2) loosely correlated races, and 3) benign data races. We perform an in-depth study about how the three types of kernel concurrency bugs occur and identify the following requirements for diagnosing their root causes.

Comprehensiveness: In the presence of multi-variable races, a diagnosis system should report all necessary information to fix the bugs.

Pattern-agnostic: A diagnosis system should not rely on the use of specific interleaving patterns or assumptions (e.g., correlations among memory objects).

Conciseness: The output should not contain failure-irrelevant information such as benign races.

As shown in Table 1, previous studies only met part of these requirements. To meet the three requirement, we define the root causes as a *causality chain* and propose a system to build a causality chain.

In this section, we describe how these requirements are derived from real-world concurrency bugs and how the causality chain satisfies these requirements while previous studies do not. For clarity, we adapt the definitions of *conflicting* memory accesses and *data race* specified in the Linux kernel memory model [49] throughput this paper; Conflicting memory accesses refer to a case in which two operations access the same memory location and at least one of them is a store operation. A data race is defined as conflicting memory accesses conducted on different CPUs (or in different threads), where two memory accesses are executed concurrently.

2.1 Challenge 1: Multi-variable Race

A multi-variable race is a race bug that multiple variables are *semantically correlated* [58]. To illustrate, let's consider two correlated variables, a pointer variable (which points to a string) and a length variable (which stores the length of the string). When the pointer variable is updated, the length variable must be updated accordingly. Such multi-variable races involves combinations of data races eventually leading to a failure. To understand the root cause of such multi-variable concurrency bugs, a diagnosis system must represent comprehensive interactions of multiple data races.

Real-World Example. we study a real-world concurrency bug, CVE-2017-15649 [69], as an example of a multi-variable data race. This is shown in Figure 2. In this example, two system calls, `setsockopt` and `bind`, communicate via two memory-accessing variables, `po->fanout` and `po->running`. The two variables are semantically correlated; `po->fanout` can be updated only if `po->running` is `1`, and `po->running` can be set to `0` only if `po->fanout` is `NULL`. The correlation is implicitly assumed by developers. If these two memory variables are accessed in an atomic manner, a failure does not occur. However, a failure occurs if a thread interleaves between the time the two correlated variables are accessed, leading to `BUG_ON()` (`B17`). Precisely, the failure occurs in the following sequence.

First, thread A checks whether `po->running` is `0` at `A2`. As an initial value of `po->running` is `1`, thread A keeps executing without returning an error at `A3`. Subsequently, thread B checks whether `po->fanout` is `NULL` at `B2`. Because its initial value is `NULL`, it proceeds up to `B11`. Up to this point, the execution order can be represented as follows: `A2` ⇒ `A5` ⇒ `B2` ⇒ `B11`. Then, thread A stores a value in `po->fanout` at `A6`,

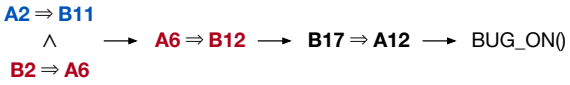


Figure 3. Causality chain of CVE-2017-15649.

violating the multi-variable semantic because `po->running` = 0 at `B11`. However, a failure does not occur at this point.

The execution of `A6` allows thread B to pass the check at `B12`, which in turn calls `fanout_unlink()` at `B13`. This `fanout_unlink()` attempts to remove `sk` from the `global_list`, but this violates the assumption behind `fanout_unlink()`, as `fanout_unlink()` assumes that the to-be-deleted `sk` must be on the linked list `global_list`, but this specific execution order causes `sk` not to be inserted before thread B calls `fanout_unlink()`, causing `BUG_ON()` at `B17`. The instruction sequence `A2` ⇒ `A5` ⇒ `B2` ⇒ `B11` ⇒ `A6` ⇒ `B12` ⇒ `B17` causes the failure.

Race-steered control flow. Note that if `B12` were executed before `A6`, the failure would not occur; because `po->fanout` would be `NULL` at `B12`, thread B would return without calling `fanout_unlink()`. As such, the control flow of thread B depends on the data race of `A6` and `B12`, which we refer to as a *race-steered control flow* caused by a multi-variable race.

Comprehensiveness. We tracked how kernel developers fixed the bug. The developers discovered the root cause of the failure is a multi-variable atomicity violation on `po->running` and `po->fanout`. They wrote a patch that makes `po->running` and `po->fanout` accessed atomically; *i.e.*,, the fix causes `B2` ⇒ `A6` and `A2` ⇒ `B11` not to occur simultaneously. As shown by this fix, developers must understand the complex interactions of multiple data races. To help developers, a diagnosis technique should report the comprehensive interactions of multiple data races caused by race-steered control flows.

Unfortunately, previous approaches have limitations to identify the root cause of multi-variable races; they either cannot capture the complex interactions of multiple data races or can diagnose the root cause partially; a single data race that the most strongly correlated interleaving pattern to the failure. If Kairux [117] is applied to the example, it can point to at most a single instruction (i.e., an inflection point) among the multiple interleavings. Likewise, cooperative bug localization techniques [7, 8, 34, 37, 38, 83, 84] infer the most strongly correlated interleaving from numerous execution traces using well-known bug patterns such as atomicity and order violations. In the example, cooperative bug localization (*e.g.*, Snorlax [38], Gist [37]) will report an order violation in `B17` ⇒ `A12` only. However, enforcing the order `B17` ⇒ `A12` is not a correct fix. Even with such a fix, both threads still can execute `fanout_link()` concurrently (at `A8` and `B7`), resulting in the corruption of `global_list` due to the insertion of a shared object twice.

Causality chain. To summarize, **i)** one should understand how multiple data races cause race-steered control flows and how they eventually lead to a failure. **ii)** the root cause cannot be precisely defined as a single data race or a single instruction. Therefore, we claim that the root cause must be described as a chain of execution orders involving multiple data races. We call this form of root cause a *causality chain*. Figure 3 shows a causality chain of the bug in Figure 2. The chain presents the race-steered control flow of `A6` ⇒ `B12` → `B17` ⇒ `A12`, and how the failure eventually occurs due to the comprehensive interactions of data races. We emphasize that the causality chain provides useful information for developers to fix the failure; it explains, *"If a fix does not allow one of the execution orders in the chain, the `BUG_ON()` does not occur."* In the example, the kernel developers fixed the bug not to allow the multi-variable race of (`B2` ⇒ `A6`) ∧ (`A2` ⇒ `B11`), as described in the causality chain.

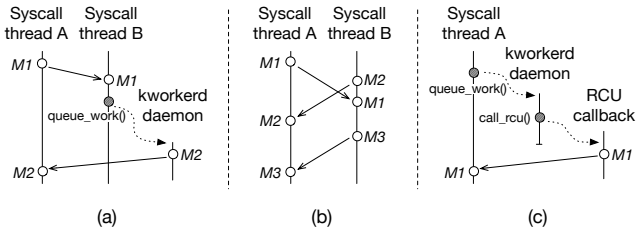


Figure 4. Complex concurrency bug patterns in the Linux kernel. Solid arrows indicate data races, and dotted arrows indicate invocations of kernel background threads. *M1*, *M2* and *M3* represent a memory object.

2.2 Challenge 2: Loosely correlated Objects

To identify multi-variable races, MUVI [58] identifies semantic correlations by means of a statistical analysis. MUVI makes the following key assumption: if two variables have semantic correlation, most of the accesses to these variables should be correlated. More specifically, if one of these two is accessed, the other variable should be accessed with a high probability.

We analyze the execution traces of kernel multi-variable races accessing two objects and find that the two are not accessed together in most cases. We call such objects *loosely correlated*. We observed the following two cases often introduce loosely correlated objects in the Linux kernel: i) two memory objects are often used independently; e.g., one system call only accesses one object, but not the other object, and ii) two memory objects exist in different kernel subsystems. For example, in the CVE-2019-6974 [75], two data races causes a failure: a data race in a file descriptor of a virtual *device* object at the VFS layer and a data race in a virtual *machine* object (*i.e.*, `kvm`) at the KVM hypervisor layer. The objects in which the two data races occur are loosely correlated. Many system calls change the attributes of the virtual device object through its file descriptor (VFS layer) *without accessing* the `kvm` virtual machine object (KVM hypervisor layer).

