A Comprehensive Study of Concurrency Bugs in the Linux Kernel

Sishuai Gong
University of North Carolina at Chapel Hill
Chapel Hill, NC, USA

Chih-En Lin
Purdue University
West Lafayette, IN, USA

Kevin Wu
Purdue University
West Lafayette, IN, USA

Edwin Lu
Purdue University
West Lafayette, IN, USA

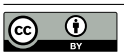
Pedro Fonseca
Purdue University
West Lafayette, IN, USA

Abstract

Concurrency bugs arise from unexpected orderings of concurrent instructions and are notoriously difficult to detect and diagnose due to their non-deterministic nature. Prior studies of concurrency bugs in user-space applications have identified common bug characteristics and informed the design of effective detection, diagnosis, and repair techniques. In contrast, far less is known about concurrency bugs in operating system kernels, where low-level systems programming and complex execution patterns introduce fundamentally different concurrent execution behaviors.

This paper presents the first comprehensive study of concurrency bugs in the Linux kernel. We analyze 200 real-world kernel concurrency bugs and systematically characterize their manifestation conditions, root causes, discovery processes, and repair characteristics. Our study reveals that kernel concurrency bugs differ substantially from their user-space counterparts. For instance, 27.5% of kernel concurrency bugs manifest only when specific interrupt events occur, yet such bugs are often overlooked by existing bug discovery approaches. Moreover, nearly half of the bugs occur in kernel drivers, and 69.3% of them stem from concurrent device operations on control paths such as device registration and deregistration, highlighting driver control logic as a critical target for concurrency bug discovery. Overall, our findings expose key limitations in current approaches to kernel concurrency bug detection and analysis. By shedding light on these challenges, this work paves the way for new operating system designs, testing tools, and verification efforts that will make modern operating systems more reliable and secure.

ACM Reference Format:
Sishuai Gong, Chih-En Lin, Kevin Wu, Edwin Lu, and Pedro Fonseca. 2026. A Comprehensive Study of Concurrency Bugs in the Linux Kernel . In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787790



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787790

1 Introduction

Operating system kernels rely on fine-grained concurrency to achieve high scalability and performance. Yet exploiting concurrency is notoriously error-prone [60], often leading to subtle, non-deterministic bugs that compromise system reliability and security [56, 59]. Kernel concurrency stems from multiple, inherently distinct sources—including system calls from user-space applications, asynchronous hardware events, and background kernel threads—and further involves low-level synchronization primitives, shared states, and long, highly-optimized code paths. Together, these factors make reasoning about concurrent correctness extremely difficult, leaving modern kernels susceptible to concurrency bugs.

Despite the kernel's foundational role in system reliability, there is a surprising lack of in-depth studies dedicated specifically to kernel concurrency bugs. Prior work [40] on user-space concurrency bugs has yielded valuable insights into bug patterns, manifestation conditions, and repair strategies—insights that have shaped the design of testing tools, programming models, and verification techniques. Unfortunately, it is unclear whether or to what extent these user-space findings generalize to kernels. This knowledge gap is particularly concerning for several reasons. First, kernel correctness is essential for applications that depend on it [20, 36]. Second, kernel concurrency is often implemented with finer granularity and more aggressive optimizations than application concurrency [15, 42, 43], further increasing the risk of subtle concurrency bugs. Third, kernels differ significantly from applications in structure and execution behavior: they interact directly with hardware, rely on low-level code, and use specialized synchronization primitives such as Read-Copy Update (RCU) [23, 42] and per-CPU variables. These differences raise a critical question: do kernel concurrency bugs follow the same patterns observed in user-space, or do they exhibit distinct characteristics, which could impact how these bugs are prevented, diagnosed, and fixed in practice?

This paper presents the first comprehensive study on kernel concurrency bugs. Our study analyzes 200 concurrency bugs drawn from 4.5 years of Linux kernel development history. We examine their manifestation conditions, root causes, discovery and repairs to shed light on the unique characteristics of kernel concurrency. In particular, our analysis centers on two key questions:

1. What are the similarities and differences between kernel and user-space concurrency bugs? One would expect the characteristics of concurrency bugs in kernels and applications to have important differences. On the one hand, identifying and characterizing such differences is crucial for understanding the kernel-specific challenges in correctly using kernel concurrency. On the other

hand, identifying the similarities between kernel and user-space concurrency bugs can underscore the common, cross-stack pitfalls of concurrent programming.

Our findings reveal that:

- 27.5% of kernel concurrency bugs only manifest under specific interrupt events, which are often underexplored by existing testing tools focused on thread interleavings.
- 44.0% of bugs occur in kernel drivers, with 69.3% of those involving control paths, such as device initialization, shutdown, suspend, and reset.
- 45.0% of bugs cannot manifest in typical virtual machines used by Syzbot [54]—the leading continuous kernel testing platform—due to hardware dependencies.
- 90.7% of non-deadlock bugs stem from atomicity or order violations, a high proportion, though not directly comparable to user-space studies. The remaining bugs reflect kernel-specific concurrency challenges, such as hardware-related kernel behaviors.
- 29.5% of bugs involve incorrect synchronization use, such as choosing the weaker synchronization primitives or missing initializations (e.g., `mutex_init()`).
- 98.5% of bugs can be triggered with just one or two threads, suggesting that it is reasonable for concurrency bug discovery techniques to prioritize interleavings involving at most two threads, rather than exhaustively exploring high thread counts.

2. How can these findings inform future tools for kernel concurrency bug detection, diagnosis, and repair? Due to the lack of comprehensive studies on kernel concurrency bugs, many prior kernel concurrency bug tools are built on assumptions rooted in user-space concurrency bug studies [30, 34, 57]. They often focus exclusively on thread-based interleavings, overlook interrupt-driven execution in the kernel, and exclude low-level or platform-specific code such as bootloaders [21] or inline assembly to reduce the analysis complexity. While these design choices are motivated by practical concerns, it is unclear whether these assumptions faithfully capture the realities of kernel concurrency bugs.

Our study sheds light on these questions by revealing where concurrency bugs actually occur and what is needed to address them. Furthermore, our findings point to new opportunities for improving tool designs, such as:

- **Interrupt execution analysis is vital.** 27.5% of bugs manifest only when specific interrupt events occur. Bug detection tools should therefore include interrupt-driven concurrency, not just concurrency induced by system calls.
- **Static and emulation-based methods are necessary.** 45.0% of bugs depend on hardware unavailable in common virtual-machine-based environments [54]. Tools that rely solely on dynamic execution will miss these cases; static analysis or specific hardware emulation is required to detect such bugs.
- **Driver-focused analysis pays off.** 44.0% of the bugs reside in kernel device drivers, with 69.3% of those involving device control operations. Constraining the analysis scope to device drivers, especially their control paths, can yield high returns with reduced complexity.
- **Assembly and bootloader exclusions appear justified.** Despite concerns about hard-to-analyze low-level code, we do not encounter any concurrency bugs involving inline assembly or bootloader logic. This observation suggests that, in practice, excluding such code—when it substantially reduces analysis complexity—may represent a reasonable trade-off for verification and testing tools targeting concurrency bugs.
- **Atomicity and order violations are prevalent.** 90.7% of non-deadlock bugs stem from atomicity or order violations. This indicates that analysis techniques targeting these root causes are broadly applicable to kernel code.
- **Systematic synchronization guidance is needed.** 29.5% of bugs involve misuse of synchronization primitives, often due to misunderstanding valid concurrency contexts. Tools that infer and explain potential interleavings can help developers choose correct synchronization mechanisms.
- **Automation remains underutilized.** 65.9% of bugs are found manually by kernel developers, highlighting a significant opportunity for automated tools that provide actionable diagnostics and reproducible test cases.

Our key findings and their implications are summarized in Table 1.

