Concurrency Fuzzing of the Linux Kernel with eBPF

Jiacheng Xu[*]
Zhejiang University[†]

Dylan Wolff[*‡]
National University of Singapore

Xing Yi Han
National University of Singapore

Jialin Li
National University of Singapore

Abhik Roychoudhury
National University of Singapore

Abstract

Concurrency is indispensable for modern software systems to meet performance and scalability demands, yet concurrency bugs remain notoriously difficult to detect and reproduce. Controlled Concurrency Testing (CCT) mitigates this challenge by systematically exploring thread interleavings through scheduling control. However, existing CCT approaches for OS kernels largely rely on external enforcement mechanisms, such as custom hypervisors or invasive kernel patches, resulting in substantial overhead and limited maintainability and extensibility. In this work, we present SECT, the first kernel-native concurrency fuzzing framework that rethinks scheduling as a first-class exploration mechanism. SECT introduces a novel CCT scheduler with temporal isolation scheduling and embeds programmable scheduling policies directly into the kernel dispatch path via eBPF, enabling fine-grained control over thread interleavings without customized hypervisors or extensive kernel core modification. In addition, SECT provides a preemption-safe instrumentation mechanism for injecting scheduling points at critical kernel events and incorporates a two-phase fuzzing workflow to jointly explore both sequential and concurrent behaviors. Our evaluation demonstrates that SECT achieves 38% more branches, 57% overhead reduction and 11.4× speed-up in bug exposure compared to a leading state-of-the-art kernel concurrency fuzzer. Moreover, SECT discovers eight previously unknown concurrency-related bugs in the Linux kernel, six of which have already been confirmed and fixed by developers.

1 Introduction

In 21st-century computing, the end of Moore's Law has led to a shift towards increasingly parallel and distributed software systems [23, 62, 63]. Although concurrency is essential for achieving scalability and performance in modern OS kernels, it also introduces complex synchronization mechanisms that are notoriously error-prone [33]. These bugs can lead to severe consequences due to the fundamental role of kernels in the software stack. One prominent example is the Dirty COW vulnerability [47], where a race condition exploited a flaw in the kernel's memory management. This allowed an attacker to gain write access to sensitive memory mappings, facilitating local privilege escalation and potentially affecting millions of computing devices. Thus, finding concurrency bugs is essential in ensuring a safe and reliable kernel.

Concurrency bugs are often more challenging to discover and identify compared to their sequential counterparts [14]. They are triggered with two important ingredients: execution of the buggy code path and observation of a particular thread interleaving. Finding these bugs thus requires exploring not just the input space but also the thread interleaving space, which is both difficult to control and exponentially large in the number of concurrently executed instructions.

Historically, kernel concurrency testing has been constrained by a *scheduling dilemma*. Conventional stress-testing [5, 16, 44, 60] and coverage-guided fuzzers [25, 28, 36, 68] typically treat the fairness- and latency-oriented kernel scheduler as an opaque component that must be overcome indirectly via exceptional workloads to manifest rare concurrency bugs. To bypass this limitation, the state-of-the-art concurrency fuzzers have relied on external interposition via delay injection, modified hypervisors or invasive kernel patches [27, 34, 35, 37]. While making notable progress over stress-testing by more directly affecting schedules from outside the kernel context, these approaches suffer from the following challenges:

C0: Limited scheduling control. Many concurrency fuzzers either provide only weak control over thread scheduling by injecting delays [15, 67] or limited coarse-grained scheduling hooks [34, 37]. With only limited, coarse-grained control, many concurrency bugs may be rendered unreachable during testing. While trading completeness for performance is sometimes reasonable, these works fundamentally cannot enforce a higher degree of control, even in cases when performance is not an issue.

C1: High overhead. Traditional concurrency fuzzers with

[*]Both authors contributed equally
[†]Research conducted primarily at the National University of Singapore
[‡]Corresponding Author: wolffdy0@gmail.com

Table 1: State-of-the-art kernel concurrency fuzzers.

Tool	Granular Control	Low-Overhead	Hypervisor Agnostic	Forward Compatible	Highly Extensible
Razzer [34]	✗	✗	✗	✓	✗
Snowboard [27]	✓	✗	✗	-	✗
SKI [26]	✓	✗	✗	✗	✗
KRACE [67]	✗	✗	✓	✗	✗
SegFuzz [35]	✓	✗	✗	✗	✗
SCT	✓	✓	✓	✓	✓

full control typically operate at the hypervisor level [27, 35]. These hypervisor-specific techniques are extremely heavyweight and exhibit poor scalability. For example, Snowboard [27] introduces a substantial performance overhead, slowing execution by more than 10×. Additionally, the limitations of the hypervisor's functionality restrict SegFuzz's support [35] for parallel fuzzing processes, resulting in a significant reduction in overall throughput during fuzzing (c.f. Section 6.2).

C2: Portability and maintenance burden. OS kernels have been under continuous development, with frequent updates and ongoing evolution. In addition to custom hypervisors, existing fuzzers usually require extensive customization tailored to specific kernel versions [34, 35, 67]. For instance, Krace [67] involves invasive kernel changes, introducing a patch of over 10k lines of code across 105 files. Such tight coupling substantially increases the cost and effort required to port the fuzzer to newer kernel versions, and may also reduce its effectiveness. As a result, the long-term usability and maintainability of such approaches are often compromised.

C3: Restricted extensibility. Scheduling algorithms in existing concurrency fuzzers are often bespoke and tightly integrated with critical components of the underlying implementation. Extending or duplicating them in a new codebase is typically extremely challenging and demands deep domain expertise. As a result, traditional fuzzers that rely on default scheduling behavior are often still used to attempt to find concurrency bugs in practice.

Table 1 summarizes existing kernel concurrency fuzzing approaches and their limitations. In this work, we rethink kernel concurrency testing by shifting from external intervention to native exploration. We propose a *scheduler-as-an-explorer* paradigm: rather than treating the scheduler as an opaque entity to be perturbed externally, we elevate it into a first-class mechanism for systematic interleaving exploration. We propose SCT (**S**ched-**E**xt **C**oncurrency **T**ester), the first *kernel-native* concurrency fuzzing framework. SCT introduces a novel CCT scheduler based on temporal isolation beyond traditional performance-driven scheduling and embeds interleaving exploration directly into the kernel scheduling path via eBPF. Therefore, SCT enables fine-grained and lightweight control over thread interleavings without requiring hypervisor modifications or invasive kernel patches. In addition, SCT incorporates programmable and testing-oriented scheduling algorithms and supports their modular extension, enabling future research in concurrency exploration strategies. To further increase exploration granularity, SCT provides a preemption-safe instrumentation mechanism for injecting scheduling points at critical kernel events. Finally, SCT integrates with Syzkaller to facilitate a two-phase fuzzing workflow, jointly exploring sequential and concurrent behaviors.

We apply SCT to recent Linux kernel versions and conduct extensive evaluation. Our results demonstrate that the *scheduler-as-an-explorer* paradigm is highly effective: SCT achieves 38% higher branch coverage than the state-of-the-art concurrency fuzzer SegFuzz. SCT also accelerates bug reproduction by 11.4× on known concurrency bugs and provides a 57% reduction in overhead compared to SegFuzz. Additionally, SCT discovers eight previously unknown concurrency-related bugs in recent Linux kernels, six of which have already been confirmed and fixed by developers.

In summary, our work makes the following contributions:

- We identify the *scheduling dilemma* as a fundamental limitation of existing kernel CCT techniques and propose a new *scheduler-as-an-explorer* paradigm, mitigating the challenges **C0**, **C1**, **C2** and **C3**.
- We design and implement SCT, the first kernel-native concurrency fuzzing framework that features the temporal isolation scheduling with programmable policies, the safe preemption injection mechanism, and the two-phase input-and-concurrency fuzzing workflow.
- We evaluate SCT on recent Linux kernel versions, demonstrating substantial improvements in coverage, efficiency, and bug discovery effectiveness.
- We make SCT available and open-source upon acceptance of this work, lowering the entry barrier for research in CCT and concurrency fuzzing for kernel code.

2 Background and Related Work

2.1 Kernel Fuzzing

Fuzzing has become the de facto standard technique for dynamically uncovering OS kernel vulnerabilities [17, 24, 40, 41, 46, 56]. Fuzzers [25, 30, 55, 61, 65, 71] explore kernel states in a biased randomized search by prioritizing test inputs that cover new branches. For instance, Syzkaller, the state-of-the-art kernel fuzzer, [28] generates system call (syscall) programs as inputs for the Linux kernel, primarily meant for sequential execution; it has identified over 5,000 bugs in the upstream Linux kernel but only roughly 10% are concurrency-related issues. The approach taken by these conventional, sequential-oriented fuzzers enforces *no control* over the thread interleavings, relying entirely on the native OS scheduler; this greatly restricts their ability to explore the scheduling space to discover rare concurrency bugs.

To fill this gap, several concurrency-oriented fuzzers have been proposed to handle exploration of concurrency space. Krace [67] introduces an alias coverage that is implemented to guide fuzzing towards new interleavings. However, Krace adopts a lightweight, but weak form of scheduling control by injecting random delays at memory access points. Delay injection alone, however, is likely to miss many uncommon thread interleavings during fuzzing [37] (**C0**). Moreover, Krace requires substantial modifications of the Linux kernel, making it difficult to maintain as the kernel evolves over time (**C2**).