Pattern-agnostic. Approaches such as MUVI and cooperative bug localization have limitations when diagnosing kernel concurrency bugs involving loosely correlated objects because the root cause of kernel concurrency bugs is out of their predefined patterns or assumptions on which they rely. As shown in Figure 4, kernel concurrency bugs often exhibit complex interleaving patterns involving multiple variables and threads. Pre-defining all possible interleaving patterns is impractical. For this reason, state-of-the-art cooperative bug localization approaches [37, 38] only focus on single-variable concurrency bugs. In contrast, we propose a *pattern-agnostic* approach. The intuition is if one can *dynamically* change the execution order of *only one pair of data-racing instructions* at runtime while the remaining orders remain unchanged, it is possible to identify how the changed execution order contributes to a failure. The approach does not count on specific assumptions or predefined patterns. We refer to this technique as a *casualty analysis* (§3.4).

2.3 Challenge 3: Benign Data Races

In the Linux kernel, complete race-free implementation is not a coding practice. For performance, developers intentionally leave data races in production code called *benign races*, such as updating statistics counters which do not need to provide accurate values, or modifying different bits of flag variables that cause data races but do not incur a failure. According to a recent study on detecting kernel race bugs, more than 50% of newly found concurrency bugs are benign races [23, 107]. The benign race is a major source of false positives [20, 21, 36, 80] in root cause diagnosis systems. Differentiating benign data races from harmful data races is time-consuming and difficult because it requires a careful analysis of the documentation and specific knowledge of the developer's intentions [107]. Therefore, previous approaches use heuristics [107] or replay analysis [80] to filter out benign data races. However, these approaches often incorrectly classify harmful data races as benign ones. According to [80], among the data races that are identified as harmful, around 40% are incorrectly identified, leaving manual inspections of benign race bugs to developers.

Conciseness. Filtering out failure-irrelevant information such as benign races significantly helps developers by reducing their debugging time. Therefore, providing concise information to developers should be one of the important objectives of root cause diagnosis techniques. In contrast, failure reproduction [16, 116] focuses on enabling developers to investigate what transpired during the failed execution. However, identifying the root cause of failed executions remains a tedious and time-consuming task. As discussed in §2.2, we test whether a data race contributes to a failure by dynamically controlling the interleaving order of data races, allowing our approach to exclude benign races effectively when building a causality chain.

3 Causality Inspection for Concurrency Bugs

This section first introduces the key intuition (§3.1) and assumptions (§3.2) behind our approach, after which it explains the two steps needed to build a causality chain, *i.e.*, Least Interleaving First Search (§3.3) and Causality Analysis (§3.4).

3.1 Approach Overview

The proposed system, AITIA, diagnoses the root causes of kernel concurrency bugs. AITIA is not a bug-finding system. AITIA helps developers to patch reported but unfixed kernel concurrency bugs, which is difficult to reason about. We study concurrency bugs in the Linux kernel, but high-level ideas presented in AITIA are generally applicable to other operating systems and user applications. Many previous failure diagnosis systems require an instruction sequence to reproduce a failure as input [60, 61, 117], and AITIA's causality inspection also requires a failure-causing instruction sequence.

The workflow of AITIA is as follows:

1. **Input**: System call traces and failure information
2. **Least Interleaving First Search (LIFS)**: Reproducing the failure (§ 3.3)
3. **Causality Analysis**: Pinpointing the root cause (§ 3.4)
4. **Output**: A causality chain

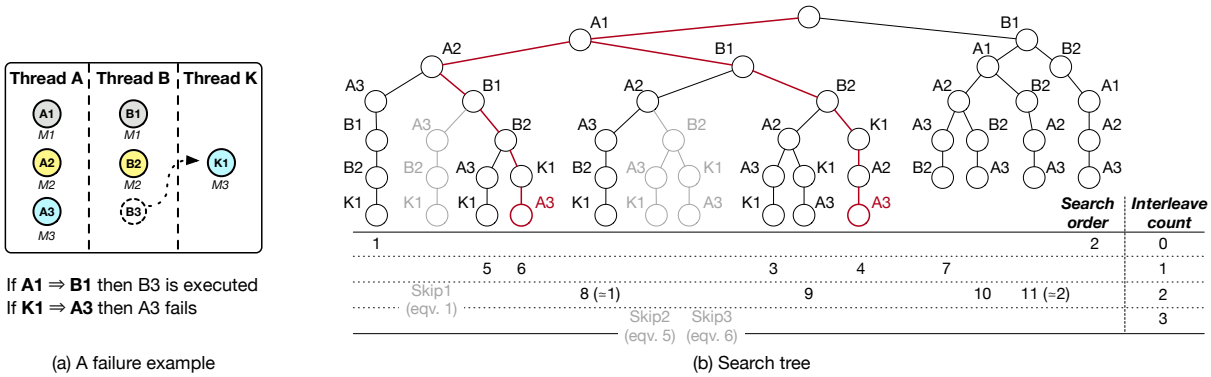
AITIA takes an input from a bug-finding system. With regard to the input, AITIA has two phases: reproducing and pinpointing. To reproduce the failure, AITIA runs a practical algorithm, called *Least Interleaving First Search (LIFS)*. For LIFS, we adopt a well-known technique DPOR [22], but practically tailor it to work in the Linux kernel. LIFS produces a totally ordered instruction sequence that causes a failure (*e.g.*, `A2` $\Rightarrow$ `A5` $\Rightarrow$ `B2` $\Rightarrow$ `B11` $\Rightarrow$ `A6` $\Rightarrow$ `B12` $\Rightarrow$ `B17` in Figure 2).

With this sequence, AITIA runs a novel algorithm, *Causality Analysis*, that systematically tests which data races contribute to the failure, as sketched in § 2.2. From Causality Analysis, AITIA constructs a causality chain—i.e., the root cause.

To apply the two aforementioned algorithms, AITIA needs instruction-level fine-grained controls over the kernel execution. To this end, the two algorithms are designed to specify precise interleaving orders of conflicting instructions, and AITIA mandates the kernel to execute code in such an order using hardware breakpoints. Moreover, AITIA controls various types of execution contexts including system calls, software interrupt handlers (softirq for RCU), and kernel background threads (kworkerd). The details of how AITIA controls the kernel are described in § 4.

3.2 Assumptions in AITIA

AITIA assumes the following: First, a concurrency bug can be observed (*i.e.* no latent bug). If AITIA cannot observe a failure's manifestation, AITIA cannot diagnose the root cause. Second, a failure should be proximate in time to the root cause. AITIA is inefficient (but not unable) to diagnose the root causes of bugs that manifest after a long time. Third,



(a) A failure example

(b) Search tree

Figure 5. A simplified example of a failure and its LIFS search tree: **(a)** shows a failure example. Thread A and thread B execute a system call. Thread K is a kernel background thread (*e.g.* kworkerd). The circles of the same color access the same memory. **(b)** shows a search tree of LIFS. The numbers below the search tree are the search orders of LIFS. Red paths indicate instruction sequences that cause a failure. Grey paths are pruned by partial order reduction.

AITIA assumes the sequential consistency memory model. If there exists a failure-causing instruction sequence, it must deterministically reproduce the failure.

3.3 Least Interleaving First Search

From bug finding tools (e.g., Syzkaller [26]), AITIA obtains timestamped system call traces, the invocation of kernel background threads, and the failure information. Then AITIA selects concurrent system calls and background threads (if invoked) as the input of LIFS. AITIA runs LIFS to reproduce the reported failure. If the failure is not reproduced from the input, AITIA runs LIFS with the next concurrent events.

To reproduce a failure, LIFS explores different interleavings of instructions. Exploring all possible interleavings of all instructions is nearly impossible due to the complexity of such an exploration. This is because only conflicting instructions are meaningful to reproduce the failure [23, 24]. Throughout this paper, we denote `A1(x)` ⇒ `B1(x)` as the execution (or interleaving) order of two conflicting instructions, `A1` and `B1`, accessing the same address x. Also, a data race is presented as such a form.

LIFS search strategies. LIFS has three strategies for exploring possible interleavings of conflicting instructions. First, LIFS searches the instruction sequences from the smallest number of interleavings between threads to the largest. LIFS is based on the observation that most concurrency failures require only a small number of interleavings to manifest [23, 79]. Second, to prune the search space, LIFS adapts the idea proposed by dynamic partial order reduction (DPOR) [22]. While exploring the search space, LIFS detects equivalent instruction sequences, which generate the same result (e.g., no failure produces) and skips the instruction sequence. Third, LIFS searches interleavings of conflicting instructions from front to back. While exploring different interleavings, LIFS *dynamically* identifies new memory-accessing instructions due to race-steered control flow and checks data races between existing instructions and newly detected instructions.