Contributions This paper presents the first comprehensive study of concurrency bugs in the Linux kernel. Our study focuses on analyzing the kernel-specific properties of kernel concurrency bugs, and the similarities and differences that kernel concurrency bugs have over user-space concurrency bugs. We manually examine 200 kernel concurrency bugs selected from a 4.5-year period of kernel commits. Although prior work has studied concurrency bugs in applications and others have studied kernel bugs [22, 47], few studies examine *kernel concurrency bugs*. The few that do cover kernel concurrency bugs focus only on narrow classes of concurrency bugs in the Linux kernel, such as use-after-free bugs [18] and data races [50, 52]. §8 further discusses the related work.

To foster future research, we publicly release our dataset[1].

2 Methodology

Our study follows a two-stage methodology. In the first stage, we curate a dataset of Linux kernel concurrency bugs through a semi-manual selection process. In the second stage, we conduct an in-depth manual analysis of each bug to extract its key characteristics. We describe these stages in detail below.

2.1 Bug selection

Drawing on common practices from prior studies of user-space concurrency bugs [10, 18, 27, 40, 53], we adopt three guiding principles when selecting bugs for the study. Specifically, we (1) focus on the main kernel development branch (i.e., mainline); (2) randomly sample bug-fixing commits; and (3) apply keyword-based filtering to obtain an initial set of potential concurrency bugs, followed by manual validation to retain only true bugs.

[1]Dataset: https://github.com/rssys/kernel-concurrency-bug-study

Findings on bug manifestations (§3)	**Implications**
Most kernel concurrency bugs (71.0%) are caused by thread-based interleavings. However, 27.5% of bugs only manifest when specific interrupts occur.	Detection approaches focused on concurrent thread interleavings are effective for most bugs, but comprehensive detection must also account for interrupt-driven concurrency, which remains underexplored.
44.0% of concurrency bugs reside in driver code. Among them, 69.3% result from concurrent operations on control paths (e.g., `init` vs. `reset`).	Constraining analysis to logically opposing operations offers a tractable trade-off between complexity and effectiveness.
45.0% of concurrency bugs cannot manifest in the virtual machines used by Syzbot, as they depend on hardware or configurations not supported in typical testing environments.	Existing VM-based dynamic testing platforms miss a significant portion of concurrency bugs. Static analysis or hardware-aware modeling is necessary to expose these hardware-dependent bugs.

Findings on bug patterns (§4)	**Implications**
Deadlocks account for 24.5% of concurrency bugs. 32.7% of them are single-thread deadlocks, mostly caused by unexpected indirect control paths or interrupt handlers acquiring locks already held by the thread.	Effective deadlock detection requires modeling indirect control flow that spans thread execution and interrupt contexts, particularly under asynchronous hardware events.
90.7% of non-deadlock bugs result from atomicity or order violations, a high proportion that partially overlaps with patterns seen in user-space, but also reflects kernel-specific variation.	Techniques for addressing atomicity and order violations are broadly applicable, though kernel-specific concurrency behaviors require additional consideration.
Synchronization misuse accounts for 29.5% of bugs, including incorrect primitive selection or missing required operations (e.g., `mutex_init()`).	Choosing correct synchronization in the kernel is non-trivial; inferring possible concurrent contexts can help guide synchronization design.
98.5% of concurrency bugs can be reproduced with two or fewer threads.	Restricting analysis to two threads captures most kernel concurrency bugs, while still requiring consideration of interrupt handlers.

Findings on bug discovery and repair (§5, §6)	**Implications**
Among 85 concurrency bugs for which we are able to determine how they are found, automated tools contribute 34.1% of discoveries while most concurrency bugs (65.9%) are found manually.	There is considerable opportunity to improve the automated detection of kernel concurrency bugs. Future tools should aim to support diverse concurrency inputs, provide actionable diagnostics, and reliably reproduce failures.
Kernel concurrency bugs exist for significantly longer periods (1,258 vs. 765 days on average) and require more invasive patches (1.9× more insertions) than typical kernel bugs.	Kernel concurrency bugs are more time-consuming and labor-intensive to address. Automated repair techniques targeting concurrency issues could substantially reduce developer burden and patch time.

Table 1: Our findings on kernel concurrency bugs and the implications for bug discovery, diagnosis and repair.

1. Bug source. We use the commit history of the mainline Linux kernel as the data source, as it provides the most complete and authoritative record of confirmed and fixed bugs. In contrast, the CVE database [3] cannot serve as a representative source for kernel concurrency bugs: CVE reporting is known to be incomplete and biased toward bugs that are easier to exploit [2], such as memory errors. For example, in 2022 alone, more than 20K bug-fix commits were made to the Linux kernel, whereas only 281 bugs were documented as CVEs [3]. This discrepancy largely stems from the voluntary nature of CVE registration and the kernel community's historical reluctance to file CVEs [8, 9]. Similar limitations apply to other sources, such as Syzbot [54], a public continuous kernel testing project, which reports only bugs found by itself. Although CVEs and Syzbot could, in principle, serve as supplementary sources, integrating them in a statically sound manner would require substantial additional validation and analysis. Therefore, we use the mainline bug-fix commits as the sole source, ensuring broad coverage over a 4.5-year period (from January 1, 2019, to June 1, 2023).

2. Keyword-based filtering. The Linux mainline history contains a large number of commits—376,288 commits during our target period. To focus the analysis on commits that are likely related to concurrency bugs, we adopt a keyword-based filtering strategy that has been widely used in prior bug studies [17, 27, 40]. Specifically, we perform a case-insensitive search over commit messages using a set of concurrency-related keywords to obtain an initial candidate set. The keyword list includes *'deadlock', 'livelock', 'lock', 'mutex', 'rcu', 'race', 'concurren', 'interrupt', 'contention', 'preempt', 'atomic', 'locking', and 'synchronization'*. This filtering step yields 63,022 candidate commits, which are then randomly ordered and forwarded to the next stage for manual inspection.

To assess the effectiveness of the filtering approach, we conduct a validation experiment on an independent sample of commits. We randomly select 100 commits from the Linux mainline history and manually inspect each to determine whether it corresponds to a concurrency bug and, if so, whether the bug is well documented. Among these, 5 commits are confirmed to be true concurrency bug fixes. We then apply the same keyword-based filter to this 100-commit sample. The filter selects 17 commits, substantially reducing the search space. Of these 17 commits, 3 correspond to true concurrency bug fixes. The remaining 2 concurrency bug fixes are not captured by the filter because their commit messages lack explicit concurrency-related terminology; our manual validation identifies them as concurrency-related primarily through the code changes, not the commit message.

The validation confirms that filtering effectively narrows the analysis to well-documented and clearly-described concurrency bugs, while potentially missing bugs that are only implicitly documented. This limitation is shared by prior studies using similar approaches [17, 27, 40], and represents a practical trade-off necessary to make a large-scale manual study of kernel concurrency bugs feasible given the complexity of analyzing concurrency bugs [40].

3. Manual validation. The keyword-based filter produces a set of commits that are likely to correspond to concurrency bug fixes. We manually examine these commits to confirm true concurrency bug fixes. Starting from the randomly ordered list, two independent reviewers analyze commits through a two-phase review process. In the first phase, each reviewer independently inspects the commit message, code changes, and any linked bug reports or mailing list discussions. A bug is classified as a concurrency bug if it manifests only under specific interleavings (i.e., specific execution orders that can occur when multiple execution contexts run concurrently). In the second phase, reviewers resolve any disagreements through joint discussion to reach a consensus.

Through the two-phase process, we examine 1,633 commits and confirm 200 kernel concurrency bugs. Confirming kernel concurrency bugs at this scale is non-trivial: each kernel commit often requires several hours of careful reasoning about low-level kernel code, interleaved execution, and patch semantics. Beyond confirmation, each bug in our dataset is analyzed in depth (§2.2) to characterize its manifestation, root cause, discovery method, and fix. To our knowledge, this dataset is the first and largest of its kind: a collection of kernel concurrency bugs that are deeply characterized through consistent and rigorous manual analysis. Our dataset also is comparable to, and often larger than, those used in prior studies that conduct similarly detailed analyses of concurrency bugs in user-space applications [17, 27, 40, 53, 55].