Razzer [34] modifies the hypervisor to include hypercalls that, when called by a kernel thread, would install hardware breakpoints on the vCPU for scheduling. These hardware breakpoints are installed at memory reads and writes, which are pre-determined by a heuristic-driven static analysis. To trigger a race, Razzer does a single step on the specified vCPU before resuming execution on all vCPUs. However, Razzer can still only enforce control at a small number of hardware breakpoints (typically four on current generation Intel processors), limiting its ability to explore schedules involving more than four events (**C0**).

SegFuzz [35], which builds on top of the Razzer architecture, improves on Razzer's limitation by replacing hardware breakpoints at runtime, enforcing a fully controlled schedule sent by the fuzzer process via hypercalls. Nevertheless, we observe that SegFuzz, constrained by its reliance on the hypervisor and virtualized hardware: it is limited to a single fuzzing process per virtual machine and thus does not scale well to larger fuzzing campaigns (**C1**).

SKI [26] and Snowboard [27] use a similar method of sending hypercalls to suspend and resume the execution of vCPUs according to a schedule. However, instead of hardware breakpoints, they make use of the ability to set a thread's CPU affinity to pin each thread to its own vCPU. While this avoids the limited parallelism of SegFuzz, it introduces high latency for fine-grained context switches, as it does not leverage hardware features to achieve fast preemption (**C0**).

2.2 Controlled Concurrency Testing

A strongly related research to concurrency fuzzing is Controlled Concurrency Testing (CCT). These works focus on exploring the space of interleavings of a *fixed input*, rather than simultaneously searching the space of inputs and interleavings as in concurrency fuzzing. Many systematic [45, 58] and randomized [59, 66] CCT scheduling algorithms have been developed which can effectively navigate the interleaving space. In practice, randomized algorithms tend to outperform their systematic counterparts at scale [64], and some of these randomized algorithms even provide probabilistic guarantees for finding certain classes of bugs [19, 70, 72]. These algorithms have seen deployments in large-scale, industrial settings at companies like Amazon [18] and Microsoft [22] for testing user-space programs. However, with the notable exception of SKI [26], which implements only the PCT [19] algorithm, CCT scheduling algorithms remain largely unevaluated on OS-kernel software, which represents one of the most important target concurrent applications. As such an explicit goal of this work is to make these algorithms available in an extensible framework for kernel code (**C3**).

2.3 eBPF Technology

The Extended Berkeley Packet Filter (eBPF) enables dynamic in-kernel customization at runtime without modifying the Linux kernel code. It allows developers to load and run restricted programs in a safe, sandboxed, in-kernel virtual machine (VM) [21, 43]. These programs attach to specific kernel hook points and interact with the kernel through vetted helper calls. By allowing developers to inject custom logic into the kernel at runtime, eBPF provides a flexible and low overhead means to monitor and manipulate kernel behavior.

One notable application of eBPF is the `sched_ext` [4, 32] scheduler class, which has been officially merged into the mainline Linux kernel since version 6.12. Traditionally, process scheduling in Linux has been limited to a fixed set of in-kernel policies, such as the Completely Fair Scheduler (CFS) [2]. `sched_ext` decouples scheduling policy from the core kernel scheduling mechanism. It exposes user-defined scheduling logic through public interfaces [69], and allows developers to implement customized scheduling policies through eBPF programs. Only tasks explicitly set with the `SCHED_EXT` scheduling class are affected by programmable policies. While `sched_ext` is originally designed for domain-specific schedulers for performance optimization, our novelty is to leverage this feature to support kernel fuzzing for CCT.

3 Motivation

Many concurrency bugs manifest only under specific instruction orderings across multiple threads. However, kernel schedulers are primarily designed to optimize fairness and performance rather than to expose such rare thread interleavings, making the natural occurrence of concurrency bugs unlikely. Thus, the ability to manipulate scheduling behaviour can be a powerful strategy to find and reproduce concurrency bugs.

3.1 Motivating Example

```
1  // fs/jfs/jfs_discard.c
2  int dbUnmount(struct inode *ipbmap, ...) {
3      // ...
4      kfree(bmp);
5      JFS_SBI(ipbmap->i_sb)->bmap = NULL;
6  }
7
8  int jfs_ioc_trim(...) {
9  -   struct bmap *bmp = JFS_SBI(ip->i_sb)->bmap;
10 +   struct bmap *bmp;
11     // ...
12 +   down_read(&sb->s_umount);
```

```
13 +   bmp = JFS_SBI(ip->i_sb)->bmap;
14
15 -   if (minlen > bmp->db_agsize ||
16 +   if (bmp == NULL ||
17 +       minlen > bmp->db_agsize ||
18         start >= bmp->db_mapsize ||
19 -       range->len < sb->s_blocksize ||
20 +       range->len < sb->s_blocksize) {
21 +       up_read(&sb->s_umount);
22         return -EINVAL;
23 +   }
24     // ...
25 +   up_read(&sb->s_umount);
26     range->len = trimmed << sb->s_blocksize_bits;
27     return 0;
28 }
```

Listing 1: A patch completing a previous partial fix in `jfs` filesystem.

To show the need for a CCT tool for the Linux kernel, we examine a concurrency bug that we found in the `fs/jfs` module that results in a null pointer dereference. It is closely related to a recent bug which had been reported as fixed [13]. The previous issue was a use-after-free (UAF) due to missing synchronization in the `jfs_ioc_trim` function that allowed it to interleave execution with `dbUnmount`. We show both functions in Listing 1, along with the full patch we provided to developers. The UAF was triggered when `dbUnmount` freed `bmap` in one thread while `jfs_ioc_trim` still had a reference to it in another thread. It was resolved by adding a read-write semaphore such that the free cannot interleave between obtaining the pointer and dereferencing it in `jfs_ioc_trim`.

Unfortunately, the accepted original patch omitted the null check at line 16, and was thus still susceptible to a null pointer dereference. This issue was not caught by kernel developers but was found by SCT. In particular, `JFS_SBI(ipbmap->i_sb)->bmap` is set to null in `dbUnmount` after it is freed. Thus, while the UAF bug can no longer occur, if the critical section in `dbUnmount` executes first, the resulting pointer will be null instead of the old address. As the pointer is dereferenced in `jfs_ioc_trim`, this leads to a null pointer dereference under some interleavings. We observed that this bug cannot be reproduced by the default OS scheduler in 10,000 executions of a syscall sequence. In contrast, SCT consistently reproduces the issue in fewer than ten executions. This case highlights the need for more effective concurrency testing for kernel code.

3.2 From Scheduling Black-Box to Scheduler-as-an-Explorer

In principle, kernel concurrency testing could be realized by implementing a dedicated scheduler class directly within the Linux kernel. Such an approach would offer stronger control over scheduling decisions, but it comes at the cost of invasive kernel modifications, compromised safety, and limited deployability. More importantly, tightly coupling a concurrency-testing scheduler with the kernel core contradicts our goal of enabling flexible and extensible exploration of scheduling behaviors (**C3**). To realize the *scheduler-as-an-explorer* paradigm while preserving kernel safety and practicality, we build our CCT scheduler on top of the `sched_ext` infrastructure and eBPF. This design choice enables systematic control over thread interleavings without modifying the kernel core or relying on external enforcement mechanisms.

Safety and Reliability. The `sched_ext`-based schedulers are validated and sandboxed by the eBPF compile-time verification and runtime checks, which enforce strict safety guarantees. These mechanisms eliminate a broad class of implementation errors that could otherwise lead to kernel crashes. Avoiding these failures is particularly important for kernel concurrency testing, as unintended crashes can prematurely terminate testing campaigns.

Deployability and Portability. A `sched_ext` scheduler can be loaded, attached, and replaced at runtime without kernel rebuild or system reboot. The eBPF and `sched_ext` APIs have remained stable over recent years, providing a consistent interface for scheduler development [4]. In addition, the presence of BPF Type Format [1], libbpf [6], and the eBPF compiler enables scheduler portability without rebuilds across kernel versions. While we cannot guarantee complete forward compatibility, we believe that our approach significantly reduces maintenance effort and improves the long-term practicality of kernel concurrency testing (**C2**).

4 Design of SCT

SCT alternates between two-phases of fuzzing: sequential input exploration and concurrent schedule fuzzing. The first phase is sequential fuzzing, which aims to explore the input space with traditional generation and mutation strategies, maximizing code coverage. The second phase, concurrency fuzzing, constitutes the core of SCT and focuses on exploring the thread interleaving space.