While LIFS is inspired by well-known algorithms such as PCT [11] and DPOR [22, 113], LIFS can be characterized in terms of its practical approach to handle concurrency bugs in large kernel code. In particular, kernel concurrency bugs are often caused by complex, asynchronous interactions between system calls and a kernel background thread (shown in Figure 4). For instance, a kernel background thread is asynchronously invoked only when a race-steered control flow occurs (Figure 4-(a)), and even a single system call can race with kernel background threads resulting in a failure (Figure 4-(c)). In our evaluation, LIFS effectively reproduces all bug patterns described in Figure 4.

LIFS search example. Given the failure presented in Figure 5-(a), Figure 5-(b) depicts how LIFS explores the search space to reproduce. The interleaving count indicates how many interleavings were performed in each search step. LIFS begins the search from the interleaving count 0, continuing to increase the count as it proceeds with the search. **Interleaving count 0.** LIFS simply executes thread A and thread B without interleaving (search order 1 and 2). During this step, LIFS detects all memory-accessing instructions, but LIFS does not know what data races exist. Search order 2 does not include `K1` due to the race-steered control flow. **Interleaving count 1.** LIFS preempts thread A or thread B *exactly once* while performing the search. For example, during search order 4, an interleaving occurs at `A1(m1)` ⇒ `B1(m1)`. After executing threads B and K, LIFS executes the remaining instructions of thread A, generating the following execution order: `A1(m1)` ⇒ `B1(m1)` ⇒ `B2(m2)` ⇒ `K1(m3)` ⇒ `A2(m2)` ⇒ `A3(m3)`. In search order 4, when performing a preemption at `A1(m1)`, LIFS monitors what instructions of thread B access the memory address, `m1` that `A1` accesses. Therefore, after threads A, B, and K finish, LIFS can identify the data racing instructions with respect to `A1` (`{A1, B1}` is identified). Likewise, in search order 5 and 6, a preemption occurs at `A2(m2)` ⇒ `B1(m1)`, and LIFS detects data races with respect to `A2` (`{A2, B2}` is identified). In search order 4, LIFS reproduces the

failures (the data race, K1(m3) ⇒ A3(m3), causes the failure) and terminates. LIFS outputs the failure-causing instruction sequence at step 4, A1(m1) ⇒ B1(m1) ⇒ B2(m2) ⇒ K1(m3) ⇒ A2(m2) ⇒ A3(m3) (fails). After reproducing the failure, LIFS identifies all data races in the failure-causing instruction sequence, which is required for the next step.

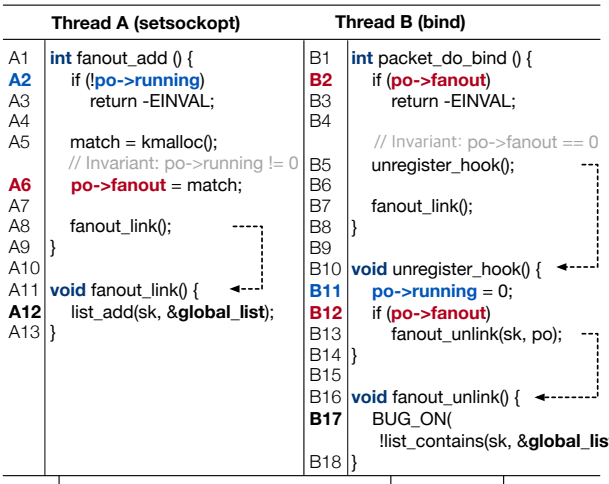
Guarantees of LIFS. LIFS does not assume specific bug patterns to reproduce. LIFS systematically searches for all possible interleavings until it reproduces the failure. LIFS dynamically executes kernel code with the execution order specified by the search algorithm, identifying race-steered control flow and adding these factors to the search space on the fly.

3.4 Causality Analysis

LIFS produces a failure-causing instruction sequence. While performing LIFS, AITIA detects data races in the instruction sequence, but it does not know what data races contribute to the failure. From the instruction sequence, Causality Analysis performs two tests: i) testing *causality to the failure* so as to identify the root cause and ii) *testing causality to other data races* so as to build a causality chain.

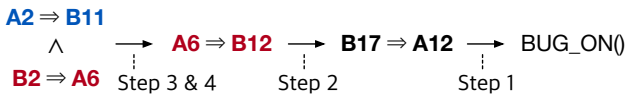
Identifying the root cause. Initially, Causality Analysis creates two sets; the `test set` and the `root cause set`. The test set is initialized using a failure-causing instruction sequence. Each element in the test set specifies a data race with an interleaving order. For example, given that A1(x) ⇒ B1(y) ⇒ B2(x) ⇒ A2(y) is a failure-causing instruction sequence, the test set is initialized as {A1(x) ⇒ B2(x), B1(y) ⇒ A2(y)}. Causality Analysis pops one element at a time from the `test set`, and *flips* the popped interleaving order while the remaining elements' interleaving order remains unchanged. The flipped interleaving order creates a new instruction sequence from the failure-causing execution order. AITIA executes the kernel to follow the new instruction sequence and checks whether the kernel fails or not. If the result is *not failed*, it means that the element (the data race) contributes to the failure, Hence it is added to the `root cause set`. If the kernel still fails, Causality Analysis identifies that the data race is benign. This testing idea is consistent with the formal definition of the root cause; "if removed (flipped in our test), it would prevent a failure from occurring" [105]. Causality Analysis repeats this process until the test set becomes empty.

Causality Analysis is designed to pop an element from backward in the failure-causing instruction sequence (in the example, B1(y) ⇒ A2(y) is selected at first because A2(y) is the last instruction). Testing from backward is convenient because if Causality Analysis starts flipping from forward, a flipped interleaving may cause a new (or known) race-steered control flow, which does not affect the failure or which may cause existing instructions disappear due to by a race-steered control flow.



	Instruction sequence	Flipping	Disappeared
Input	B2 → A2 → A6 → B11 → B12 → B17		
Step 1	B2 → A2 → A6 → B11 → B12 → A12 → B17	B17 → A12	Failure
Step 2	B2 → A2 → B11 → B12 → A6	A6 → B12	B17
Step 3	B2 → B11 → A2 → B12	A2 → B11	A6
Step 4	A2 → A6 → B2	B2 → A6	B11

(a) Details of each step of Causality Analysis.



(b) The constructed causality chain. The corresponding step is written under each arrow.

Figure 6. Causality Analysis steps to construct the causality chain for CVE-2017-15649.

Building a causality chain. The idea behind building a causality chain is identical to identifying the root cause. If A1(x) ⇒ B2(x) has a causality to B1(y) ⇒ A2(y), flipping to B2(x) ⇒ A1(x) will make the data race *disappear* mostly due to a race-steered control flow (i.e., B1(y) ⇒ A2(y) does not occur). Following this intuition, AITIA pops an element from the root cause set (say R1) and runs the kernel with a flipped R1. While running, AITIA attempts to identify any R2 in the root cause set that meets the following two conditions: i) R1 is present in the root cause set, and ii) R2 does not occur. If both conditions are met, AITIA finds that R1 has a causality to R2. Similar to the root cause identification, AITIA builds the causality chains in the backward direction.