2.2 Bug analysis

In-depth analysis of kernel concurrency bugs is inherently difficult and time-consuming. Nevertheless, every bug in our dataset is cross-reviewed by two independent reviewers following the two-phase review process described earlier—independent inspection followed by joint discussion to resolve disagreements—to ensure robustness. Building on this labor-intensive effort, we systematically analyze each bug along several key dimensions, including manifestation conditions, bug patterns and root causes, discovery methods, and repair. Our goal is to uncover actionable insights to guide future research on reliable and secure kernel concurrency.

Concurrency bug manifestation conditions. Kernel concurrency bugs can arise under complex and diverse execution contexts. While user-space programs may handle asynchronous events such as signals, interrupt handling in the kernel is far more complex: hardware interrupts can preempt almost any kernel instruction, execute on any CPU, and manipulate shared kernel state. Consequently, some kernel concurrency bugs manifest only when specific interrupts occur. Beyond interrupts, kernel concurrency bugs may also depend on particular hardware configurations. Large portions of the kernel—especially device drivers [51]—are hardware-specific and execute only in the presence of the corresponding devices. As a result, bugs in such code may never manifest in virtualized or testbed environments that lack the required hardware.

Despite the widespread belief [37, 43] that some kernel concurrency bugs depend on hardware interrupts or configurations, these manifestation conditions have not been systematically quantified. We present the first quantitative study of how kernel concurrency bugs manifest in practice. Specifically, we study (1) how many bugs depend on interrupts, (2) how many can manifest in virtual machine environments commonly used by existing dynamic testing tools, and (3) how these bugs are distributed across kernel subsystems, with particular emphasis on bugs located in device drivers.

Concurrency bug patterns and root causes. We analyze the root causes of kernel concurrency bugs and examine whether well-established patterns observed in user-space concurrency bugs still hold on the kernel. First, we classify each bug by its underlying cause and compare the distribution of bug patterns with those reported in prior studies of user-space concurrency bugs [40]. Second, we conduct a kernel-specific investigation into how misuse of synchronization primitives (e.g., spinlocks, mutexes, rcu) leads to concurrency bugs. Because the kernel provides a broader and more performance-sensitive set of synchronization mechanisms than user-space programs, understanding their misuse is essential for improving the correctness of concurrent kernel code and for guiding the design of future detection tools.

Concurrency bug discovery and repair. We next study how kernel concurrency bugs are discovered and repaired. For each bug, we identify the discovery method—such as manual inspection or automated tools including static analyzers and dynamic testing tools—to assess the effectiveness of tool support for finding kernel concurrency bugs. Then, we analyze the bug repair process by measuring (1) the time between bug introduction and fix, and (2) the patch complexity, including the size of code changes and the number of modified files or functions. We compare these metrics against those of general kernel bugs to determine whether concurrency bugs pose greater challenges during maintenance and repair.

2.3 Threats to validity

Concurrency bugs are particularly difficult to reproduce, and many require specific hardware or execution environments. As a result, our analysis is based on commit messages, code changes, kernel source code, mailing list discussions, and related artifacts, rather than direct bug reproduction. While this approach is necessary given the scale of the study and the practical barriers to reproducing kernel concurrency bugs, it introduces some uncertainty in characterizing certain bugs. We explicitly note such cases in our results when definitive analysis is not possible.

3 Concurrency bug manifestation conditions

This section examines three key factors that influence the manifestation of kernel concurrency bugs: (1) whether a bug manifests only when certain interrupt events occur; (2) whether a bug depends on the specific hardware, particularly whether it can occur in the virtual machine environments commonly used by existing testing tools; and (3) whether a bug is tied to specific kernel subsystems.

Category	Bugs
Interrupt-based	55 (27.5%)
Thread-based	142 (71.0%)
Unknown	3 (1.5%)
Total	200

Table 2: Distribution of concurrency bugs by whether bug manifestation requires at least one interrupt event.

3.1 Interrupt-based concurrency bugs

Concurrency in the kernel arises from two main sources: system calls and hardware interrupts. System calls are invoked by user-space applications and executed by kernel threads through well-defined entry points. In contrast, hardware interrupts are asynchronous events generated by devices; interrupt handlers may preempt running threads at almost any point, introducing additional interleavings with thread execution. Most prior work on dynamic kernel concurrency testing [28, 30–32, 35, 54] focuses on concurrency triggered by system calls, leaving interrupt-driven concurrency largely unexplored. As a result, bugs whose manifestation fundamentally depends on interrupt execution may be systematically overlooked, motivating a closer examination of interrupt-driven concurrency in the kernel.

We determine whether a bug's manifestation requires interrupts by inspecting the bug-fix commit and the affected code paths. A bug is classified as interrupt-based only if at least one involved code region is reachable exclusively through interrupt handling (e.g., an interrupt handler or deferred interrupt context). If the same buggy code path can also be triggered through system calls, interrupt involvement is not considered essential, to avoid over-attributing bugs to interrupts. We label a bug as unknown when evidence is insufficient to determine its execution context.

The agreement between two independent reviewers (§2.2) is strong. When including unknown cases, Cohen's K [41] is 0.79 (95% confidence interval [0.68, 0.86]), indicating moderate agreement; when excluding unknown cases, K increases to 0.92 (95% confidence interval [0.83, 0.97]), indicating nearly perfect agreement. Each reviewer's classifications also show high concordance with the final consensus (accuracy 0.92 and 0.98, respectively), suggesting that disagreements are rare and that the analysis is robust.

Table 2 summarizes the results. A nontrivial fraction of kernel concurrency bugs (27.5%) are interrupt-based, highlighting the critical role of interrupt-driven concurrency in the kernel. Figure 1 shows an example involving two thread contexts and an interrupt handler. In this bug, thread A holds `iommu->block`, while thread B holds `domain->lock`. Because interrupts are not disabled by thread A, an interrupt may preempt execution at an inopportune moment and attempt to acquire `domain->lock`, resulting in a deadlock.

Interrupt-based bugs are challenging to find because interrupt handlers are inherently asynchronous and often execute under constrained synchronization rules. These properties substantially expand the space of feasible interleavings and complicate systematic exploration, making such bugs particularly difficult to trigger and reproduce in practice. Existing tools such as Syzkaller predominantly focus on concurrency arising from system calls and leave interrupt paths largely unexplored. Our findings therefore expose a critical gap in current testing practice and motivate the development

```
1   /*  Thread A on CPU0 */  10 /*  Thread B on CPU1   */
2   /*intel_svm_bind_mm()*/  11 /*intel_svm_unbind_mm()*/
3 - lock(&iommu->lock);       12
4 + lock_irqsave(...);        13
5                             14 local_irq_disable();
6                             15 lock(&domain->lock);
7                             16 lock(&iommu->lock);
8   <Interrupt>               17
9   lock(&domain->lock);      18
    *** Deadlock ***
```

Bug-inducing interleaving: 3 -> 14, 15, 16 -> 8

Figure 1: An interrupt-based deadlock in the VT-d driver [19], which manifests only when thread execution interleaves with an interrupt handler.

of tools that more directly exercise interrupt-driven concurrency. Interestingly, we do not observe any concurrency bugs triggered solely by multiple interrupts. This likely reflects strict serialization rules enforced by the kernel, which prevent simultaneous execution of interrupt handlers [25].

Finally, we observe that most kernel concurrency bugs (71.0%) are thread-based. These bugs commonly involve concurrent system calls accessing shared kernel state without adequate synchronization, underscoring the continued importance of implementing thread-safe system call paths. This result also provides quantitative support for the effectiveness of existing testing tools that primarily explore interleavings of concurrent system calls.