Figure 1 illustrates the design components and the overall workflow of SCT. To increase exploration granularity in this phase, SCT optionally instruments the kernel under test with additional preemption-safe scheduling points at compile time (①). The SCT scheduler can be loaded once at test-time (②) without needing to reboot or recompile the kernel. During the concurrency fuzzing, SCT employs a concurrency-aware mutation strategy (③) to generate multi-threaded syscall programs. As the core component of SCT, we propose a novel CCT scheduler based on temporal isolation scheduling to control the concurrent execution via eBPF (④). During the execution of a multi-threaded test input (⑤) in the concurrency fuzzing phase, SCT delegates the scheduling control of target tasks to the CCT scheduler. When a scheduling point (e.g. a CPU yield) is reached, SCT enforces a schedule decision and selects the next thread to run from the set of runnable candidates according to specific CCT scheduling strategies. SCT repeats this scheduling process until all concurrent threads are finished. In doing so, SCT can effectively serial-

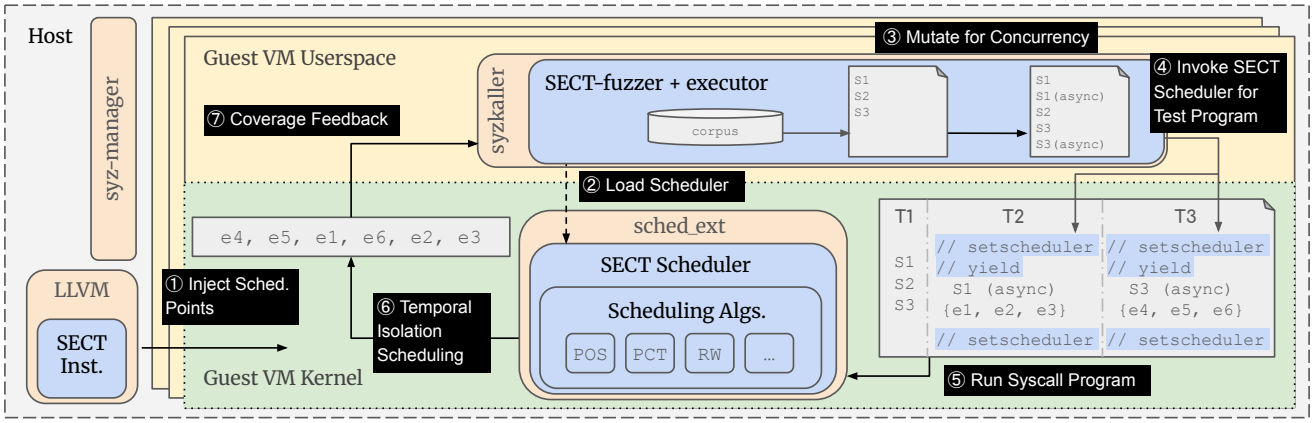

Figure 1: Overview of SECT (components of SECT in blue).

ize and control (⑥) multi-threaded executions into temporal isolation, thereby increasing the likelihood of triggering bugs that manifest due to rare interleavings. SECT always monitors whether the kernel under test crashes or terminates with any errors, in addition to collecting code coverage feedback (⑦) to drive exploration of the input space.

Through steps ① - ⑦, there are four main components in the design of SECT, as shown in blue in Figure 1: the CCT scheduler, the programmable scheduling strategies, the preemption-safe instrumentation, and the fuzzing loop. In the following section, we provide a detailed description of each design component.

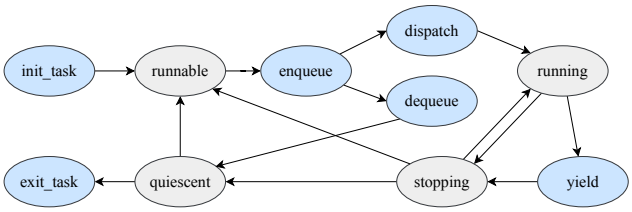

Figure 2: Task lifecycle within the `sched_ext` infrastructure. Grey ovals represent the states that a task can be in, and blue ovals represent actions that a task can undergo.

4.1 Temporal Isolation Scheduling

We define *temporal isolation* as a scheduling paradigm in which (a) tasks can be partitioned into domains, with each domain's scheduling decisions remaining isolated from the other and (b) only one task from each domain is running at a time, serializing the execution. The goal of the SECT scheduler is to explore the interleaving of events on different kernel tasks to expose concurrency vulnerabilities via temporal isolation scheduling. In this section, we present the core logic of the SECT scheduler and how it realizes temporal isolation scheduling using eBPF.

Task Lifecycle We begin by briefly reviewing the task lifecycle model of the `sched_ext` scheduling class illustrated in Figure 2. In kernel terminology, both threads and processes are uniformly represented and scheduled as tasks, and we use these terms interchangeably. Each task is created in the `init_task` state and enqueued in a run queue and later dispatched to a CPU with an assigned time budget. A task transitions to a quiescent state either by exhausting its time slice or voluntarily yielding the CPU due to blocking operations [4].

Temporal Isolation Serialization. Rather than treating scheduling merely as a performance-oriented resource allocation mechanism, SECT adopts a *scheduler-as-an-explorer* perspective, in which the scheduler actively drives the exploration of concurrent behaviors. To enable this exploration, we enforce *temporal isolation serialization* among tasks under test. Concretely, the SECT scheduler serializes task execution by dispatching a runnable task only after ensuring that all other eligible tasks are either quiescent or safely re-enqueued. Once a task is dispatched, the scheduler will not dispatch any additional tasks (even if idle cores are available) until the running task either blocks or yields its CPU. The scheduler iteratively repeats this process until all tasks involved in the test have terminated.

Under this temporal isolation serialization paradigm, concurrent execution proceeds as a sequence of kernel events rather than an imprecise, uncontrolled set of parallel events. As a result, temporal isolation serves as an exploration primitive that grants programmable control over concurrency, enabling the scheduler to systematically expose memory races and synchronization interactions that would otherwise remain implicit under uncontrolled parallel execution.

Partitioning of Scheduling Domains. While the temporal isolation serialization facilitates controlled exploration within a single test, enforcing it globally across the entire system would result in severe underutilization of system resources. To support high-throughput testing, SECT partitions tasks from concurrent test instances into isolated scheduling domains.

Algorithm 1: Temporal Isolation Scheduling

```
   Input: set of active tasks T
   Input: current task t
   Output: serialized execution trace S
1  initialize_scheduling_algorithm()
2  while |T| > 0 do
3      d ← get_tasks_in_domain(t)
4      if is_sleeping(t) then
5          t.enabled ← False
6      else if is_exiting(t) then
7          T ← T \ t
8      else
9          t.enabled ← True
10         T ← update_task_weights(T, d, t)
11     update_fairness_bound(t)
12     T' ← get_enabled_tasks(d, T)
13     t' ← pick_highest_weight_task(T')
14     next_t ← dispatch_and_run(t')
15     S ← append(S,t'.next_event)
16     t ← next_task()
```

Specifically, the scheduler of SECT identifies these domains by inspecting attributes within the task structure and leverages these attributes (e.g., command name or group identifiers), to cluster tasks belonging to the same test instance into a unified domain. At each scheduling event, the scheduler first resolves the task's domain membership and then enforces the temporal isolation serialization discipline strictly within that domain. This mechanism allows multiple test instances to execute in parallel without interference. In this way, SECT decouples the scheduling decisions for each domain, ensuring that the controlled scheduling are applied locally without impeding parallel execution.

Algorithm 1 presents the core scheduling procedure of SECT scheduler. When the scheduler is invoked, either due to timer interrupt or the running task transitioning to the quiescent state, the algorithm locates the corresponding scheduling domain (line 3), removes the current task if it is exiting (line 6) or disables the task from scheduling if it is blocked (line 4). Otherwise, the weight of each task in the domain (line 10) are updated based on CCT strategies that is discussed in Section 4.2. Finally, the scheduler selects an enabled task from the domain with the highest weight and dispatches the task to run. This architecture enables flexible task scheduling to support diverse CCT strategies, addressing **C3**: only the weight-assignment function needs to be defined, without requiring modifications to the overall scheduling lifecycle.

Shadow Scheduling Metadata. While SECT leverages the `sched_ext` feature to interact with the native kernel, this infrastructure is designed for uncontrolled scheduling and optimized for latency and fairness, rather than exploration. This leads to several impedance mismatches for CCT, which SECT overcomes. For example, while the Linux scheduling infrastructure provides a native priority queue interface for task dispatch [20], it prohibits modifying task attributes (e.g., priority) once a task has been enqueued. This restriction fundamentally limits its applicability for CCT, as many scheduling strategies (such as PCT [19] and POS [70]) require dynamic, event-driven priority updates based on runtime states. This dynamism is crucial for reacting to program behaviors as they unfold. For example, when a running task is observed to race with a paused task, the scheduler may need to elevate the priority of the paused task to realize a specific interleaving [70]. Therefore, the native dispatch queue is insufficient to support exploration-oriented CCT workloads, directly relating to **C1**.

To overcome this limitation, SECT bypasses the native kernel data structures and instead maintains a shadow scheduling metadata layer implemented using eBPF maps [3]. This shadow structure stores per-task metadata, including dynamically adjustable priority values, scheduling eligibility, domain membership, and other runtime attributes required by CCT scheduling strategies. In addition, priority computation and updates are delegated to the scheduling strategies themselves rather than being embedded in the dispatch mechanism. As later discussed in Section 4.2, this decoupling of scheduling logic from dispatch logic enables different CCT strategies to define, refine, and evolve their own policies with minimal overhead, addressing **C3**.

4.2 Programmable Scheduling Strategies

A scheduling strategy in SECT represents a specific algorithm by defining a weight-assignment function invoked at each scheduling point. SECT exposes this extensible and programmable scheduling interface for diverse algorithms which promote exploration of the interleaving space, rather than latency or fairness. By interacting with the shadow scheduling metadata, these algorithms can react to dynamic program states in real-time, overcoming the inherent rigidity of native kernel scheduling interfaces. To demonstrate the versatility of SECT, we have implemented four representative exploration-oriented algorithms: Random Walk, Random Priority, POS [70], PCT [19]. SECT is also extensible to support other more sophisticated scheduling strategies.