Causality chain of CVE-2017-15649. Figure 6 shows how Causality Analysis builds a causality chain. The CVE is the example discussed in § 2.1. LIFS generates B2 ⇒ A2 ⇒ A6 ⇒ B11 ⇒ B12 ⇒ B17 as a failure-causing instruction sequence (denoted as `input` in Figure 6). In this example, there are four data races in the test set:{B2 ⇒ A6, A2 ⇒ B11, A6 ⇒ B12, B17 ⇒ A12}. At step 1, Causality Analysis starts flipping B17 ⇒ A12 as it is the last data race. In this new instruction sequence, thread A inserts `sk` into `global_list`

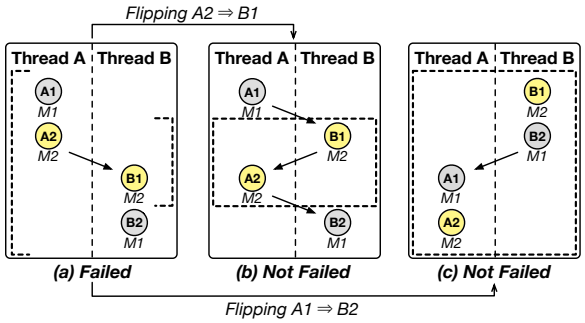


Figure 7. A1 ⇒ B2 surrounds A2 ⇒ B1 in (a).

at A12 before thread B executes BUG_ON() at B17. Therefore, BUG_ON() does not trigger a failure, and Causality Analysis adds B17 ⇒ A12 to the root cause set. At step 2, Causality Analysis flips A6 ⇒ B12 to B12 ⇒ A6. When running the kernel code, Causality Analysis observes that the kernel does not execute B17 because it returns at B13 (a race-steered control flow), such that a failure does not occur. Consequently, A6 ⇒ B12 is added to the root cause set, and Causality Analysis concludes that A6 ⇒ B12 has a causality to B17 ⇒ A12. At step 3, Causality Analysis concludes that A2 ⇒ B11 has a causality to A6 ⇒ B12 because B11 ⇒ A2 (flipped) does not execute A6 (it returns at A3, and no failure occurs). Finally, after analyzing the causality from B2 ⇒ A6 to A6 ⇒ B12, Causality Analysis constructs the causality chain.

Properties of Causality Analysis. Causality Analysis tests all data races in the failure-causing instruction sequence; therefore, Causality Analysis does not cause false-negatives (*i.e.*, it never misses a data race that contributes to the failure). Also, Causality Analysis does not have false-positives; it excludes all benign races during the testing process. In addition, unlike previous statistical approaches, Causality Analysis does not rely on well-known bug patterns, precisely expressing the causalities of multi-variable data races.

Liveness. When Causality Analysis has to interleave a thread holding a lock, Causality Analysis does not progress if the other thread attempts to acquire the lock. To guarantee liveness, when Causality Analysis meets a critical section protected by a lock, Causality Analysis treats the entire critical section as a single data race because the execution order of critical sections may contribute to the failure [59]. Causality Analysis flips the execution order of a critical section as a unit instead of individual instructions in the critical section.

Ambiguity. However, Causality Analysis cannot always flip a single data race. As shown in Figure 7, a data race may surround another data race; A1 ⇒ B2 surrounds A2 ⇒ B1 in Figure 7-(a). We refer to a data race such as A2 ⇒ B1 a *nested data race*. B2 is the last instruction, but Causality Analysis cannot flip A1 ⇒ B2 while preserving the A2 ⇒ B1 interleaving order. In this case, Causality Analysis flips the nested data race first; flipping A2 ⇒ B1 (Figure 7-(b)). Then, Causality Analysis flips the surrounding race, A1 ⇒ B2 (Figure 7-(c)).

An ambiguous case arises when both a nested and a surrounding data race are in the root cause set. For example, in Figure 7, a failure does not occur in (b) or (c). In this case, Causality Analysis cannot determine whether the data race A1 ⇒ B2 truly causes the failure; Causality Analysis cannot determine whether a failure does not occur in (c) because A1 ⇒ B2 is flipped or A2 ⇒ B1 is flipped. Therefore, Causality Analysis reports A1 ⇒ B2 is ambiguous. Note that such an ambiguous case does not arise if the nested data race is not the root cause. If the failure still occurs in (b) and does not occur in (c), we can clearly ascertain that A2 ⇒ B1 causes the failure. In practice, ambiguous cases are not common. In our 18 Linux concurrency bug study, we observe only a single ambiguous case.

4 AITIA

4.1 AITIA Overview

AITIA takes two inputs from a Linux kernel bug finder: timestamped system call traces from a failed execution, and failure information such as a Linux coredump. The current implementation takes inputs from the well-known kernel fuzzing system Syzkaller [26]. AITIA works through three stages: *modeling execution history* (§4.2), *reproducing stage* (§4.3), and *diagnosing stage* (§4.5).

The execution history contains the sequence of system calls and invocation events of kernel background threads such as the deferred work in Linux. They are timestamped to identify concurrent events. After the execution history is modeled, the reproducer produces a failure-causing instruction sequence by LIFS. The execution history contains multiple groups of threads[2] that are executed concurrently, AITIA launches multiple instances of reproducers, each of which is in charge of performing LIFS with a single group of threads. AITIA fully parallelizes the reproducing stage by assigning reproducers for each group. When a reproducer reports a failure-causing instruction sequence, AITIA forwards the result to a diagnoser. In the diagnosing stage, AITIA makes plans regarding which data races are to be flipped according to Causality Analysis. For each flipped data race, AITIA parallelizes the diagnosing stage with multiple instances of diagnosers to run Causality Analysis. Finally, AITIA cleans up the result of the diagnosers and reports a causality chain with instruction-level information, such as line numbers in the kernel.

4.2 Modeling Execution History

By analyzing a crash report, AITIA identifies the symptom of the failure (e.g., kernel panic or watchdog report) and the location of the failure. In the modeling stage, AITIA generates the execution history based on the system call traces from the bug-finding system, which is obtained by enabling

[2]A thread refers to a system call or a kernel background thread.

kernel-event tracing (e.g., ftrace [46] in Linux). More precisely, the execution history consists of i) executed system calls with their parameters and ii) kernel events such as invocations of kernel threads (e.g., RCU kernel thread) with the source of the invocation (a system call, another background thread, or a software interrupt). All entries in the history are annotated with a fine-grained timestamp for AITIA to identify concurrent events.

Consequently, AITIA splits the history into groups of concurrently executed threads, called *slice*. AITIA builds a slice without breaking semantics across system calls; for example, if `write()` or `read()` is in a slice, AITIA adds `open()` and `close()` of the same file descriptor by searching the history. If a slice contains a concurrent event (e.g., two system calls and a kernel background thread), the slice is further split into smaller slices that contain up to three threads.[3] AITIA creates slices backward from the point of a failure because the root cause is likely not far from the failure point, which is the common wisdom in previous failure diagnosis work [16, 38]. A slice may not contain the root cause. If AITIA cannot reproduce the failure, AITIA selects the next slice until the failure is reproduced.

4.3 Reproducing a Failure-Causing Instruction Sequence

In the reproducing stage, AITIA assigns a slice for each reproducer and makes it execute LIFS. To conduct LIFS, the reproducer requires instruction-level controls of the kernel execution to enforce the specified instruction sequence. AITIA uses virtualization to take fine-grained control of the kernel execution. AITIA invokes a guest virtual machine as a reproducer. A reproducer consists of three components: the user agent, guest operating system (OS), and the AITIA hypervisor. The user agent running on a guest OS receives a slice and runs the system calls specified in the slice. If the user agent must execute two concurrent threads, it runs one of the concurrent threads while the other thread is suspended. When running the threads, the user agent collects the address of memory-accessing instructions.

Identifying memory-accessing instructions. AITIA instruments the kernel to obtain control flows of threads. Modern OSes provide coverage testing tools such as kcov [48], which detects basic blocks executed by a thread. These tools allow one to register a callback function triggered at an entry point of all basic blocks, with the callback function then informing the user agents of the basic block address. The user agent has a map of the disassembled kernel code and searches for memory-accessing instructions from the pertinent basic block.

[3]We find that that kernel concurrency failures that occure due to more than four contexts are rare.

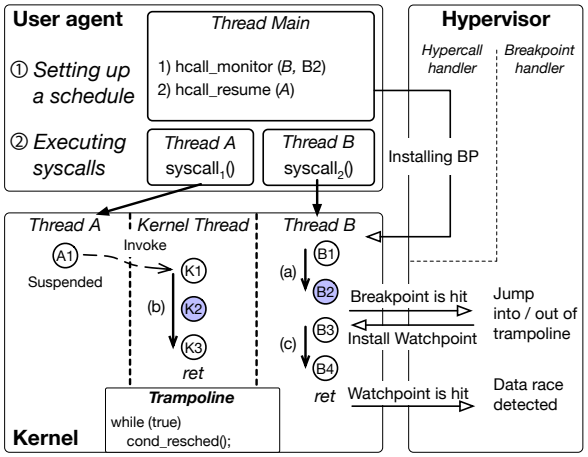


Figure 8. Workflow of AITIA's components. Circles are memory-accessing instructions.