> **Finding:** Although thread-based concurrency accounts for the majority of kernel concurrency bugs (71.0%), a nontrivial 27.5% manifest only under interrupt-driven execution.
>
> **Implication:** The presence of interrupt-based concurrency bugs indicates that tools centered on thread interleavings are necessary but insufficient, motivating testing approaches that explicitly consider concurrency involving interrupts.

3.2 Hardware-dependent concurrency bugs

In addition to the emphasis on system-call–driven concurrency, existing dynamic kernel testing approaches are constrained by their reliance on virtualized execution environments. Modern kernel testing platforms such as Syzbot [54] typically run the kernel inside virtual machines (VMs) configured with common CPU architectures (e.g., `x86-64`, `ARM`, and `RISC-V`) and a limited set of standard virtual devices. This design enables scalable and automated bug discovery by simplifying kernel deployment and execution, but it also restricts testing coverage to code paths that can execute under these virtualized hardware configurations. As a result, substantial portions of the kernel remain untested [48, 51], particularly code that depends on specific physical devices or less common architectural features. We therefore quantify how often kernel concurrency bugs require hardware configurations unavailable in typical VM-based testing environments such as Syzbot, by comparing each bug's hardware requirements against Syzbot's supported architectures and devices.

For each bug, we determine whether it can manifest in a typical Syzbot environment. Our analysis proceeds as follows. First, we review Syzbot's supported architectures and virtual devices. Second, we examine the fix commit and the relevant kernel code to identify the hardware involved in the buggy execution, such as specific device drivers or architecture-dependent behavior. We classify a bug

Category	Bugs
VM-compatible	110 (55.0%)
Hardware-dependent	90 (45.0%)
Total	200

Table 3: Distribution of concurrency bugs by whether they can manifest in the virtual machine environments used by Syzbot.

as *hardware-dependent* if its manifestation requires physical devices, peripherals, or architectural behaviors that are not supported or exercised in Syzbot's VM configurations. Bugs that can manifest under standard VM environments are classified as *VM-compatible*.

Two reviewers perform the classification of each bug's hardware requirements. The inter-reviewer agreement is moderate, with a Cohen's K [41] of 0.63 (95% confidence interval [0.51, 0.73]) when including unknown cases and 0.65 (95% confidence interval [0.53, 0.75]) when excluding them. Each reviewer's results align well with the final consensus (accuracy 0.92 and 0.90, respectively), suggesting that the overall classification is consistent and reliable.

Table 3 summarizes the results. We find that 45.0% of concurrency bugs in our dataset are hardware-dependent and cannot manifest in typical VM-based testing environments. These bugs include issues in device drivers for specific peripherals (e.g., network cards, GPUs, and USB devices), as well as bugs that arise only on particular CPU architectures. Figure 2 illustrates a bug specific to the `powerpc` architecture. Due to weak memory ordering [33] on `powerpc`, both CPUs can observe stale values and incorrectly conclude that the reset is not necessary, leading to a missed watchdog reset.

These findings reveal a fundamental limitation of VM-based dynamic testing: Virtualized devices often lack the fidelity needed to reproduce real-world kernel concurrency behaviors, leaving important code regions—particularly in `drivers/` and `arch/`—largely unexplored. While static analysis tools are not constrained by runtime environments and could, in principle, reason about hardware-dependent code, they often struggle with low-level semantics (e.g., memory models) and pervasive indirect control flow [11, 13, 14, 26, 38, 46]. Among the 85 bugs for which discovery methods can be identified, none are discovered by static analysis tools (§5).

Finding: 45.0% of kernel concurrency bugs are hardware-dependent and cannot manifest in typical VM-based testing environments.

Implication: Future research should explore bug finding techniques that can effectively analyze hardware-specific code under concurrency.

3.3 Subsystem-specific concurrency bugs

Finally, we analyze how kernel concurrency bugs are distributed across subsystems to identify components where such bugs most frequently arise and where bug finding efforts may be most effective. To determine each bug's location, we manually locate the culprit source files and classify each bug according to the kernel directory structure, following conventions used in prior kernel studies [45]. The mapping between components and top-level directories is as follows: { Drivers: `drivers/`, `sound/`, File systems: `fs/`, Networking: `net/`, Core kernel: `kernel/`, Architecture: `arch/`, Memory management: `mm/`, Headers: `include/`, Others: `block/`, `crypto/`, etc. }

```
1   /*  Thread A on CPU0 */   8   /*  Thread B on CPU1 */
2   /*set_cpumask_stuck()*/   9   /*set_cpumask_stuck()*/
3   set(thiscpu, STUCK);     10   set(thiscpu, STUCK);
4   set(thiscpu, DONE);      11   set(thiscpu, DONE);
5 + smp_mb();                12 + smp_mb()
6   if check(allcpu, DONE);  13   if check(allcpu, DONE);
7     reset_watchdog();      14     reset_watchdog();
            *** Watchdog never resets ***
Bug-inducing interleaving: 4 -> 6 -> 13 -> 11
```

Figure 2: A hardware-dependent concurrency bug [49] that manifests only on the `powerpc` architecture due to its weak memory ordering.

Kernel component	Bugs	Interrupt-based bugs
Drivers	88	29 (33.0%)
File systems	27	1 (3.7%)
Networking	34	4 (11.8%)
Core kernel	10	1 (10.0%)
Architecture	8	4 (50.0%)
Memory mgt.	12	7 (58.3%)
Headers	3	1 (33.3%)
Others	18	8 (44.4%)
Total	200	55 (27.5%)

Table 4: Distribution of concurrency bugs across kernel components, including the number and fraction of interrupt-based bugs per component.

As shown in Table 4, the majority of concurrency bugs reside in drivers (44.0%), file systems (13.5%), and networking code (17.0%). These subsystems are among the largest in the kernel and frequently involve interactions with hardware devices, or asynchronous I/O. Our findings are consistent with prior work, such as Shi et al. [52], who reported that 40% of kernel data races were found in drivers, 15% in file systems, and 13% in networking code. This alignment reinforces the observation that large, modular, and I/O-intensive subsystems are particularly susceptible to concurrency bugs.

Interrupt-based concurrency bugs in drivers. Among the 88 driver concurrency bugs in our dataset, 29 (33.0%) involve interrupt events, slightly higher than the overall average of 27.5%. This is unsurprising, as driver code frequently interacts with hardware via interrupts that may interleave with thread-based execution paths such as system calls. This result highlights the elevated concurrency risk in drivers and underscores the importance of analyzing driver code under realistic interrupt scenarios.

Control-path versus data-path concurrency in drivers. To further understand driver-specific concurrency bugs, we examine whether they occur in the control path or the data path. Control-path code manages device state transitions, such as registration, initialization, suspension, and reset, whereas data-path code handles routine I/O operations and data transfers between the device and the kernel. Despite the common assumption that control paths are relatively serialized and less error-prone, we find that 61 out of 88 driver concurrency bugs (69.3%) occur in control-path code.

Closer inspection reveals that many bugs arise from concurrent execution of logically conflicting operations, such as `registration`

Component	Order viol.		Atom. viol.		Deadlock		Other	
Drivers	20	(22.7%)	40	(45.5%)	22	(25.0%)	6	(6.8%)
File systems	3	(11.1%)	11	(40.7%)	10	(37.0%)	3	(11.1%)
Networking	10	(29.4%)	17	(50.0%)	6	(17.7%)	1	(2.9%)
Core kernel	2	(20.0%)	6	(60.0%)	1	(10.0%)	1	(10.0%)
Architecture	1	(12.5%)	2	(25.0%)	4	(50.0%)	1	(12.5%)
Memory mgt.	0	(0%)	9	(75.0%)	2	(16.7%)	1	(8.3%)
Headers	0	(0%)	3	(100%)	0	(0%)	0	(0%)
Others	0	(0%)	13	(72.2%)	4	(22.2%)	1	(5.6%)
Total	36	(18.0%)	101	(50.5%)	49	(24.5%)	14	(7.0%)

Table 5: Breakdown of concurrency bug root causes [40] (order violations, atomicity violations, deadlocks, and others) across kernel components.

versus `deregistration`, or `init` versus `reset`. Such operations can leave devices in inconsistent states, trigger use-after-free errors, or cause incomplete cleanup. This finding indicates that concurrency between conflicting device control operations is error-prone, and that stronger synchronization or serialization in control paths could substantially reduce real-world concurrency bugs [61].