Random Walk. Random Walk (RW) is our most naive scheduling policy. At each scheduling point, we select the next task to run from the pool of enqueued tasks uniformly at random. The algorithm assigns the same weight to all runnable tasks. Notably, RW provides probabilistic fairness — tasks are unlikely to be delayed for an extended period of time. It also results in many preemptive context switches, as each scheduling decision has a $\frac{N-1}{N}$ probability of switching tasks if there are N enqueued tasks available to run.

Random Priority. For the Random Priority (RP) policy, at each scheduling point, we assign the most recently executed task a priority. The algorithm then picks the runnable task

with the highest weight to execute next. This approach differs from RW in that the weights of most of the tasks persist across scheduling points. As a result, RP is prone to starvation, in that a task with low weight will likely not be selected by the algorithm, and thus stick with its low priority for subsequent scheduling decisions.

Partial Order Sampling. In Partial Order Sampling (POS), each task is assigned a random weight, and the weight of the most recently executed task is reset at each scheduling point, similar to the RW policy. POS further re-assigns the weight of any task whose event interferes with the most recent event of the previously scheduled task [70]. Here, interference refers to accessing the same heap object, with at least one access being destructive (e.g., a `load` and `store` instruction accessing the same address). This adjustment enables POS to select different orderings of interfering events with equal probability. Consequently, POS is the only semantically aware concurrency testing algorithm in SECT, requiring source code information to compute interference. Similar to RP, POS can also suffer from starvation.

PCT. Probabilistic Concurrency Testing (PCT) [19] is a well-known algorithm developed for concurrency testing that provides a lower bound on the probability of detecting *low depth* concurrency bugs – parameterized by d. It is an extremely *unfair* algorithm: It only enforces $d-1$ context switches in an entire execution of thousands, or even millions of events, where d is often recommended to be set as low as three [64]. In PCT, d intervals are sampled uniformly from possible locations in the program execution. Tasks are assigned static weights at the beginning of execution. At each of the d intervals, the task with the highest weight is de-prioritized, giving the second highest weighted task primacy. To compute the interval sizes, PCT requires an estimate of the total number of events in a given execution, usually approximated by counting the events of a "trial run".

4.3 Scheduling Point Injection

Under temporal isolation scheduling, once a task is dispatched by SECT, it continues execution until it exhausts its assigned time slice or voluntarily yields its CPU, at which point the scheduler re-gains control [4]. Although kernels already contain some existing scheduling points, such as blocking operations and slice expires, these points are primarily designed to ensure fairness and responsiveness rather than to expose concurrency bugs. Indeed, many kernel concurrency vulnerabilities arise from subtle re-orderings around shared-memory accesses and synchronization operations. Consequently, relying solely on existing native scheduling points can leave a large portion of the concurrency-relevant interleaving space unexplored. To bridge this semantic gap, SECT introduces additional preemptions comprehensively throughout the kernel, elevating these events to explicit scheduling points. As a result, it aligns scheduling control with the semantic locations more likely to trigger concurrency bugs.

Identifying Preemption. In principle, any operation that allows tasks to synchronize or otherwise communicate with each other could lead to buggy behavior if reordered. Therefore SECT targets two classes of runtime events that mediate the vast majority of inter-thread interactions: shared-memory accesses and explicit synchronization operations. Specifically, we identify and instrument memory access operations at the LLVM IR level [39] (such as `load` and `store` instructions) which are not marked as thread-local by the compiler. In addition to shared-memory accesses, we instrument a set of high-level synchronization primitives which we manually identified in the kernel source code, including spin locks, semaphores, RCU synchronization, and other widely used kernel concurrency constructs. By injecting scheduling points *only* at semantically motivated locations, SECT avoids unnecessary overhead of context switching at points which do not involve communication between tasks and thus cannot trigger additional concurrency bugs. While we believe this set of additional preemptions to be highly comprehensive and sufficient for finding most concurrency bugs in practice, we discuss the completeness limitations of SECT in Section 8. We note that potential incompleteness arising from missing synchronization points can be mitigated by extending the instrumentation to inject scheduling points at such locations.

To control runtime overhead introduced by the additional instrumentation of the kernel, SECT supports configurable probabilistic triggering of the scheduler at injected points. By default, the scheduler is invoked with a 10% probability at each instrumented location during fuzzing. This design exposes fine-grained control over the trade-off between exploration depth and execution throughput without sacrificing probabilistic completeness (Section 8), and can be readily adapted to different scheduling algorithms or kernel targets.

Preemption Safety. After identifying a set of candidate scheduling points, SECT instruments these locations with an explicit call to the scheduler of SECT. However, scheduling points in kernel code must adhere to the implicit invariants governing where voluntary context switches are permitted. For example, preempting execution within interrupt contexts, non-preemptible regions, or critical sections protected by spin locks may lead to kernel instability or deadlocks. To preserve execution fidelity, SECT first performs a runtime preemption admissibility check that evaluates the safety of a potential scheduling point prior to intervention. Instead of attempting to formalize all non-preemptible regions, SECT leverages kernel native synchronization assertions, such as predicates derived from `might_sleep()` in the Linux kernel. These predicates encapsulate developer-defined safety invariants, indicating whether the current context permits re-scheduling. Once a candidate point satisfies the admissibility check at *runtime*, SECT triggers a voluntary CPU yield. This yield serves as the entry point for scheduler intervention, allowing SECT to suspend the current task and perform a context switch

according to the active exploration strategy.

Notably, while generally conservative, the admissibility checks are not a *guarantee* of safety. However, we find they work extremely well in practice to avoid false-positive bug reports due to unsafe preemptions (discussed in Section 8). As demonstrated in our evaluation of an ablative variant of SECT without additional scheduling point instrumentation (Section 6), this mechanism generally maintains the invariants of the kernel under test while enabling more fine-grained schedule exploration .

4.4 Fuzzing Loop

To enable continuous test generation and execution, SECT incorporates the CCT-oriented scheduler into the fuzzing process. SECT alternates between two phases: sequential fuzzing and concurrency fuzzing, each of which is described below.

Sequential Fuzzing. The primary goal of sequential fuzzing is to explore a broader range of execution paths in the kernel. In this phase, SECT delegates completely to Syzkallers' existing input-fuzzing strategy [28], which we summarize below. Syzkaller relies on syscall specifications written in Syzlang [8] to guide input generation and mutation towards valid syscall sequences. During test generation, Syzkaller selects syscalls at random from candidates, appends them to the end of the current sequence, and instantiates their parameters. The resulting sequences are then executed successively in their original order by the kernel under test. If a sequence triggers previously unexplored code branches, it is considered promising and will be preserved in the corpus for further mutation. During test mutation, Syzkaller picks up a seed from the corpus and performs one of mutation strategies in an attempt to uncover new coverage. Syzkaller employs traditional mutation strategies, such as inserting additional syscalls, removing existing syscalls, and modifying the values of the parameters of syscalls. While for now, SECT merely leverages the default exploration strategy of Syzkaller, in the future, SECT can integrate more heavy-weight input generation techniques [27, 65, 68]. However, because sequential input fuzzing is orthogonal to the core contributions of SECT, we leave such enhancement to future work.

Concurrency Fuzzing. Existing fuzzers such as Syzkaller usually use ad-hoc approaches to increase the chances of interesting concurrent behaviors without controlling the schedule. SECT proposes a custom concurrency mutation strategy tailored to the controlled temporal isolation scheduling provided by the SECT scheduler. Algorithm 2 presents the overall procedure of the mutation strategy. The input to this algorithm is test cases consisting of syscall sequences p, generated via the sequential fuzzing phase. For a given seed input p, SECT begins by randomly picking up N syscalls as S from the sequence to execute concurrently. The subset S represents the targeted syscalls for scheduling control. SECT then replicates each syscall in p to the new sequence P. During this process, if a syscall belongs to the selected subset S, it is also duplicated in the new sequence. This duplication strategy introduces opportunities for concurrent behavior while maintaining the structural and semantic correctness of the original sequence.

Algorithm 2: Concurrency Mutation

```
Input: syscall sequence p
Input: number of syscalls to execute in parallel N
Output: mutated syscall sequence P
1  P ← empty-list
2  S ← pick_parallel_syscalls(N, p)
3  if S ≠ ∅ then
4      for i ← 0 to |p| − 1 do
5          P ← append(P, p[i])
6          if p[i] ∈ S then
7              s ← duplicate(p[i])
8              s' ← annotate_async(s)
9              P ← append(P, s')
10 return P
```

To facilitate concurrent exploration, SECT leverages an existing annotation mechanism provided by Syzkaller to designate specific syscalls in P as asynchronous. At runtime, a specialized executor derived from Syzkaller interprets these annotations and instantiates each annotated syscall with its own dedicated thread. These threads are bound to the `SCHED_EXT` scheduling class [4], effectively delegating their execution control to SECT scheduler. In contrast, unannotated syscalls run under the native scheduler, incurring no additional scheduling overhead. To establish a consistent starting state for interleaving exploration, the executor injects a synchronization barrier, aligning the entry points of all participating tasks before initiating controlled scheduling. Once the barrier is released, the temporal isolation scheduling described in Section 4.1 begins and explores the interleaving space of targeted syscalls for multiple iterations.