Detecting data races. After identifying memory-accessing instructions, the user agent detects data races while executing LIFS. Figure 8 shows the workflow. To detect a data race with a memory-accessing instruction (Ⓑ2 in Figure 8), the user agent makes a hypercall (`hcall_monitor`) to let the AITIA hypervisor install a breakpoint in the memory-accessing instruction. The user agent starts executing thread B while thread A is suspended. When thread B hits the installed breakpoint, the AITIA hypervisor takes control. The AITIA hypervisor stores context information (e.g., program counter, registers) in the hypervisor's memory, and suspends thread B by directing its program counter to the trampoline code. The trampoline code continually yields its context in a busy loop. The AITIA hypervisor disassembles the kernel code of the memory-accessing instruction to identify the address to which the instruction refers, installs a watchpoint at the memory location, and notifies the user agent to resume thread A. Then, the user agent calls a hypercall to resume thread A (`hcall_resume`). In Figure 8, Thread A invokes a kernel background thread. If an instruction (Ⓚ2) in the kernel thread accesses the memory address indicated by the watchpoint (trapped for the AITIA hypervisor), the AITIA hypervisor identifies the breakpointed instruction (Ⓑ2), and the instruction (Ⓚ2) has a data race. The AITIA hypervisor notifies the user agent that the pair of instructions has a data race.

Generating a schedule. From the identified data races, the user agent creates a *reproduce schedule* according to the LIFS, a manifestation of an instruction sequence consisting of i) a system call to be started initially and ii) scheduling points. A *scheduling point* specifies an instruction address and interleaving order (e.g., Thread A is interleaved to Thread B at address 0x601020). The AITIA hypervisor enforces the interleaving orders of the data races as manifested in the reproduce schedule. After finishing a run of LIFS, the AITIA hypervisor reverts the memory contents of the reproducer.

4.4 Enforcing a Given Schedule

The reproducer and the diagnoser issue a hypercall to the AITIA hypervisor to enforce a scheduling manifestation. In response to the hypercall, the AITIA hypervisor holds the execution of the concurrent threads by installing breakpoints at their entry points. When both threads are ready, the AITIA hypervisor informs the user agent to start the concurrent threads. The AITIA hypervisor proceeds with only one thread specified in a schedule while the other thread is suspended.

Suspending a thread. The AITIA hypervisor modifies the program counter of a suspended thread to execute the trampoline code. The trampoline code is a busy loop, yielding its context to the kernel scheduler by calling the `cond_resched()` function. This method is required to make the suspended thread responsive to in-kernel notifications or hardware events. For example, TLB shootdown sends inter-process interrupts (IPIs) to all cores [6] expecting all cores to execute the TLB shootdown handler. By locating suspended threads to the trampoline, AITIA does not break the semantics of in-kernel communications such as IPI and RCU invocations.

Performing interleavings. To interleave a running thread A to a suspended thread B, the AITIA hypervisor installs breakpoints on the thread A's scheduling point (i.e., instruction address). Once a running thread hits the scheduling point, `VM_EXIT` occurs, and the AITIA hypervisor gains control of the virtual CPU executing the thread. The AITIA hypervisor stores the virtual CPU context information into AITIA hypervisor's memory and sets the program counter of thread A to the trampoline code. Then, the AITIA hypervisor resumes the suspended thread B by restoring the saved CPU context information from AITIA hypervisor's memory.

4.5 Testing Causality Between Data Race and Failure

After running the reproducing stage, AITIA receives a failure-causing instruction sequence and the data races found by the reproducer. Following Causality Analysis, AITIA assigns the data races that are to be flipped to the user agent in a diagnoser, and the user agent creates a *diagnosis schedule*. A diagnosis schedule has the same form as a reproduce schedule, which describes what thread starts initially and scheduling points to enforce the interleavings of the data races. To start a failure diagnosis, the user agent creates a hypercall and send it to the AITIA hypervisor. The AITIA hypervisor enforces the schedule manifested in the diagnosis schedule follwing §4.4. AITIA monitors which diagnosers successfully finish their tasks without causing a failure. Once AITIA detects alive diagnosers, AITIA checks which data race is flipped for these diagnosers and adds them to its root cause set. Because the diagnosing stage consists of independent tasks, AITIA launches multiple virtual machines to perform Causality Analysis in a parallel manner. As the final outcome, AITIA collects the complete root cause set from alive diagnosers and builds a causality chain to explain how the data races lead a kernel to the failure point.

4.6 Implementation

AITIA is implemented in various software layers. The manager, which orchestrates and manages multiple instances of a virtual machine, is implemented in 2,889 lines[4] of GO. The user agent is implemented in 730 lines of C. Reproducer and Diagnoser contains two parts: schedule generators and the AITIA hypervisor. The reproducer's schedule generator and the diagnoser's schedule generator are implemented with 2,765 lines and 1,051 lines of GO respectively. The AITIA hypervisor is implemented in KVM hypervisor and QEMU-3.0.0 with 10 and 3,004 lines of C respectively. The trampoline is implemented in 97 lines of a Linux kernel module.

Limitations. AITIA does not implement cases in which concurrency bugs occur in hardware IRQ contexts, such as concurrency bugs between a system call and a hardware interrupt handler. However, we believe that AITIA is able to diagnose such concurrent bugs if the AITIA hypervisor injects an IRQ through the VT-x mechanism as is done for system calls. We leave such cases as a future work.

5 Evaluation of AITIA

We evaluate AITIA in 22 real-world concurrency bugs found in the Linux kernel. To verify AITIA's correctness, we firstly test AITIA on concurrency failures from the CVE database [99]. Their root causes are well-analyzed. After that, we connect AITIA with an automated bug-finding tool to show how they work together to fulfill the three requirements: comprehensiveness, pattern-agnostic, and conciseness. We choose `Syzkaller` [26] because it is one of the most influential bug-finding systems.

All of our evaluations are performed on an Intel(R) Xeon(R) CPU E5-4655 v4 @ 2.50GHz (30MB cache) with 512GB of RAM. We run Ubuntu 18.04.03 LTS with Linux 4.15.18 64-bit. We use an instrumented kernel with KASAN [47, 89]. We use Linux coredump and ftrace to collect a failure information. For LIFS and Causality Analysis, we launch 32 VMs with the AITIA hypervisor.

5.1 Diagnosing Concurrency Failures in Linux Kernel

To show how AITIA works with real-world failures, We collect 10 concurrency failures from the CVE database since 2016 using keywords such that `Linux`, `race`, and `concurrent`. Many of these failures and their root causes are well-studied by the security community. For example, we can find the root cause and a detailed explanation of CVE-2017-2636 [70] in [5]. We use the cases to evaluate AITIA by comparing the result of AITIA with reported root causes.

[4]We use scc [1] and sloccount [104] to measure the LoC of GO and C respectively.

Bug ID	Subsystem	LIFS			Causality Analysis	
		Time(s)	# of sched.	Inter.	Time(s)	# of sched.
CVE-2019-11486 [74]	TTY	44.7	225	1	497.6	130
CVE-2019-6974 [75]	KVM	103.8	664	1	1183.8	688
CVE-2018-12232 [73]	SockFS	37.8	536	1	511.4	680
CVE-2017-15649 [69]	Packet socket	88	1052	2	337.9	257
CVE-2017-10661 [68]	Timer fd	32.8	99	1	336.1	266
CVE-2017-7533 [72]	Inotify	64.5	1056	1	1846.7	1578
CVE-2017-2671 [71]	IPV4	33.2	130	1	195.3	159
CVE-2017-2636 [70]	TTY	34.3	197	1	270	215
CVE-2016-10200 [66]	L2TP	32.8	112	1	184.9	159
CVE-2016-8655 [67]	Packet socket	47.8	213	1	184	135

Table 2. CVEs caused by a concurrency failure in Linux. *# of sched.* indicates the total number of interleavings executed by LIFS and Causality Analysis. *Inter.* indicates the interleaving count of LIFS.

Correctness. Table 2 shows the kernel concurrency bugs with which we test AITIA. All failures from the CVE database are caused by a race condition between two system calls. For 9 out of 10 failures, AITIA successfully builds their causality chain. As those failures are found in major parts of the Linux kernel, AITIA can identify the root cause of concurrency bugs in various kernel subsystems. We manually compare the developers' patch in the CVE reports with the results of AITIA. We confirm that AITIA identifies the root causes of the 9 failures; developers fixed the failures by eliminating the specific execution order(s) of data races represented in AITIA's causality chains. CVE-2016-10200 [66] is an exception that AITIA encounters an *ambiguity* case described in §3.4.