Finding: Drivers account for 44.0% of concurrency bugs, and 69.3% of those are caused by logically conflicting device operations on the control path (e.g., `init` vs. `reset`).

Implication: Kernel drivers should be a priority for concurrency testing and hardening. Targeting control path logic, particularly antonymous operations, can reduce complexity while covering a large portion of real-world concurrency bugs.

4 Concurrency bug root causes and patterns

This section analyzes the root causes and patterns of kernel concurrency bugs, focusing on bug taxonomies, thread requirements for reproduction, and synchronization misuse.

4.1 Atomicity and order violations

Non-deadlock concurrency bugs constitute the majority of our dataset (151 out of 200). To understand their root causes, we follow the taxonomy proposed by prior work on user-space concurrency bugs, which categorizes such bugs—based on incorrect assumptions about how operations execute concurrently—into two types [40]: atomicity violations and order violations. An atomicity violation occurs when a developer intends a sequence of operations to execute without interference but fails to enforce this property, allowing concurrent execution to observe intermediate states. An order violation arises when an assumed ordering between concurrent operations is not properly enforced, permitting unintended reordering.

To classify bugs, we manually inspect bug-fix commits and the corresponding code changes to infer the violated concurrency assumption. Specifically, we determine whether a bug stems from missing atomicity across a sequence of operations or from an unexpected ordering between concurrent operations that violates the developer's intended execution order.

As shown in Table 5, atomicity and order violations account for 90.7% (137 out of 151) of non-deadlock concurrency bugs in our dataset. Prior studies of user-space applications reported a

```
1   /* Thread A on CPU0 */          16   /* Thread B on CPU1   */
2   /* tls_sw_recvmsg() */          17   /* tls_decrypt_done() */
3                                   18 + spin_lock_bh(...);
4                                   19   // initial: pending = 1
5                                   20   a = atomic_dec(pending);
6 + spin_lock_bh(...);              21
7   p = atomic_read(pending);       22
8 + spin_unlock_bh(...);            23
9   if (p)                          24
10    wait();                       25
11  else                            26
12    reinit_completion();          27
13                                  28   if (!a)
14                                  29     complete();
15                                  30 + spin_unlock_bh(...);
        *** Double completion, connection lost ***
Bug-inducing interleaving: 20 -> 7 ~ 12 -> 29
```

Figure 3: An atomicity violation in the TLS subsystem [58], where atomic memory accesses are insufficient to enforce atomic execution at the code-region level.

higher fraction—97%—of non-deadlock bugs falling into these two categories [40]. This result confirms that incorrect assumptions about atomicity and ordering remain the dominant root causes of concurrency bugs in both user-space and kernel code.

Figure 3 illustrates a representative atomicity violation in the TLS subsystem that leads to broken network connections. The bug occurs when two kernel threads concurrently execute `tls_sw_recvmsg()` and `tls_decrypt_done()`. One thread (`tls_decrypt_done()`) decrements a shared variable `pending` from 1 to 0, while the other thread (`tls_sw_recvmsg()`) observes `pending` as 0 and invokes `reinit_completion()`. This interferes with the `complete()` operation concurrently performed in `tls_decrypt_done()`, eventually causing connection loss. Notably, although atomic operations are used to update `pending`, the higher-level sequence of operations is not protected, demonstrating that atomic memory operations alone are insufficient to enforce atomicity at the control-flow level.

Our analysis identifies several recurring factors that make atomicity and order violations particularly prevalent in kernel code. First, the kernel must handle a wide variety of asynchronous execution sources—ranging from user-space system calls to hardware interrupts—making it difficult for developers to fully enumerate all interleavings. Second, the same code may execute under different preemption or scheduling models (e.g., real-time kernels), exposing it to substantially different interleavings than originally anticipated. Third, complex control flow and deep call chains complicate reasoning: a single function may be invoked from multiple call sites with distinct synchronization contexts and ordering assumptions. Finally, continuous kernel evolution through refactoring and performance optimizations can subtly alter concurrency behavior, invalidating assumptions that are previously correct.

While atomicity and order violations are prevalent, we also observe a reduction from the 97% reported in prior user-space studies to 90.7% in the kernel. The remaining 9.3% of non-deadlock bugs fall into the *other* category. These bugs often involve kernel-specific concurrency behaviors, such as interactions with timers, CPU hotplug operations, or deferred execution mechanisms. They confirm concurrency challenges that do not fit neatly into standard atomicity or order violation taxonomies and suggest the need for additional techniques tailored to kernel-specific concurrency behaviors.

Finding: Atomicity and order violations account for the vast majority (90.7%) of non-deadlock kernel concurrency bugs.

Implication: Classic concurrency bug patterns remain highly prevalent in the kernel, indicating that techniques targeting atomicity and ordering assumptions remain broadly effective. At the same time, a nontrivial set of kernel-specific concurrency bugs falls outside these patterns and often requires specialized, context-aware detection approaches.

4.2 Deadlocks

Deadlocks are a distinct class of concurrency bugs that manifest as indefinite blocking, where one or more threads wait for resources that will never be released. Because deadlocks differ fundamentally in both manifestation and root cause, they require specialized detection techniques [24, 44]. Consistent with prior studies [27, 39], we therefore analyze deadlocks separately from non-deadlock bugs.

In our dataset, deadlocks account for 24.5% (49 out of 200) of all concurrency bugs, which is comparable to the 29.5% reported in prior studies of user-space applications [40]. To better understand how deadlocks arise in the kernel, we examine the minimum number of threads required to reproduce each bug. In this analysis, we do not treat interrupt handlers as separate threads because interrupt handlers execute in the context of a preempted thread [45]. This modeling aligns with kernel execution semantics and reflects the actual conditions under which these bugs manifest.

Our analysis shows that 32.7% of deadlocks (16 out of 49) can occur with a single thread. These bugs—often referred to as double-lock bugs—arise when a thread re-enters code that attempts to acquire a lock it already holds. A common source of such reentrancy is indirect control flow via function pointers. For example, in Figure 4, `usb_function_deactivate()` acquires a spinlock and then invokes a chain of function calls that eventually reaches `composite_disconnect()`, which attempts to acquire the same lock again. This deadlock is particularly hard to detect because the call occurs through a function pointer (`disconnect()`), which can resolve to many implementations, only one of which causes the bug.

Another major cause of double-lock bugs is interrupt-driven reentrancy. In these cases, a thread acquires a lock and is interrupted before releasing it, and the interrupt handler, executing in the same thread context, attempts to acquire the same lock. For instance, one bug in our dataset involved a function that was assumed to always execute with interrupts disabled; along a less common execution path, however, it was invoked with interrupts enabled, allowing a subsequent interrupt handler to re-acquire a lock already held by the thread. Although both causes are well known [1, 6], their continued occurrence highlights the challenge in enforcing locking and interrupt assumptions across all call paths in kernel code.

The majority of the remaining deadlocks (63.3%) require two threads to manifest. These bugs are typically caused by cyclic lock dependencies, where two threads acquire locks in different orders, resulting in a circular wait. This distribution closely matches observations from prior user-space studies [40], which also found that two-thread deadlocks dominate.

```
1   /*      Thread A on CPU0     */
2   /* usb_function_deactivate() */
3   spin_lock_irqsave(lock);
4 + spin_unlock_irqrestore(lock);
5   usb_gadget_deactivate();
6     usb_gadget_disconnect();
7       gadget->udc->driver->disconnect();
8         configfs_composite_disconnect();
9           composite_disconnect();
10            spin_lock_irqsave(lock);
11 + spin_lock_irqsave(lock);
              *** Deadlock ***

Bug-inducing double locking: 3 -> 5~9 -> 10
```

Figure 4: A single-thread deadlock—also known as a double-lock bug—involving indirect control flow [16].