We exemplify the mutation and execution process in Figure 1. For a given program consisting of syscalls $\{S_1, S_2, S_3\}$, SECT may mutate it to mark S_1 and S_3 as asynchronous (③). Each of these syscalls has some preemption events $\{e_1, e_2, e_3\}$ and $\{e_4, e_5, e_6\}$ which are separated by scheduling points injected in Section 4.3. During the execution phase (⑤), SECT cross-interleaves the events of the asynchronous syscalls S_1 and S_3 (⑥) to expose potential race conditions, while the execution of other syscalls remains unaffected.

5 Implementation

We implement the scheduler of SECT as a C/BPF program. In total, the SECT scheduler framework is a 790 line of code (LoC) as C/BPF programs, with another 96 LoC in C for the command line utility. Each scheduling algorithm is implemented as individual C/BPF files, which are included by

the SECT scheduler at compile time. In total, the algorithms together add another 345 lines of code in aggregate to the SECT. The SECT fuzzing loop builds on top of Syzkaller, changing 358 LoC. Lastly, we implement the instrumentation as a LLVM pass in 409 LoC in C++ . The helper function that invokes the scheduler is injected by the pass. It consists of an additional 23 LoC in C, including the necessary preemption admissibility check. We show the simplified snippet of the admissibility check in Listing 2, including those extracted from kernel native assertions in `might_sleep()`.

The LLVM pass instruments load and store instructions that access cross-thread variables, which are identified in LLVM IR by their synchronization scope (e.g., `syncScope::System`). For semantic scheduling algorithms such as POS, the instrumentation sends a message to the scheduler before yielding the CPU. This message encodes the target memory address and whether the operation is destructive (i.e., a store instruction), using scheduling-policy-specific fields in the task struct.

```
if (current->policy == SCHED_EXT
  && get_current_state() == TASK_RUNNING
  && !current->non_block_count
  && !is_idle_task(current)
  && preempt_count() == 0
  && !irqs_disabled()
  && rcu_preempt_depth() == 0 ) {
      // ...
      yield();
}
```

Listing 2: The simplified snippet of the admissibility check for additional scheduling points inject by SECT.

6 Evaluation

In this section, we evaluate SECT on recent `sched_ext`-capable Linux kernels in comparison with the state-of-the-art fuzzers.

We seek to answer the following research questions:

- **RQ1:** How does SECT perform in branch coverage?
- **RQ2:** How much overhead is introduced by SECT?
- **RQ3:** How does SECT perform in bug detection and bug triggering?
- **RQ4:** Does SECT achieve our goals of maintainability and extensibility?

6.1 Experiment Setup

Kernel under Test. All experiments in this section use Linux 6.13-rc4 to keep the kernel and baseline ports fixed across fuzzers. Our separate real-world bug-finding campaigns run on multiple Linux kernels that support `sched_ext`. The kernels are compiled using the same compilation configurations recommended by syzbot [7], with `KCOV` enabled for collecting coverage feedback and `KASAN` enabled for detecting memory safety violations that arise from concurrency bugs, such as concurrent Use-After Free bugs. We omit experiments with other test oracles, including KCSAN, because these are generally orthognal — test-drivers like SECT generate many different executions and test oracles like KCSAN determine whether a given execution manifested a bug. Indeed, many concurrency bugs such as the JFS bug (Figure 1) are not data-races and thus cannot be detected by KCSAN, but rather are an issue with a particular ordering of two critical sections.

Baselines. We compare SECT with Snowboard [27], SegFuzz [35], and Syzkaller [28]. Snowboard and SegFuzz are state-of-the-art concurrency kernel fuzzers, while Syzkaller is the most widely used sequential-oriented kernel fuzzer.

We manually ported these fuzzers to the recent kernel version. Unfortunately, we found Snowboard to have two severe practical limitations: (1) it required a prohibitively large amount of resources for pre-processing, including roughly 100TB of disk space and (2) the pre-processing analysis of potential memory conflicts by Snowboard does not support many newer kernel features, several of which are present in the reproducers for bugs in our benchmark. As such, we only compare against Snowboard's throughput and not its bug-finding ability. We evaluate three CCT algorithms in SECT: SECT_RW, SECT_POS, SECT_PCT as well as an ablative baseline, SECT-. The baseline SECT- consists of SECT_RW without any additional injected scheduling points (c.f. Section 4.3). We exclude the RP algorithm as POS is a refinement of it. Unless otherwise noted, SECT refers to the default algorithm of RW. In addition, all the fuzzers use the same seed corpus according to [54].

Experimental Parameters. For each fuzzing experiment, we conduct 10 trials of 48 hours. For a fair comparison, we configure Syzkaller, SegFuzz and SECT with the same resources, including 2 cores and 2 GB of memory for each VM. We ran 4 VMs each with 2 fuzzing processes for all experiments with SECT, Syzkaller and Snowboard. For SegFuzz, we ran with the same compute resources (CPU/RAM) but twice as many VM instances (to offset the limitation of 1 SegFuzz process per VM).

Hardware. All experiments were conducted on a 64-bit Ubuntu 22.04 LTS machine with Intel Xeon Platinum 8468V (96 physical cores) and 504 GB of RAM.

6.2 Code Coverage and Performance Overhead

To answer **RQ1**, we monitor the fuzzing process and record the branch coverage achieved by Syzkaller, SegFuzz and SECT. Figure 3 illustrates the comparison of branch coverage obtained by each fuzzer. SECT achieves 38% higher branch coverage than SegFuzz. In comparison to the sequential-oriented fuzzer Syzkaller, SECT and SegFuzz reach 88% and 63% of its branch coverage, respectively.

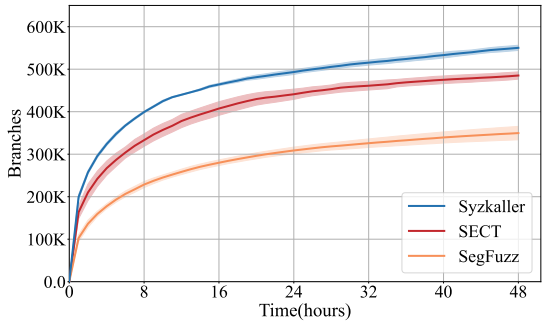

Figure 3: Code coverage achieved by Syzkaller, SegFuzz and SECT (N=10).

Table 2: Average fuzzing throughput and concurrent testcase-latency over 48 hours (N=10).

Type	Fuzzer	Exec. Time (ms/test)	Throughput (#tests)
Sequential Oriented	Syzkaller	89	8.3M
Concurrency Oriented	Snowboard	2498	-
	SegFuzz	228	0.38M
	SECT	99	4.4M

To analyze this reduction in coverage and respond to **RQ2**, we also examine the throughput and latency of each fuzzer, as detailed in Table 2. In terms of the latency per test-case, SECT's execution overhead for a given syscall program is only 11% more than that of the sequential fuzzer Syzkaller. Critically, this represents a 72% reduction in latency compared to existing concurrency fuzzers on average. We consulted the authors of Snowboard and they confirmed that a significant drop in performance was expected for their fuzzer as a result of the heavyweight control it imposes at the hypervisor level. In contrast, SECT 's kernel-native approach avoids these expensive mechanisms and cross-layer interventions. While SECT 's total throughput is approximately half that of Syzkaller, it vastly outperforms other concurrency fuzzers. SegFuzz achieves only 9% of SECT's throughput. A primary bottleneck for SegFuzz is how it enforces scheduling control: it is restricted to a single test instance per VM due to a reliance on a limited number of hardware breakpoints. In contrast, Syzkaller and SECT are able to run multiple parallel instances per VM, providing better scalability (**C1**).

We explain SECT's coverage reduction compared with Syzkaller with three factors. First, concurrency fuzzers allocate a portion of the computing resources to exploring thread interleavings, which generally contributes little to branch coverage. As a result, fewer resources are available for sequential fuzzing, leading to reduced coverage. Second, the CCT scheduling for tasks and the kernel instrumentation for injecting scheduling points incur additional overhead for kernel execution and thus reduces the throughput. Third, SECT has a higher chance to trigger concurrency issues (discussed in Section 6.3), which causes a costly VM reboot and also diminishes the throughput.

6.3 Scheduling Effectiveness

To answer **RQ3**, we construct a ground-truth benchmark consisting of known concurrency bugs in the Linux kernel. We collect these bugs by referencing prior work [29] or analyzing Linux commit messages in the past two years based on heuristic rules (e.g., `^(?=.syzbot)(?=.race).*`). We excluded entries that lacked an associated patch or PoC, manually ported the remaining bugs to our evaluation kernel, and verified their reproducibility. On the groundtruth, we evaluate two key capabilities: (1) the number of concurrency bugs discovered within a given time budget, which reflects the overall bug-finding capability of the fuzzers, and (2) the number of attempts required to reliably trigger a concurrency bug from a given sequential input, which highlights the effectiveness of the temporal isolation based CCT scheduler. Accordingly, we evaluate SECT's scheduling effectiveness in two settings: when integrated with fuzzing campaign to search the input and interleaving space (bug-finding) and when used for scheduling independently (bug-reproduction).

6.3.1 Results: Bug-Finding

In this first setting, we evaluate the efficacy of SECT in bug detection by conducting a systematic fuzzing campaign on the established benchmark. At the end of the time budget, we identify the unique bugs discovered and record their cumulative occurrences across all experimental trials. We present the results of this experiment in Table 3. During a 48-hour period, SECT successfully identify up to eight of the ten benchmark bugs, with the performance varying slightly depending on the specific exploration strategy employed. In contrast, the baseline fuzzers exhibit significantly lower detection capabilities. Syzkaller, SegFuzz and SECT- find only five, one and six bugs, respectively. Two bugs in the benchmark remain undiscovered by all fuzzers.