Performance. As shown in Table 2, AITIA reproduces a failure within up to 2 minutes and pinpoints the root cause within up to 30 minutes. AITIA completes the diagnosis with at most thousands interleavings (# of schedules performed). Causality Analysis takes a longer time compared to LIFS because most of interleavings executed by Causality Analysis cause a failure. When a failure occurs, AITIA has to reboot the virtual machine. We observe that the number of interleavings required to reproduce the failure is small. AITIA reproduces most of the failures with one or two interleaving(s) even for multi-variable concurrency failures; 6 out of 10 failures involve multiple variables.

5.2 Cooperation with an automated bug-finding system

Bug selection criteria. Because we do not know the root cause of reported failures in `Syzkaller`, we randomly select reported failures according to the following three criteria: **i)** A reported fix if exists, includes keywords of "race" or "lock". **ii)** A crash report contains multiple contexts (e.g., two system calls, a system call, and a kernel thread). **iii)** we exclude the cases related to hardware IRQ (it is our limitation). With selected failures, we enable tracing ftrace events when generating coredumps, so AITIA uses the coredumps as an input to extract kernel traces and failure information. To identify root causes of unfixed bugs, we run `Syzkaller` with enabling tracing ftrace events before starting fuzzing, and extract coredumps when failures are detected.

Table 3 shows concurrency failures that we use to evaluate AITIA. Six failures are collected from the open failure database of `Syzkaller` provided by Google, and the remaining six *unfixed* failures (bold entries in Table 3) are collected by running the modified `Syzkaller` in our testing environment. Among 12 failures, eight failures are caused by concurrent execution between two system calls and the rest of the failures are caused by system calls and a kernel background thread.

Comprehensiveness. We first confirm that a causality chain contains comprehensive information to fix each bug. The most intuitive way is to compare the root cause with the causality chain reported by AITIA. However, unlike failures used in §5.1, the root causes of all failures are not well-documented. Therefore, for bugs already fixed, we manually compare causality chains generated by AITIA with submitted kernel patches to fix the bugs. We verify that race conditions reported by the reported causality chains disappear if the patches are applied. For the six unfixed bugs, *two of them were reported by us and confirmed by Linux kernel developers.* One of them is still in the progress of confirmation (we are waiting for developers' response). AITIA also detects the root cause of the rest of the three bugs (#7, #8, #9), but their patches were submitted before we reported the root cause. So, we compare the patches with AITIA's results.

AITIA *successfully reproduces and diagnoses all the failures in Table 3*. For all failures, their fixes and our submitted fixes do not allow the specific execution order of data races in the causality chain; all fixes make the causality chain *cut* by removing interleaving(s) in the chain, preventing the failure. We confirm that AITIA does not encounter an *ambiguity* case for all failures in Table 3; whenever a data race surrounds a nested data race, developers regulate the nested one. In total, only for one among 22 failures (*i.e.*, CVE-2016-10200 as described in §5.1), developers fix the failure out of the causality chain.

Through the result, we re-emphasize that reporting all necessary information is crucial in kernel concurrency bugs considering they are usually caused by a combination of multiple data races. Otherwise, developers have to invest their effort into missing information with a part of the root cause.

Pattern-agnostic. We further categorize the concurrency bugs according to the number of racing variables involved. Six out of total 12 concurrency bugs have multi-variable races, and three bugs among them involve loosely-correlated racing variables, emphasizing that a practical diagnosis tool must consider such multi-variable concurrency bugs.

As AITIA is designed without relying on specific assumptions on interleaving patterns, AITIA can diagnose all failures regardless of the number of variables in which data

Bug ID	Subsystem	Bug type	Multi variables?	Least Interleaving First Search			Causality Analysis		
				Time (s)	# of schedules	Interleaving(s)	Time (s)	# of schedules	# of races in chain
#1 [90]	L2TP	Slab-out-of-bound access	Yes*	165.7	751	1	251.3	236	2
#2 [95]	Packet socket	Assertion violation	No	318	133	1	1152	471	4
#3 [92]	L2TP	Use-after-free access	Yes	65.8	178	1	1035.6	773	2
#4 [91]	KVM	Use-after-free access	Yes*	152.1	503	1	189.6	138	2
#5 [93]	RxRPC	Use-after-free access	No	45.7	2	1	930.4	405	1
#6 [94]	BPF	General protection fault	Yes	755	176	1	988	388	4
#7 [96]	Block device	Use-after-free access	No	872.7	231	1	1575	523	4
#8 [98]	CAN	Use-after-free access	Yes	2818.8	1044	2	3286	1469	5
#9 [97]	Seccomp	Memory leak	Yes*	1526.4	628	1	1452.6	848	2
#10 [45]	Software RAID	Assertion violation	No	70.8	101	1	2365.1	1032	4
#11 [52]	Floppy	Assertion violation	No	72.4	15	1	1692.9	627	2
#12 [55]	Bluetooth	Use-after-free access	No	740.1	272	1	2032	843	4

Table 3. Concurrency bugs to evaluate the AITIA's efficiency. For failures involving multi-variable races, asterisks (*) mean that the variables are loosely correlated. Bug IDs in bold were not fixed at the time of evaluation. Interleaving(s) presents the number of interleavings.

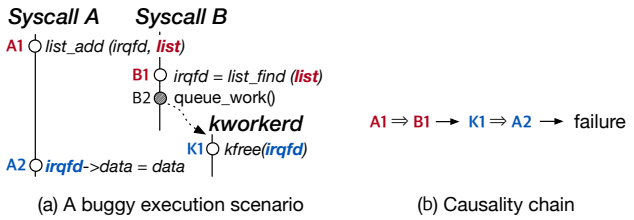


Figure 9. A buggy scenario of #4 and its causality chain

races occur or the relationship between memory objects. Unlike AITIA, previous approaches relying on specific assumptions have limited diagnosis capability. For example, cooperative bug localization approaches (*e.g.*, Snorlax [38] and Gist [37]) cannot diagnose the half of bugs because it assumes a small set of single-variable interleaving patterns (*i.e.*, single-variable atomicity violation and order violation), and three out of six multi-variable races do not hold the MUVI [58]'s assumption describing the relationship between variables.

Conciseness. To show how much a causality chain can reduce the developers' effort, we count the number of memory accessing instructions and total data races from failed executions of each failure. Then, we measure the number of data races in causality chains built from the failed executions.

On average, there are 9592.8 memory accessing instructions in failed executions, ranging from 189 to 20090. From the memory accessing instructions, we measure the number of individual data races which is 108.4 on average, ranging from 5 to 322. Therefore, simply detecting or reproducing concurrency failures [16, 25, 78] leaves a huge amount of effects to developers to identify the root cause. In contrast, Causality Analysis builds a causality chain with 3.0 data races in average, indicating that causality chains can significantly reduce debugging efforts. We also manually confirm that causality chains do not contain any benign data race.

Case study. We find that a concurrency bug (#4 [91]) shows an interesting case to explain why correctly fixing a kernel concurrency bug is difficult. Figure 9 shows a simplified view of the concurrency bug. In this example, two instructions (*i.e.*, `A1` and `A2`) in syscall A are a part of an initialization of a memory object, therefore, need to be executed atomically. In a buggy execution scenario, syscall A adds `irqfd` into `list` (`A1`). Then syscall B retrieves `irqfd` from `list` (`B1`) and invokes a kernel thread with it (`B2`). While syscall A still is in the middle of the initialization, the kernel thread frees `irqfd` (`K1`), causing a use-after-free failure (`A2`). This bug is particularly difficult to diagnose because *an outcome of an data race (`A1` ⇒ `B1`) affects another thread*. Even after developers find out this is a use-after-free bug and its direct reason (*i.e.*, the failed instruction (`A2`) and the instruction freed `irqfd` (`K1`)), they still need to figure out asynchronous events, the invocation of the kernel thread (`B2`) and the freeing instruction (`K1`), are caused by the race-steered control flow by another data race that occurred in a *different* thread (syscall B). This example shows how AITIA satisfies comprehensiveness and pattern-agnostic requirements and why it can help developers: 1) the causality chain contains all data races needed to fix the failure, and 2) AITIA can infer the causality of data races even across the thread boundary.

Performance. To measure how quickly AITIA can reproduce bugs and diagnose the root cause, we measure the elapsed times and the number of executed interleaving of LIFS and Causality Analysis respectively. On average, reproducing takes 633.6 seconds, and diagnosing takes 1412.5 seconds, suggesting AITIA is feasible to aid developers to diagnose concurrency bugs during the development phase. The reason for the short diagnosis time comes from two aspects: 1) many instructions do not access global memory objects, and 2) a small interleaving count is enough to reproduce a failure.