Component	Number of threads					
	1		2		>2	
Drivers	16	(18.2%)	71	(80.7%)	1	(1.1%)
File systems	2	(7.4%)	24	(88.9%)	1	(3.7%)
Networking	3	(8.8%)	31	(91.2%)	0	(0%)
Core kernel	1	(10.0%)	9	(90.0%)	0	(0%)
Architecture	3	(37.5%)	5	(62.5%)	0	(0%)
Memory mgt.	1	(8.3%)	10	(83.3%)	1	(8.3%)
Headers	0	(0%)	3	(100%)	0	(0%)
Others	0	(0%)	16	(100%)	0	(0%)
Total	28	(14.0%)	169	(84.5%)	3	(1.5%)

Table 6: Minimum number of threads required to reproduce concurrency bugs, grouped by kernel component.

Finding: About 32.7% of kernel deadlock bugs can occur with only one thread, often due to interrupt handlers acquiring locks already held by the interrupted thread.

Implication: Deadlock detection tools could prioritize the analysis of two-thread lock interactions, but must also model interrupts and check locking assumptions across different control flow paths to detect single-thread deadlocks.

4.3 Number of threads involved

Beyond deadlocks, we examine the minimum number of threads required to reproduce all kernel concurrency bugs in our dataset. This analysis provides a broader view of concurrency complexity in the kernel and helps assess whether kernel bugs typically require high degrees of concurrency to manifest.

As shown in Table 6, most bugs—197 out of 200—can be reproduced with two or fewer threads. Specifically, 28 bugs (14.0%) can occur with a single thread, 169 bugs (84.5%) require two threads, and only 3 bugs (1.5%) require more than two threads.

The distribution closely mirrors observations from prior studies of user-space concurrency bugs [40], which likewise found that most concurrency bugs involve one or two threads. Despite the kernel's scale and complexity, kernel concurrency bugs rarely require more than two threads to manifest. Therefore, tools that focus on exploring interleavings involving one or two threads can reasonably achieve high coverage of real-world kernel concurrency bugs while substantially reducing the search space. At the same time, unlike user-space applications, kernel concurrency is heavily influenced

Synchronization	Wrong choice		Missing operation		Incorrect critical section	
spinlock	5	(33.3%)	1	(6.7%)	9	(60.0%)
mutex	5	(35.7%)	4	(28.6%)	5	(35.7%)
rcu	1	(16.7%)	4	(66.7%)	1	(16.7%)
seqlock	1	(100%)	0	(0%)	0	(0%)
rw_semaphore	1	(20.0%)	0	(0%)	4	(80.0%)
rwlock	0	(0%)	0	(0%)	1	(100%)
memory barrier	1	(50.0%)	1	(50.0%)	0	(0%)
atomic instructions	3	(100%)	0	(0%)	0	(0%)
other	8	(66.7%)	3	(25.0%)	1	(8.3%)
Total	25	(42.4%)	13	(22.0%)	21	(35.6%)

Table 7: Breakdown of concurrency bugs due to synchronization misuse, by synchronization primitive and misuse type.

by interrupts. As discussed in §3.1, effectively modeling interrupt-driven execution is essential for exposing kernel concurrency bugs, even when only one or two threads are considered.

Finding: All but three concurrency bugs in our dataset can be reproduced with no more than two threads.

Implication: Concurrency bug detection tools should prioritize exploring one- and two-thread interleavings. For the kernel, however, such tools must also account for interrupt-driven execution to achieve comprehensive bug coverage.

4.4 Synchronization misuse

Kernel concurrency relies on a wide range of synchronization primitives, each designed for specific execution contexts and performance trade-offs. Misusing these primitives remains a major source of concurrency bugs [12, 27], particularly given the kernel's complex execution model and diverse concurrency scenarios. We therefore analyze how synchronization misuse contributes to kernel concurrency bugs and identify common patterns in incorrect synchronization design. Specifically, we categorize concurrency bugs in our dataset according to the type of synchronization misuse involved. We identify three major classes. First, *wrong choice* refers to cases where developers select an inappropriate synchronization primitive that cannot provide sufficient protection. A common example is the use of atomic operations to guard complex shared state that requires mutual exclusion. Second, *missing operation* captures cases where a required synchronization step is omitted along some execution path—for instance, failing to acquire or release a lock, or neglecting to wait for an RCU grace period after modifying shared data. Third, *incorrect critical section* refers to situations where synchronization is applied to the wrong code region: the protected region may be too narrow to cover all shared accesses, or overly broad in ways that introduce deadlocks or unnecessary contention.

As shown in Table 7, synchronization misuse accounts for a substantial 29.5% of concurrency bugs in our dataset. Among these, *wrong choice* is the most prevalent category (42.4%), followed by *incorrect critical section* (35.6%) and *missing operation* (22.0%). This distribution suggests that selecting an appropriate synchronization mechanism is often more challenging than applying it correctly.

Discovery Method	Bugs
Manual	56
Syzbot	18
Other tools	11
Total	85

Table 8: Discovery methods for concurrency bugs with known provenance (85 out of 200).

Breaking down misuses by synchronization primitive provides additional insight. Spinlocks and mutexes account for most misuse cases. Spinlock-related bugs are heavily skewed toward *incorrect critical sections* (60.0%), reflecting the difficulty of precisely identifying which code regions must be protected in preemptible and interruptible contexts. In contrast, RCU-related bugs are dominated by *missing operations* (66.7%), typically due to omitted calls to `synchronize_rcu()` needed to ensure a grace period after updates. Atomic instructions appear exclusively in the *wrong choice* category. This pattern is not accidental: atomic operations provide only low-level, single-variable guarantees and do not naturally extend to protecting multi-step operations or larger critical regions, making them easy to misuse when stronger synchronization is required.

These synchronization misuses often stem from incomplete reasoning about possible concurrent execution contexts. For example, as illustrated in Figure 1, developers use a spinlock to serialize two functions that are known to execute concurrently. However, they overlook the possibility that one function may be interrupted in the middle, allowing an interrupt handler to access the same shared state and violate the intended mutual exclusion. Such cases demonstrate how subtle execution contexts—especially interrupts—can invalidate otherwise reasonable synchronization decisions.

Finding: Synchronization misuse contributes to a high percentage (29.5%, 59/200) of kernel concurrency bugs.

Implication: Automated analysis that can infer all possible concurrent execution contexts can help avoid synchronization misuse. Future research should focus on techniques that assist developers in selecting and correctly applying synchronization mechanisms based on the full range of possible kernel execution scenarios.

5 Concurrency bug discovery

While much prior work has focused on designing automated detection techniques [29, 32, 35, 48], it remains unclear to what extent such tools actually account for real-world concurrency bug discoveries. In this section, we examine the discovery sources of concurrency bugs in our dataset to assess the current balance between automated tooling and human-driven debugging.

To determine how each bug is discovered, we manually examine bug-fix commits and associated discussions, including `Reported-by` tags, references to testing tools, and links to mailing list reports. For 85 of the 200 bugs, sufficient information is available to identify the discovery method; the remaining bugs lack explicit attribution and are excluded from the breakdown below.

As shown in Table 8, most concurrency bugs with known discovery sources (65.9%, 56 out of 85) are discovered manually by

developers through code review, in-house testing or debugging. This observation is consistent with prior bug studies [27] and suggests that developer insight and deep familiarity with subsystem behavior remain central to uncovering concurrency bugs.