Based on these results, SECT demonstrates a superior capacity for uncovering concurrency bugs compared to Syzkaller and SegFuzz. While certain cases exhibit a higher variance in discovery frequency (e.g., bug #3, $p \leq 0.018$)[1], this is a side-effect of the intrinsic throughput trade-off in concurrency fuzzing. SECT allocates a significant portion of its execution budget to the exploration of the interleaving space. While this focus allows it to find more hard-to-reach concurrency bugs quickly, it does reduce the total volume of generated *inputs* which can reduce consistency when searching both the

[1]Using the chi-squared test [57] with $\alpha = 0.05$ for SECT _PCT and Syzkaller

Table 3: Known concurrency bugs found by SegFuzz, Syzkaller, SECT (N=20). Note that SECT_RW is default SECT.

ID	Vulnerability	SegFuzz	Syzkaller	SECT-	SECT_RW	SECT_POS	SECT_PCT
1	CVE-2023-31083 [48]	-	-	-	-	-	-
2	CVE-2024-42111 [49]	-	-	● (3)	● (8)	● (9)	●(10)
3	CVE-2024-44941 [50]	-	●(10)	●(12)	● (1)	● (2)	● (3)
4	CVE-2024-49903 [51]	-	●(13)	●(11)	● (7)	●(10)	● (9)
5	CVE-2024-50125 [52]	-	● (4)	● (5)	● (1)	● (2)	● (1)
6	CVE-2024-57900 [53]	-	-	-	● (1)	-	-
7	cb2239c1 [12]	-	-	-	-	-	-
8	61179292 [11]	-	-	-	● (1)	● (4)	● (1)
9	3b9bc84d [9]	●(15)	●(16)	●(18)	●(16)	●(19)	●(19)
10	88b1afbf [10]	-	●(15)	●(12)	●(12)	●(18)	●(15)
	Total	1	5	6	8	7	7

input and interleaving space. Indeed, as we will discuss in Section 6.3.2, the SECT scheduler is significantly more effective than the native scheduler at exposing buggy interleavings. Thus the variation in the consistency stems from differences in *throughput*. In other words, SECT pays a cost in terms of consistency of input-space fuzzing for increasing focus on the interleaving-space, though it is still overall a net benefit for concurrency bug-finding on our benchmark. Finally, SegFuzz significantly underperforms in the benchmark, finding only one bug in all trials. We directly consulted with the authors of SegFuzz to ensure an optimal configuration. We attribute this result to SegFuzz's inherently low execution throughput (Table 2).

Scheduling Point Ablation Study. To assess how the additional scheduling points impact SECT's effectiveness, we conduct an ablation study with SECT-, a variant of SECT that relies solely on the kernel's natural scheduling points without the additional preemption instrumentation. Without the additionally injected scheduling points, SECT- discovers fewer bugs overall than full SECT. However, even with the reduced scheduling granularity, SECT- is still able to uncover one bug (#2) that Syzkaller fails to detect in all trials. At the same time, the limited scheduling points significantly constrain its ability to expose bugs that require precise interleavings (e.g., bug #2 and #8). These results demonstrate the effectiveness of SECT- scheduler and highlight the benefits of injecting additional scheduling points to enable finer-grained exploration.

6.3.2 Results: Bug-Reproduction

In addition to the bug-finding setting, we further evaluate the effectiveness of SECT scheduler in exploring the thread interleaving space. Specifically, we feed the scheduler of SECT with the PoC syscall programs belonging to the benchmark, which are able to trigger the corresponding concurrency bugs. For each bug, we record the number of execution attempts required to induce a kernel crash. The native OS scheduler, utilized by existing fuzzers such as Syzkaller, serves as the baseline for this comparison. Bugs without an available syscall program are omitted in this experiment. We utilize Kaplan-Meier survival curves [38] to visualize the reproduction efficiency for each scheduler in Figure 4. These plots show the probability that a bug has *not been found yet* (y-axis) after a certain number of schedules have been explored (x-axis). Curves that exhibit a steeper decline and reach zero more rapidly indicate superior bug-triggering capabilities.

The results demonstrate that, while the number of schedules required to expose concurrency bugs varies significantly, the temporal isolation scheduler of SECT across all implemented algorithms can reproduce these bugs in consistently fewer attempts than the native OS scheduler. On average, SECT achieves an 11.4× speed-up in bug reproduction. This advantage is particularly evident in cases such as bugs #1 and #7. For instance, while the native OS scheduler requires up to 13,000 schedules to trigger the bug #1, SECT consistently exposes the vulnerability in fewer than 500 schedules. In addition, we find that the PCT algorithm has a worse performance than the RW algorithm in some cases, such as bug #1 and bug #7 ($p < 0.005$)[2], whereas the RW and POS algorithms exhibit comparable effectiveness on the benchmark. This discrepancy may be due to the tendency of the PCT algorithm to induce severe starvation of some tasks. While these algorithms are often effective for user-space program testing, kernel-level tasks may rely on stronger implicit fairness assumptions. Violating these assumptions may lead to redundant or unproductive error paths. These findings suggest that incorporating more sophisticated, fairness-aware scheduling algorithms [72] represents a promising direction for future work.

6.4 Qualitative Examination of SECT

To answer **RQ4**, we examine several aspects of SECT's implementation. While maintenance burden **(C2)** is difficult to quantify directly, a major source of incompatibility in prior work stems from modifications to the kernel source code.

[2]Using the log-rank test [42] with $\alpha = 0.05$

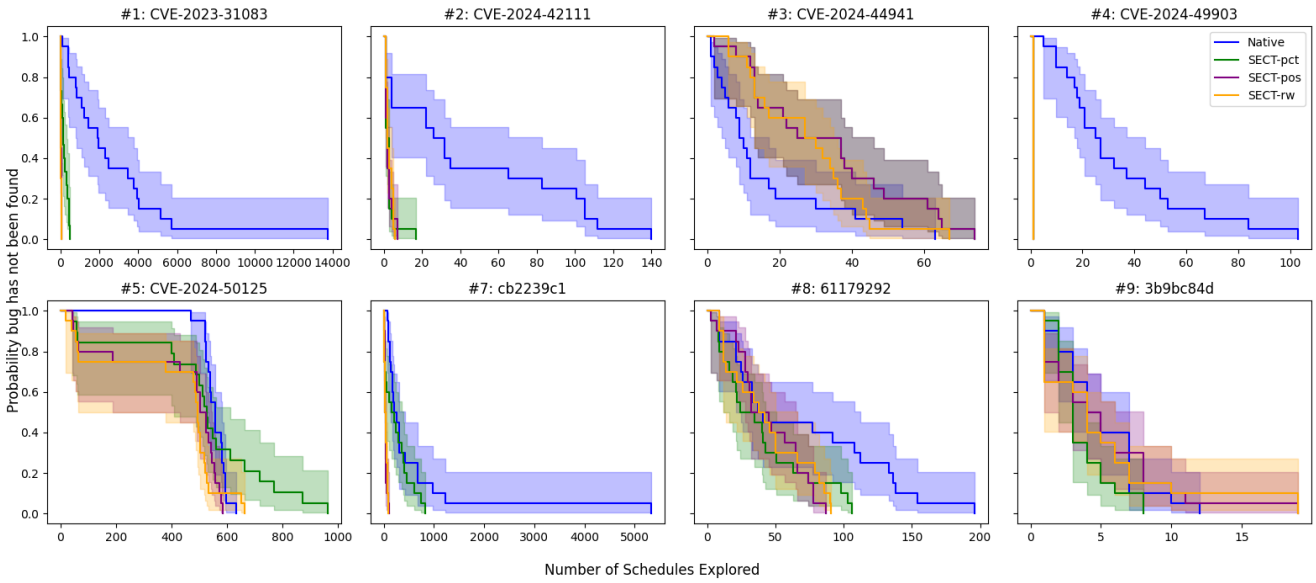

Figure 4: Cumulative probability of failing to detect the bug relative to the number of explored schedules. Lower values indicate higher efficiency. (N=20).

Table 4: LoC for four representative CCT scheduling algorithms implemented in SECT.

Algorithm	LoC.
PCT [19]	216
POS [70]	66
RW	40
RP	23

Such changes can result in merge conflicts when ported to newer kernel versions or lead to unexpected functionality failures as kernel evolves. For example, Krace [67] requires over 10k LoC of kernel code modifications. SegFuzz [35] not only introduces over 5k LoC changes to the VM guest kernel, but also requires additional invasive patches for the *host kernel* as well! In contrast, SECT leverages the compiler infrastructure for kernel instrumentation and adds only a single helper function to the kernel source, comprising fewer than 25 LoC. This is even fewer than Razzer (40 LoC changed) [34] which provides only limited scheduling control. As a result, SECT substantially reduces the likelihood of merge conflicts when adapting to newer kernel versions. This does not guarantee that SECT will remain compatible with all future releases of the Linux kernel without maintenance. However, `sched_ext` has remained stable for more than two years, and SECT avoids the invasive kernel-source patches that make prior tools difficult to port across kernel versions. Even if some APIs change, the migration cost is therefore expected to be substantially lower than for patch-heavy approaches.