5.3 Comparison to Prior Approaches

In this section, we compare AITIA to various root cause diagnosis techniques in the perspective of comprehensiveness, pattern-agnostic, and conciseness requirements described in §2,

Kairux [117]. It defines the root cause of a failure as *an inflection point*, which is an instructions that resides in a failed run and deviates from all non-failed runs. As the definition of the root cause differs between AITIA and Kairux, AITIA has a notable difference from Kairux.

Since Kairux represents the root cause of all failures as a single instruction, more manual effort is required to fully understand a concurrency failure even after an inflection point is found. Thus, Kairux does not satisfy the *comprehensive* requirement. In Figure 9, an inflection point might be `K1` since in a failed run `A1` ⇒ `B1` ⇒ `K1` ⇒ `A2`, `K1` is an instruction that firstly deviates from non-failed runs (*e.g.*, `A1` ⇒ `B1` ⇒ `A2` ⇒ `K1`). Even with the inflection point, developers still need to investigate further across multiple threads. The inflection point does not explain that `K1` is executed because of the control flow steered by a data race `A1` ⇒ `B1`. As shown in Figure 9, understanding this race-steered control flow is also important to properly mitigate the failure. We believe a causality chain can aid developers in understanding how such complex concurrency failures occur.

Cooperative Bug Localization [7, 8, 34, 37, 38]. Almost all cooperative bug localization techniques consists of two steps. They first predefine a set of interleaving-related patterns that frequently cause a concurrency bug. And then, they find out a pattern that have the strongest statistical correlation to a concurrency failure. Because AITIA and cooperative bug localization have different methodologies, AITIA has significant differences from them in two aspects. First, cooperative bug localization techniques cannot reason concurrency bugs that their root causes do not fall into a predefined set of patterns. For example, a few state-of-the-art cooperative bug localization techniques [37, 38] cannot diagnose *all* multi-variable concurrency bugs in Table 3, indicating their methodology is limited in satisfying the *pattern-agnostic* requirement. Second, Cooperative bug localization techniques rely on a statistical method, and do not attest how a suspicious pattern drives a program into a failure during the runtime. Thus, they may point out a failure-irrelevent pattern as a root cause. In contrast, AITIA verifies the causality of each data race to a failure by observing how execution is changed according to an interleaving order of each data race.

MUVI [58]. Although MUVI is not specifically designed to diagnose concurrency failures, its idea can be applied to diagnose concurrency failures. MUVI reasons how multi-variable concurrency failures occur based on the assumption that multiple variables usually have an access correlation. In other words, multiple variables are usually correlated and need to be accessed together with their correlated peers in a consistent manner. MUVI statically analyzes a program to point out instructions that improperly access multiple variables that have access correlations. Although the assumption is held for many cases and MUVI successfully reasons many multi-variable concurrency failures, MUVI is limited in reasoning single-variable concurrency failures and loosely-correlated multi-variable concurrency failures, and thus its approach suffering from satisfying the *pattern-agnostic* requirement. In Table 3, only 3 out of 12 failures satisfy the assumption of MUVI and can be explained by the MUVI's approach. Whereas, 6 out of them are single-variable concurrency failures, and 3 are loosely-correlated multi-variable concurrency failures. AITIA can diagnose all 12 failures described in Table 3, indicating AITIA has the strong the diagnosis capability.

6 Related Work

Automatic root cause diagnosis. A huge number of prior works fall into the same category called *cooperative bug localization* [7, 8, 13, 34, 35, 37, 38, 43, 56, 108]. These approaches figure out one that has the strongest statistical correlation to a failure among predefined patterns such as well-known interleaving patterns (*e.g.*, atomicity violation and order violation [37, 38]) or interleaving-related predicates (*e.g.*, concurrent execution of the same function [34]). While they are useful for bugs that the root cause falls into the predefined set, they suffer from reasoning complex interleaving patterns which are hard to be specified as pre-defined predicates. On the other hand, Kairux [117] adopts a different strategy. It locates a *single* instruction as the root cause based on a simple and powerful intuition called *inflection point hypothesis*. In the context of kernel concurrency failures, the root cause of complex concurrency failures may not be a single instruction but multiple data races on multiple variables.

Failure reproduction. *Record & replay* [19, 27–29, 39, 41, 57, 62, 64, 77, 81, 82, 85–87, 103] promises to deterministically reproduce a failure with a cost of the high runtime overhead. To overcome the high runtime overhead, recent approaches reconstruct a *partial* failing execution through analyzing by-products such as logs [110, 116, 118, 119] and/or a coredump [111]. Another promising technique to reproduce a failure is to recover a program's state by reverting executed instructions [2–4, 10, 15, 16, 31, 65, 78, 101, 106, 112]. While they can reconstruct the exact memory state of a failed execution, figuring out the root cause is still an error-prone task, and developers often misunderstand the root cause [109]. In this sense, AITIA complements these approaches by drawing developers' attention to the root cause.

Concurrency bug detection. By adopting the idea of controlling an interleaving at runtime [11, 23], plentiful tools [12, 32, 33, 42, 107] are proposed to expose concurrency failures. While these tools successfully find many concurrency bugs in complex system softwares including the Linux kernel, they concentrate on mostly how to diversify interleaving to increase the chance of finding bugs. In contrast, AITIA controls the interleaving to identify the causal effect between data races and the failure. Another major research direction is to

detect data races [40, 44, 88, 114, 115]. Despite their usefulness, they cannot filter out benign data races [36]. They also cannot inspect the unintended execution order of critical sections; it is well known that the unintended order of critical sections may cause a concurrency failure [18]. There are also significant attempts to detect concurrency failures through static analysis [9, 17, 30, 58]. They have advantages of being able to find concurrency bugs without running a program, but they are fragile to false positives and hard to scale to complex system softwares.

7 Conclusion

We perform in-depth studies of Linux kernel concurrency bugs and drive three requirements: comprehensiveness, pattern-agnostic, and conciseness. This work proposes AITIA to satisfy the three requirements. AITIA fully automates the process of identifying the root cause of kernel concurrency bugs reported from existing bug-finding systems and analyzes the root cause as a form of a causality chain. We evaluate AITIA on 22 real-world concurrency bugs and successfully diagnose six unfixed bugs using AITIA.

8 Acknowledgment

We would like to thank anonymous reviewers and our shepherd Jia-Ju Bai for their insightful and valuable comments that helped us improve this paper. This work was supported in part by NRF-2020R1A2C2005479, IITP (2020-0-00209), IITP (2021-0-00871, Development of DRAM-Processing-In-Memory Chip for DNN Computing, 30%) and ERC (NRF-2018R1A5A1059921).

References

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

[2] T. Akgul and V. J. Mooney III. Instruction-level reverse execution for debugging. *ACM SIGSOFT Software Engineering Notes*, 28(1): 18–25, 2002.

[3] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 13(2):149–198, 2004.

[4] T. Akgul, V. J. Mooney, and S. Pande. A fast assembly level reverse execution method via dynamic slicing. In *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Scotland, UK, May 2004.

[5] Alexander Popov. CVE-2017-2636: exploit the race condition in the n_hdlc Linux kernel driver bypassing SMEP , 2020. https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html.

[6] N. Amit, A. Tai, and M. Wei. Don't shoot down tlb shootdowns! In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Heraklion, Crete, Greece, Apr. 2020.

[7] J. Arulraj, P.-C. Chang, G. Jin, and S. Lu. Production-run software failure diagnosis via hardware performance counters. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[8] J. Arulraj, G. Jin, and S. Lu. Leveraging the short-term memory of hardware to diagnose production-run software failures. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, UT, Mar. 2014.

[9] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu. Effective static analysis of concurrency {Use-After-Free} bugs in linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[10] B. Biswas and R. Mall. Reverse execution of programs. *ACM SIGPLAN Notices*, 34(4):61–69, 1999.

[11] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 167–178, 2010.

[12] 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)*, BOSTON, MA, Aug. 2020.

[13] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31th International Conference on Software Engineering (ICSE)*, Vancouver, Canada, May 2009.

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

[15] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, Austin, TX, May 2016.

[16] W. Cui, X. Ge, B. Kasikci, B. Niu, U. Sharma, R. Wang, and I. Yun. Rept: Reverse debugging of failures in deployed software. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, CARLSBAD, CA, Oct. 2018.