Automated tools account for the remaining 29 bugs (34.1%). Among these, 18 bugs (21.2%) are reported by Syzbot [54], while 11 bugs (12.9%) are discovered using other testing approaches [4, 5, 7], including kernel-specific testing tools, custom test suites, and platform- or vendor-specific frameworks. These tools typically rely on dynamic execution combined with sanitizers such as KASAN, KCSAN, and Lockdep to detect memory errors, data races, and deadlocks. While automated tools do contribute meaningfully to concurrency bug discovery, their coverage remains limited. In particular, their effectiveness often depends on the quality of test harnesses, kernel configuration choices, and the ability to reproduce timing-sensitive interleavings—limitations that mirror the manifestation challenges identified in earlier sections (§3.1, §3.2, and §4).

Finding: Among concurrency bugs with known discovery methods, most (65.9%) are found through manual developer effort. Automated tools account for 34.1%.

Implication: There is substantial room for automated bug discovery tools to improve, particularly in generating actionable diagnostics to aid kernel developers.

6 Concurrency bug repair

Diagnosing and fixing concurrency bugs imposes significant cost on kernel development. In this section, we examine kernel concurrency bug repair along two dimensions: how long such bugs persist before being fixed, and the complexity of the corresponding patches. We compare these characteristics against kernel bugs of all types to quantify the additional burden imposed by concurrency bugs.

6.1 Bug persistence time

To understand how long kernel concurrency bugs exist, we compare their persistence time with that of kernel bugs of all types. We define the persistence time as the number of days between the commit that introduced the bug and the commit that fixed it, capturing how long a bug remains latent in the codebase before it is resolved.

We identify bug introduction commits using the `Fixes` tag, which is widely used in Linux kernel development to reference the original faulty commit. Our analysis includes 104 concurrency bugs and 57,217 kernel bugs of all types with identifiable introduction commits from the same time period.

Figure 5 presents the cumulative distribution of persistence times for concurrency bugs and kernel bugs of all types, which reveals a substantial disparity. On average, concurrency bugs persist for 1,258 days before being fixed, compared to 765 days for kernel bugs overall—over a 1.6× difference. One contributing factor is the difficulty of diagnosing and reproducing concurrency bugs. In our dataset, only 7.5% of concurrency bugs are reported with a reproducer, substantially complicating root cause analysis and validation and contributing to longer bug lifetimes.

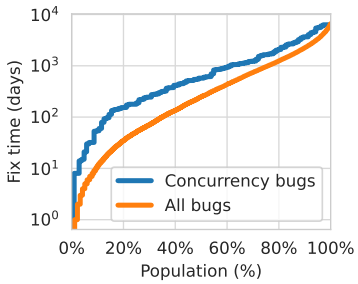


Figure 5: Persistence time of kernel concurrency bugs versus kernel bugs of all types (denoted as all bugs), showing that concurrency bugs exist for longer periods.

Metric	All bugs	Concurrency bugs
# of directories	1.1	**1.2**
# of files	1.4	**1.7**
# of functions	2.0	**3.3**
# of hunks	4.4	**7.5**
# of insertions	11.4	**21.1**
# of deletions	6.6	**9.9**

Table 9: Average patch characteristics for kernel bugs of all types (all bugs) and for concurrency bugs.

6.2 Patch complexity

We next compare the characteristics of patches used to fix concurrency bugs with those used to fix kernel bugs of all types. We assess patch complexity using several metrics, including the number of modified directories, files, functions, and hunks, as well as the number of lines added and removed.

At coarse granularity, concurrency bug patches and patches for bugs of all types appear similar: concurrency patches modify an average of 1.2 directories and 1.7 files, compared to 1.1 directories and 1.4 files for kernel bug fixes overall. However, differences become more pronounced at a finer granularity. On average, concurrency bug patches modify 3.3 functions and 7.5 hunks, whereas fixes for bugs of all types modify only 2.0 functions and 4.4 hunks. Concurrency bug patches also introduce more changes, with 21.1 lines of insertions and 9.9 lines of deletions on average—nearly double those of kernel bug fixes overall (11.4 insertions and 6.6 deletions). The results indicate that concurrency bug fixes involve finer-grained and more scattered modifications rather than broad structural changes. This pattern is consistent with the nature of concurrency repair, which often requires coordinated updates across multiple code locations, such as adjusting lock acquisition and release, reordering accesses to shared data, or inserting synchronization primitives at several points. Such localized yet nontrivial edits increase patch complexity and raise the likelihood of introducing regressions.

Finding: Concurrency bugs persist longer and require more complex patches than kernel bugs of all types.

Implication: The high cost of repairing concurrency bugs highlights the need for improved diagnostic support and automated repair techniques tailored to concurrency bugs.

7 Discussion

Beyond quantitative results, our study identifies two broader implications: the relationship between concurrency and performance regressions, and the security interpretation of concurrency bugs.

Concurrency-induced performance regressions. While our study focuses on correctness bugs, we observe that concurrency issues can also manifest as severe performance regressions under high levels of concurrency. In our dataset, we identify a concurrency bug that requires more than three threads to expose and manifests as a performance degradation rather than a functional failure. In the `mm` subsystem, developers introduced a resource-sharing mechanism that improves throughput for moderate levels of concurrency (e.g., 2–20 threads). But under higher degrees of concurrency (e.g., hundreds of threads), this optimization leads to severe contention and significant slowdowns. This case highlights a subtle class of concurrency bugs that blur the boundary between correctness and performance. Such bugs may evade testing approaches that focus on explicit failure symptoms or modest concurrency levels, yet can have substantial impact in real-world deployments.

Security interpretation of kernel concurrency bugs. None of the 200 concurrency bugs in our dataset are assigned CVEs. This should not be interpreted as evidence that kernel concurrency bugs lack security relevance. Rather, it reflects current practices in the Linux kernel community, where CVE assignment is optional and typically reserved for bugs with clear exploitability. For instance, in 2022, only 281 Linux kernel CVEs were assigned, despite over 20,000 kernel bug fixes during the same period.

In practice, concurrency bugs often manifest as crashes, deadlocks, or subtle state corruptions, making their exploitability difficult to assess. Nevertheless, they can undermine system availability, reliability, and isolation, and in some cases may serve as building blocks for more sophisticated exploits. Consequently, relying solely on CVE statistics likely underestimates the security impact of kernel concurrency bugs, highlighting the need for more careful consideration of their security implications.

8 Related work

Studies of user-space concurrency bugs. Concurrency bugs in user-space applications have been extensively studied [17, 27, 40, 53, 55] using broadly similar empirical methodologies (§2). These studies typically construct datasets of one to a few hundred real-world bugs collected from representative applications, followed by detailed manual analysis to identify common bug patterns, root causes, manifestation conditions, and repair strategies. Collectively, this line of work has yielded influential insights into user-space concurrency bugs and has informed the design of testing tools, debugging techniques, and programming models. Notably, Lu et al. [40] conducted one of the most comprehensive early studies, analyzing 105 concurrency bugs across four large applications to characterize their causes, manifestations, and fixes.

In contrast to these efforts, our study focuses exclusively on the kernel domain. While we revisit several observations established in user-space studies, we show that kernel concurrency bugs exhibit distinct characteristics due to factors unique to kernel execution, such as interrupt-driven concurrency, and hardware dependence. Thus, our work complements prior user-space studies by extending understanding to kernel concurrency and by identifying challenges and opportunities that are specific to kernel programming.

Studies of specific kernel concurrency bug classes. Several efforts have examined particular classes of kernel concurrency bugs. For example, a prior study [57] analyzed the distribution of kernel data races over years, modules and kernel versions, and performed detailed analysis on a subset of 30 sampled data races. Similarly, Bai et al. [18] analyzed 949 use-after-free bugs in the Linux kernel drivers and found that nearly 42% of bugs were caused by concurrency, highlighting the prevalence of concurrency-related errors in kernel code.