For evaluating extensibility **(C3)**, LoC can serve as a meaningful proxy for the implementation complexity of new scheduling strategies. To demonstrate the modularity of SECT's strategies, Table 4 details the LoC required to implement each of the four representative CCT scheduling algorithms within the framework. Most of these scheduling algorithms can be implemented in less than 100 LoC, reflecting a relatively low barrier for extending with diverse exploration strategies. This indicates that the goal of extensibility (**C3**) has been achieved. The implementation of the PCT algorithm involves additional complexity due to the logic required to mitigate busy-wait loops, which are common in kernel code but lead to pathological behavior for highly unfair scheduling algorithms such as PCT.

7 Real-world Bug Finding

We evaluate SECT across multiple `sched_ext`-capable Linux kernels along with the development of `sched_ext`. Specifically, we conducted four independent 7-day fuzzing campaigns. Each campaign was provisioned with four VMs, each allocated 2 CPUs and 2GB of RAM. These campaigns produced roughly 500 raw crash reports. For all encountered crashes, we performed a systematic triage procedure consisting of: (1) searching on the internet and syzbot [7] to identify and discard duplicate reports; (2) minimizing the collected syscall programs and excluding non-concurrency bugs; (3) performing manual analysis to validate the issue. Most reports that did not become new bug reports were duplicates of known issues and the remaining discarded reports were

Table 5: Previously unknown concurrency-related bugs in Linux kernel discovered by SCT.

ID	Version	Bug Type	Location	Subsystem	Status
1	6.13-rc4	use-after-free read	sl_sync	driver/net/slip	confirmed
2	6.13-rc4	use-after-free read	netdev_walk_all_lower_dev	net/core	fixed†
3	6.13-rc4	null-ptr-deref	jfs_ioc_trim	fs/jfs	fixed
4	6.13-rc4	deadlock	nr_del_node	net/netrom	confirmed†
5	sched_ext	RCU anno missing	xa_get_order	kernel/mm	fixed
6	sched_ext	deadlock	pernet_ops_rwsem	kernel/sched	fixed
7	sched_ext	memory leak	scx_ops_enable	kernel/sched	fixed
8	sched_ext	deadlock	lockdep_assert_rq_held	kernel/sched	fixed

either non-reproducible or unrelated to concurrency. Through this process, we have identified eight previously unknown concurrency-related bugs in the Linux kernel, as detailed in Table 5. Among these bugs, five have already been fixed, and one has been confirmed by kernel developers. The remaining two bugs are currently under review. We present two representative case studies of these bugs that have been fixed or confirmed by developers, illustrating the practical effectiveness of our approach:

Case Study I: use-after-free in `sl_sync`. This bug is a concurrent use-after-free in the `sl_sync` function. It is caused by a kernel thread improperly freeing a Serial Line IP Device (`slip_dev`) without first acquiring an `rtnl_lock`. The use occurs when attempting to clean up existing open SLIP channels. Average number of trials to reproduce this bug appears in Figure 5. SCT reproduces this bug in an order of magnitude fewer schedules than the native OS scheduler.

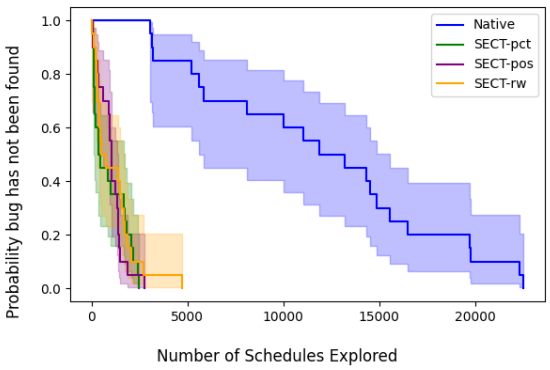

Figure 5: Kaplan-Meier survival analysis for the confirmed use-after-free bug in `sl_sync` identified by SCT (N=20).

Case Study II: deadlock in `pernet_ops_rwsem`. Our development of SCT also lead to improvements in the `sched_ext` framework. Figure 6 shows a deadlock bug discovered by SCT in the `sched_ext` subsystem of a fork of the Linux kernel [31]. This deadlock could be triggered if a task takes `cpu_hotplug_lock` between `scx_pre_fork()` and `scx_post_fork()`. One possible unsafe locking scenario is depicted: CPU0 is currently holding the lock

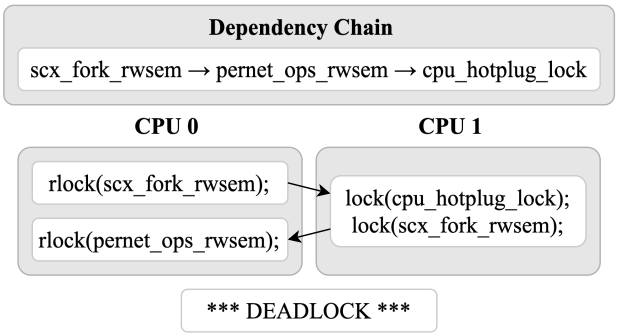

Figure 6: The root cause of a deadlock bug in the `sched_ext` class identified by SCT.

`pernet_ops_rwsem`, and requires the `cpu_hotplug_lock`. However, CPU1 already holds `cpu_hotplug_lock`, and requires the `pernet_ops_rwsem` lock that CPU0 holds. Since both CPUs are waiting on each other and have no way to preempt the resource, a deadlock scenario is reached.

8 Limitations

Soundness. Without additional instrumentation, SCT is sound in that it will not emit schedules or report false-positive kernel bugs that are not possible to reproduce without SCT. However, when using additional instrumentation (c.f. Section 4.3) the runtime admissibility checks extracted from `might_sleep()` do not *guarantee* the safety of additionally injected preemptions. As a result, if the safety checks incorrectly report an unsafe context switch to be safe, it is possible for SCT to report false positive bugs. However, when evaluating SCT on recent versions of the Linux kernel, we manually triaged more than 500 crash reports and did not see any reports that we could trace to false positives from unsafe preemptions. Among the non-reported crashes which we manually investigated, 87% were duplicates of previously reported crashes on `syzbot`[3], roughly 6% were not reproducible, and roughly

[3] https://syzkaller.appspot.com/upstream
† Concurrently discovered and reported by syzbot

7% were not related to concurrency. To avoid reporting false-positive bugs, we did verify bugs with the native scheduler when possible (e.g. Figure 5), though in many cases this can be prohibitively expensive due to the weak exploratory power of the native scheduler. However, we note that attempting to reproduce a bug on an uninstrumented kernel is not a definitive false-positive test: failure to reproduce may simply mean the buggy interleaving is too rare to occur naturally, not that the report is spurious. When we weren't able to reproduce bugs outside of SECT, we collaborated with kernel developers to ensure bugs were not false positives, though this did not end up being the case for any of our found bugs.

Completeness. The CCT approach of schedule serialization and injection of scheduling points is theoretically complete in that it can produce any sequentially consistent schedule which is realizable by the native OS scheduler. However, SECT makes three practical compromises which may make some behaviors impossible to observe. For completeness, a scheduling point must be inserted and triggered at runtime between any two visible (non-commuting across threads) operations, so that those operations can be reordered by the scheduler. (1) Our LLVM instrumentation pass may miss some of these visible operations in the kernel due to inline assembly code, synchronization primitives we may have failed to identify as instrumentation targets, and core modules or IRQ contexts omitted from instrumentation. Additionally, (2) the admissibility checks (c.f. Section 4.3) may spuriously mark some safe preemption contexts as unsafe at runtime, resulting in some instrumented scheduling points not being triggered during execution. Lastly, (3) SECT uses a compile-time configurable sampling-rate for each additional scheduling point, allowing users to increase performance at the cost of scheduling granularity. Sampling may cause SECT to miss bugs in a particular execution, due to not sampling a key scheduling point. However, it notably does not affect SECT's *probabilistic completeness* [72] in that the missed bug would still be observable with some probability in future executions where the key scheduling point *is* sampled.

9 Conclusion

SECT is the first kernel-native concurrency fuzzing framework built upon the *scheduler-as-an-explorer* paradigm. It is designed to overcome the architectural bottlenecks of existing kernel concurrency fuzzers by providing fine-grained, lightweight, and extensible scheduling control for CCT. To achieve this, SECT proposes the temporal isolation scheduling with programmable policies, the safe preemption instrumentation mechanism, and the two-phase input-and-concurrency fuzzing workflow. Our evaluation demonstrates that SECT achieves substantial improvements in branch coverage, efficiency, and bug discovery effectiveness. We anticipate that the design of SECT's components will facilitate developers and researchers to leverage it to increase the fundamental robustness of OS kernels.

10 Ethical Considerations

Stakeholders The list of possible stakeholders for this work is large, including not just maintainers and developers of the Linux kernel but also direct and indirect users and vendors of Linux as well. For example, historically, high-severity concurrency vulnerabilities such as the Dirty COW bug [4] have affected commercial vendors of Linux distributions, such as Google and RedHat, consumers of those and other distributions, such as Android phone users, enterprises which deploy their applications on Linux servers, such as Amazon. These stakeholders risk damage to their reputations, expensive availability outages, and/or costly remediations in the case of vendors. For consumers and enterprise users, concurrency vulnerabilities can also lead to personal or commercial data theft and denial-of-service or even active exploitation of affected devices. Given that the Linux kernel is part of the most widely deployed operating system in the world (Android) and is the leading operating system family used by enterprises for server deployments, vulnerabilities and bugs in the kernel can affect innumerable number of users, businesses and customers.