[17] P. Deligiannis, A. F. Donaldson, and Z. Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, NE, Nov. 2015.

[18] Dirty Cow. Dirty Cow, 2016. https://dirtycow.ninja/.

[19] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2008.

[20] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[21] 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.

[22] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages (POPL)*, Long Beach, California, Jan. 2005.

[23] 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.

[24] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997.

[25] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2021.

[26] Google. Syzkaller, 2020. https://github.com/google/syzkaller.

[27] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[28] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: Replication at the speed of multi-core. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[29] N. Honarmand and J. Torrellas. Replay debugging: Leveraging record and replay for program debugging. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Minneapolis, MN, June 2014.

[30] S. Hong and M. Kim. Effective pattern-driven concurrency bug detection for operating systems. *Journal of Systems and Software*, 86(2): 377–388, 2013.

[31] C. Hou, G. Vulov, D. Quinlan, D. Jefferson, R. Fujimoto, and R. Vuduc. A new method for program inversion. In *Proceedings of the 21st International Conference on Compiler Construction (CC)*, Tallinn, Estonia, Apr. 2001.

[32] 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.

[33] 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.

[34] G. Jin, A. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of the 21th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Reno/Tahoe, Nevada, Oct. 2010.

[35] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Long Beach, CA, Nov. 2005.

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

[37] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[38] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[39] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 155–166, 2010.

[40] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[41] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, (4): 471–482, 1987.

[42] 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.

[43] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[44] Linux. The kernel concurrency sanitizer (kcsan), 2019. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[45] Linux. md: fix a warning caused by a race between concurrent md_ioctl()s, 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c731b84b51bf7fe83448bea8f56a6d55006b0615.

[46] Linux. ftrace - Function Tracer, 2020. https://www.kernel.org/doc/html/latest/trace/ftrace.html.

[47] Linux. The Kernel Address Sanitizer (KASAN), 2020. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[48] Linux. kernel: add kcov code coverage, 2020. https://lwn.net/Articles/671640/.

[49] Linux. Explanation of the Linux-Kernel Memory Consistency Model, 2020. https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/explanation.txt.

[50] Linux. fix locking in bdev_del_partition (2020-09-01), 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=08fc1ab6d748ab1a690fd483f41e2938984ce353.

[51] Linux. Concurrency Managed Workqueue (cmwq), 2020. https://www.kernel.org/doc/html/latest/core-api/workqueue.html.

[52] Linux. WARNING in schedule_bh, 2021. https://lore.kernel.org/lkml/YbbKf6fU7y3GGZum@archdragon/.

[53] Linux. add sysfs_lock to synchronize sysfs code paths (2021-03-29), 2021. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4e9c93af7279b059faf5bb1897ee90512b258a12.

[54] Linux. fix uaf for rx_kref of j1939_priv (2021-09-26), 2021. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d9d52a3ebd284882f5562c88e55991add5d01586.

[55] Linux. Bluetooth: fix dangling sco_conn and use-after-free in sco_sock_timeout, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=7aa1e7d15f8a5b65f67bacb100d8fc033b21efa2.

[56] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[57] H. Liu, S. Silvestro, W. Wang, C. Tian, and T. Liu. ireplayer: In-situ and identical record-and-replay for multithreaded applications. In *Proceedings of the 2018 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Philadelphia, PA, June 2018.

[58] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, Oct. 2007.

[59] S. Lu, S. Park, and Y. Zhou. Detecting concurrency bugs from the perspectives of synchronization intentions. volume 23, pages 1060–1072. IEEE, 2011.

[60] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. volume 50, pages 586–595. ACM New York, NY, USA, 2015.

[61] N. Machado, B. Lucia, and L. Rodrigues. Production-guided concurrency debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[62] A. J. Mashtizadeh, T. Garfinkel, D. Terei, D. Mazieres, and M. Rosenblum. Towards practical default-on multi-core record/replay. *ACM SIGPLAN Notices*, 52(4):693–708, 2017.

[63] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[64] T. Merrifield, S. Roghanchi, J. Devietti, and J. Eriksson. Lazy determinism for faster deterministic multithreading. In *Proceedings of the 24nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[65] Microsoft Corporation. Time travel debugging, 2020. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview.

[66] MITRE. CVE-2016-10200, 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10200.

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

[68] MITRE. CVE-2017-10661, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10661.

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

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

[71] MITRE. CVE-2017-2671, 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2671.

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

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

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

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

[76] MITRE. CVE-2020-8832, 2020. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8832.

[77] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Beijing, China, June 2008.

[78] Mozilla Corporation. Mozilla rr, 2020. https://rr-project.org.

[79] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, 42(6):446–455, 2007.

[80] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[81] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, Mar. 2009.

[82] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192, 2009.

[83] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, May 2010.

[84] S. Park, R. Vuduc, and M. J. Harrold. Unicorn: a unified approach for localizing non-deadlock concurrency bugs. *Software Testing, Verification and Reliability*, 25(3):167–190, 2015.

[85] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. Coreracer: a practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Porto Alegre, Brazil, Dec. 2011.

[86] S. Ren, L. Tan, C. Li, Z. Xiao, and W. Song. Samsara: Efficient deterministic replay in multiprocessor environments with hardware virtualization extensions. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, Denver, CO, June 2016.

[87] M. Ronsse and K. De Bosschere. Recplay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)*, 17(2):133–152, 1999.

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

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

[90] Syzkaller. KASAN: use-after-free Read in pppol2tp_connect, 2018. https://syzkaller.appspot.com/bug?id=63fac1a987fb08f242a98a35578b3eb14c7a9a93.

[91] Syzkaller. KASAN: use-after-free Write in irq_bypass_register_consumer, 2018. https://syzkaller.appspot.com/bug?id=45591ae3053c59fb50169401fb61cb596735f9d1.

[92] Syzkaller. KASAN: use-after-free Read in pppol2tp_connect, 2018. https://syzkaller.appspot.com/bug?id=ddc83e209f712ce63078e146f7c0fe63e1edbc2f.

[93] Syzkaller. KASAN: use-after-free Read in rxrpc_queue_local, 2019. https://syzkaller.appspot.com/bug?id=a0e85017c37a10447d1729523e7ba4fda8ef3f4a.

[94] Syzkaller. general protection fault in dev_map_hash_update_elem, 2019. https://syzkaller.appspot.com/bug?id=051976ec3138aa9e5ffbc0c1afd02e3f6bea2c68.

[95] Syzkaller. general protection fault in packet_lookup_frame, 2019. https://syzkaller.appspot.com/bug?id=def39b541a4d5fdd258ec710d2a159010c8601f3.

[96] Syzkaller. KASAN: use-after-free Read in delete_partition, 2020. https://syzkaller.appspot.com/bug?id=a09edb22abb4af520fef12024cc9f860c4307c8f.

[97] Syzkaller. memory leak in do_seccomp, 2020. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a566a9012acd7c9a4be7e30dc7acb7a811ec2260.

[98] Syzkaller. WARNING: refcount bug in j1939_netdev_start, 2021. https://syzkaller.appspot.com/bug?id=573a8a7b9071e7549e4f6dc77599564187e677fc.

[99] The MITRE Corporation. CVE - The MITRE Corporation, 2020. https://cve.mitre.org/.

[100] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 131–144, 2007.

[101] Undo. UndoDB: The interactive reverse debugger for C/C++ on Linux and Android., 2020. https://undo.io.

[102] J. D. Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[103] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. *ACM Transactions on Computer Systems (TOCS)*, 30(1):3, 2012.

[104] D. A. Wheeler. Sloccount, 2020. https://dwheeler.com/sloccount/.

[105] P. F. Wilson. *Root cause analysis: A tool for total quality management*. ASQ Quality Press, 1993.

[106] J. Xu, D. Mu, X. Xing, P. Liu, P. Chen, and B. Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

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

[108] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[109] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 17th European Software Engineering Conference (ESEC) / 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Szeged, Hungary, Aug. 2011.

[110] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, Mar. 2010.

[111] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, Paris, France, Apr. 2010.

[112] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated debugging for arbitrarily long executions. In *14th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XIV)*, Santa Ana Pueblo, NM, May 2014.

[113] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation*, pages 250–259, 2015.

[114] 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.

[115] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. *ACM SIGPLAN Notices*, 52(4):149–162, 2017.

[116] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[117] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.

[118] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, 2014.

[119] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.