Our work differs from these studies in both scope and objective. Rather than focusing on a single bug category such as data races or use-after-free bugs, which do not always correspond to full-fledged concurrency bugs, we perform a comprehensive and systematic analysis across all types of kernel concurrency bugs. By manually analyzing a dataset of 200 real-world concurrency bugs, we are able to quantitatively characterize their manifestation conditions, root causes, discovery mechanisms, and repair costs. This broader perspective enables a holistic understanding of kernel concurrency bugs and provides empirical guidance for improving kernel reliability and concurrent kernel design.

9 Conclusion

This paper presents the first comprehensive study of concurrency bugs in the Linux kernel. We analyze 200 real-world kernel concurrency bugs fixed over a 4.5-year period and systematically characterize their manifestation conditions, root causes, discovery processes, and repair characteristics. Our results show that kernel concurrency bugs differ fundamentally from those in user-space applications, particularly in their reliance on interrupts, hardware dependencies, and synchronization patterns, highlighting the need for kernel-specific techniques for addressing concurrency bugs.

Acknowledgments

We thank anonymous reviewers for their insightful feedback. We also thank the Reliable and Secure System Lab members for their detailed and helpful comments on this work and earlier drafts. This work was funded in part by the National Science Foundation grants CNS-2140305 and CNS-2145888 and gifts from Google and Intel.

References

[1] Unreliable Guide To Locking — The Linux Kernel documentation, September 2017. [Online; accessed 30. Nov. 2023].
[2] What to do about cve numbers. https://lwn.net/Articles/801157/, 2019.
[3] Cve. https://cve.mitre.org/, 2022.
[4] KMS Tests: igt-gpu-tools Reference Manual, November 2023. [Online; accessed 30. Nov. 2023].
[5] lkp-tests, November 2023. [Online; accessed 30. Nov. 2023].
[6] Locking — The Linux Kernel documentation, November 2023. [Online; accessed 30. Nov. 2023].
[7] tests/tcrypt-compat-test · master · cryptsetup / cryptsetup · GitLab, November 2023. [Online; accessed 30. Nov. 2023].
[8] How kernel cve numbers are assigned. https://lwn.net/Articles/978711/, 2024.
[9] The kernel becomes its own cna. https://lwn.net/Articles/961961/, 2024.
[10] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 421–432, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. SHARD: Fine-Grained kernel specialization with Context-Aware hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2435–2452. USENIX Association, August 2021.

[12] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. Kard: lightweight data race detection with per-thread memory protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 647–660, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Daroc Alden. Finding locking bugs with smatch. https://lwn.net/Articles/1023646/, 2025.

[14] Jade Alglave, Will Deacon, Feng Boqun, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. Calibrating your fear of big bad optimizing compilers. https://lwn.net/Articles/799218/, 2019.

[15] Jade Alglave, Will Deacon, Feng Boqun, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. Who's afraid of a big bad optimizing compiler? https://lwn.net/Articles/793253/, 2019.

[16] Sriharsha Allenki. usb: gadget: Fix spinlock lockup on usb_function_deactivate, 2020.

[17] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1):1–15, 2017.

[18] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency Use-After-Free bugs in linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 255–268, Renton, WA, July 2019. USENIX Association.

[19] Lu Baolu. iommu/vt-d: Fix lockdep splat in sva bind()/unbind(), 2020.

[20] Matthias Brun, Reto Achermann, Tej Chajed, Jon Howell, Gerd Zellweger, and Andrea Lattuada. Beyond isolation: Os verification as a foundation for correct applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HotOS '23, page 158–165, New York, NY, USA, 2023. Association for Computing Machinery.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems - TOCS*, 30:1–49, 02 2012.

[22] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 73–88, New York, NY, USA, 2001. Association for Computing Machinery.

[23] The Kernel Development Community. Linux rcu documentation, 2020.

[24] Jonathan Corbet. The kernel lock validator. https://lwn.net/Articles/185666/, 2006.

[25] Jonathan Corbet. Interrupts, threads, and lockdep. https://lwn.net/Articles/321663/, 2009.

[26] Jonathan Corbet. Indirect branch tracking for intel cpus. https://lwn.net/Articles/889475/, 2022.

[27] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 221–230, New York, NY, 2010. IEEE, IEEE Press.

[28] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, Broomfield, CO, October 2014. USENIX Association.

[29] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 328–343, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 35–51, New York, NY, USA, 2023. Association for Computing Machinery.

[32] Sishuai Gong, Wang Rui, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. *Snowplow: Effective Kernel Fuzzing with a Learned White-box Test Mutator*, page 1124–1138. Association for Computing Machinery, New York, NY, USA, 2025.

[33] ISO/IEC JTC1 SC22 WG21 P0124R7. *Linux-Kernel Memory Model*. ISO, Geneva, Switzerland, 2017.

[34] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, New York, NY, 2019. IEEE Press.

[35] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2104–2121. IEEE Computer Society, 2023.

[36] Andrey Konovalov. Coverage-guided usb fuzzing with syzkaller. *Talk, Offensive-Con, Berlin. Feb*, 2019.

[37] Chao Li, Rui Chen, Boxiang Wang, Zhixuan Wang, Tingting Yu, Yunsong Jiang, Bin Gu, and Mengfei Yang. An empirical study on concurrency bugs in interrupt-driven embedded software. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1345–1356, New York, NY, USA, 2023. Association for Computing Machinery.

[38] Congyu Liu, Sishuai Gong, and Pedro Fonseca. Kit: Testing os-level virtualization for functional interference bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 427–441, New York, NY, USA, 2023. Association for Computing Machinery.

[39] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, Berkeley, CA, 2013. USENIX Association.

[40] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. Association for Computing Machinery.

[41] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[42] Paul McKenney. The rcu api, 2019 edition, 2019.

[43] Paul E. McKenney. Is parallel programming hard, and, if so, what can you do about it?, 2025.

[44] Ingo Molnar and Arjan van de Ven. Runtime locking correctness validator. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt, 2022.

[45] Gary Lawrence Murphy, Richard Sevenich, Tim Waugh, Red Hat UK, Juan-Mariano de Goyeneche, Manuel J Petit de Gabriel, Tom Lees, GNU Debian, Richard West, Christian Poellabauer, et al. The linux kernel. *Resource*, 2000:000.

[46] Tapti Palit and Pedro Fonseca. Kaleidoscope: Precise invariant-guided pointer analysis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 561–576, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: ten years later. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 305–318, New York, NY, USA, 2011. Association for Computing Machinery.

[48] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing usb drivers by device emulation. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[49] Nicholas Piggin. powerpc/watchdog: Fix missed watchdog reset due to memory ordering race, 2021.

[50] Alexander Potapenko, Dmitry Vyukov, Kees Cook, Marco Elver, and Paul McKenney. Kernel sanitizers office hours. lpc '24, 2024.

[51] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. SymDrive: Testing drivers without devices. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 279–292, Hollywood, CA, October 2012. USENIX Association.

[52] Jianjun Shi, Weixing Ji, Yizhuo Wang, Lifu Huang, Yunkun Guo, and Feng Shi. Linux kernel data races in recent 5 years. *Chinese Journal of Electronics*, 27(3):556–560, 2018.

[53] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 865–878, New York, NY, USA, 2019. Association for Computing Machinery.

[54] Dmitry Vyukov. syzbot: automated kernel testing, 2018.

[55] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in node.js. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 520–531, New York, NY, 2017. IEEE, IEEE Press.

[56] Todd Warszawski and Peter Bailis. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 5–20, 2017.

[57] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, New York, NY, 2020. IEEE, IEEE Press.

[58] Vinay Kumar Yadav. net/tls: fix race condition causing kernel panic, 2020.

[59] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*, 2012.

[60] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM*

SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, page 26–36, New York, NY, USA, 2011. Association for Computing Machinery.

[61] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. Ddrace: finding concurrency uaf vulnerabilities in linux drivers with directed fuzzing. In *Proceedings of the 32nd USENIX Conference on Security Symposium*, SEC '23, USA, 2023. USENIX Association.