Potential Negative Outcomes and Mitigations During the research process, those most directly affected are the kernel maintainers to whom we reported bugs. Our reports take up their time and effort to process, which can be a burden if the reports are of low quality or spurious – which we attempted to minimize by not submitting duplicate reports. Post-publication, we hope SECT can positively affect downstream users by improving kernel security and reliability and also positively affect maintainers and researchers by providing a more robust substrate with which to test the kernel for concurrency bugs (and reproduce them).

It is possible that in reporting these bugs, a malicious actor could use our reports for exploitation of consume or vendor software before the bugs are patched. However, there are several mitigating circumstances: (1) the `sched_ext` fork was not widely deployed at the time of reporting (2) the Linux kernel maintainers are trusted – if a maintainer has already been compromised, the security of that module or subsystem is completely compromised as well and (3) we attempted to provide patches and assisted maintainers in debugging and fixing these bugs as quickly as possible, limiting unpatched exposure time. In the case of the JFS bug, our patch was accepted and merged into the Linux kernel. As far as we know, none of the bugs we have reported were actively exploited, but this nonetheless remains a risk of our research. There is an additional risk in that we did not escalate all issues to

[4] https://nvd.nist.gov/vuln/detail/CVE-2016-5195

the security team by default. However, we opted to defer to maintainers in this regard as they are experts in their respective modules.

After publication, we will make SECT openly available to assist developers and maintainers to find and debug concurrency issues more reliably. However, this comes with the risk that SECT could be used by malicious actors to find and exploit concurrency bugs more easily. We believe the benefits of open-sourcing SECT outweigh this risk. Lastly, there is a risk of users of SECT inadvertently exposing themselves to exploitation via concurrency bugs. While we believe this risk to be low due to the difficulty of actively exploiting concurrency bugs, we address this by providing instructions for setting up SECT only within a virtualized environment (despite the scheduler being possible to run natively) and documenting risks involved with using it.

Bug Disclosure Process We disclosed bugs directly to kernel maintainers and developers to facilitate swift remediation. In the case of the 4 `sched_ext` bugs, we reported them in the developer Slack channel. In the case of the mainline kernel bugs, we reported 2 of them to relevant trusted maintainers via emails identified by the `get_maintainers.pl` script. We inquired directly to these maintainers if bug reports required additional escalation to the kernel security team, but in all cases they did not indicate an escalation was necessary. The remaining 2 bugs were found and reported by syzbot before we had a chance to submit a bug report.

Bug Impacts All of the `sched_ext` bugs have been fixed by developers after our initial reports. We believe the extent of the impact to have been reliability disruptions for the users and developers of this kernel in all cases.

Our patch for the JFS bug was accepted and merged into the mainline Linux kernel. We attempted to submit patches for the `slip_open` bug, but we weren't able to find a correct patch due to our lack of familiarity with the subsystem. The `netdev_walk` bug was patched by a maintainer soon after it was reported. Lastly, the deadlock in `nr_del_node` had a valid patch submitted by a maintainer, but this patch has been postponed in favor of a larger refactoring to clean up an error-prone locking pattern.

It is difficult to gauge the impact of the Linux kernel bugs beyond non-deterministic reliability issues for users. However, because of the broad scope of Linux's user-base, even sporadic issues can affect a large number of stakeholders.

Decision to Publish We began this project with this goal in mind: to make concurrency debugging and testing of kernel code more tractable. We knew that many of the concurrency bugs currently submitted are done so without reliable reproducers at all. Indeed we believe this to be a contributing factor to why the JFS bug was only partially fixed.

We decided to submit for publication after we were confident in its effectiveness from our experiments, as this meant it could be deployed by others to actively enhance the security and reliability of kernels.

Open Science

To facilitate future research on OS kernel fuzzing, we open-source SECT on https://anonymous.4open.science/r/sect-3616b2b2. Additionally, we will actively participate in artifact evaluation processes to ensure the robustness and reproducibility of our findings. We hope that our contributions will foster further advancements in kernel security research.

References

[1] Bpf type format. https://docs.kernel.org/bpf/btf.html. Accessed: 2026-02-01.

[2] CFS scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html. Accessed: 2025-08-07.

[3] ebpf maps. https://docs.ebpf.io/linux/concepts/maps/.

[4] Extensible scheduler class. https://docs.kernel.org/scheduler/sched-ext.html. Accessed: 2025-08-07.

[5] jcstress. https://github.com/openjdk/jcstress.

[6] libbpf: a c-based library containing a bpf loader. https://github.com/libbpf. Accessed: 2026-02-01.

[7] syzbot dashboard.

[8] Syzlang: pseudo-formal grammar of syscall description. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[9] Linux commit log 3b9bc84d. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3b9bc84d311104906d2b4995a9a02d7b7ddab2db, 2022.

[10] Linux commit log 88b1afbf. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=88b1afbf0f6b221f6c5bb66cc80cd3b38d696687, 2022.

[11] Linux commit log 61179292. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=611792920925fb088ddccbe2783c7f92fdfb6b64, 2023.

[12] Linux commit log cb2239c1. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cb2239c198ad9fbd5aced22cf93e45562da781eb, 2023.

[13] Linux commit log d6c1b3. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d6c1b3599b2feb5c7291f5ac3a36e5fa7cedb234, 2024.

[14] 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):4, April 2017.

[15] Johannes Bechberger and Jake Hillion. Concurrency Testing using Custom Linux Schedulers.

[16] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Noise makers need to know where to be silent – producing schedules that find bugs. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 458–465, 2006.

[17] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[18] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, et al. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 836–850, 2021.

[19] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3):167–178, March 2010.

[20] Jonathan Corbet. The extensible scheduler class. https://lwn.net/Articles/922405/, February 2023.

[21] Jules Dejaeghere, Bolaji Gbadamosi, Tobias Pulls, and Florentin Rochet. Comparing security in ebpf and webassembly. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 35–41, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. Industrial-strength controlled concurrency testing for c# programs with coyote. In *TACAS*, April 2023.

[23] Daniel Etiemble. Technologies and computing paradigms: Beyond moore's law? *arXiv preprint arXiv:2206.03201*, 2022.

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[25] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. ACTOR: Action-Guided kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5003–5020, Anaheim, CA, August 2023. USENIX Association.

[26] 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.

[27] 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, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Google. Syzkaller. https://github.com/google/syzkaller, 2015.

[29] Tianshuo Han, Xiaorui Gong, and Jian Liu. CARDSHARK: Understanding and stablizing linux kernel concurrency bugs against the odds. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6203–6218, Philadelphia, PA, August 2024. USENIX Association.

[30] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *44rd IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 22-25, 2023*. IEEE, 2023.

[31] Tejun Heo, David Vernet, Josh Don, and Barret Rhoden. sched_ext. https://github.com/sched-ext/sched_ext, 2023.

[32] Daniel Hodges. Scheduling at scale: eBPF schedulers with Sched_ext. Dublin, October 2024. USENIX Association.

[33] Zunchen Huang, Shengjian Guo, Meng Wu, and Chao Wang. Understanding concurrency vulnerabilities in linux kernel, 2022.

[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, 2019.

[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, 2023.

[36] Xu Jiacheng, Zhang Xuhong, Ji Shouling, Tian Yuan, Zhao Binbin, Wang Qinying, Cheng Peng, and Chen Jiming. Mock: Optimizing kernel fuzzing mutation with context-aware dependency. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[37] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. 01 2022.

[38] Edward L Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, 53(282):457–481, 1958.

[39] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[40] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[41] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. MINER: A hybrid Data-Driven approach for REST API fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4517–4534, Anaheim, CA, August 2023. USENIX Association.

[42] Nathan Mantel. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemother Rep*, 50(3):163–170, 1966.

[43] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of linux ebpf subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 87–92, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Madan Musuvathi and Shaz Qadeer. Chess: Systematic stress testing of concurrent software. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation*, pages 15–16, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[45] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 267–280, USA, 2008. USENIX Association.

[46] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Softw. Engg.*, 27(7), December 2022.

[47] NVD. Cve-2016-5195. https://nvd.nist.gov/vuln/detail/cve-2016-5195, 2016.

[48] NVD. CVE-2023-31083. https://nvd.nist.gov/vuln/detail/cve-2023-31083, 2023.

[49] NVD. CVE-2024-42111. https://nvd.nist.gov/vuln/detail/cve-2023-42111, 2024.

[50] NVD. CVE-2024-44941. https://nvd.nist.gov/vuln/detail/cve-2024-44941, 2024.

[51] NVD. CVE-2024-49903. https://nvd.nist.gov/vuln/detail/cve-2023-49903, 2024.

[52] NVD. CVE-2024-50125. https://nvd.nist.gov/vuln/detail/cve-2023-50125, 2024.

[53] NVD. CVE-2024-57900. https://nvd.nist.gov/vuln/detail/cve-2023-57900, 2024.

[54] Palash B. Oswal. *Improving Linux Kernel Fuzzing*. PhD thesis, 2023.

[55] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[56] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, 2020.

[57] Karl Pearson. X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, July 1900.

[58] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[59] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 323–332, 2007.

[60] Scott D. Stoller. Testing concurrent java programs using randomized scheduling. *Electronic Notes in Theoretical Computer Science*, 70(4):142–157, 2002. RV'02, Runtime Verification 2002 (FLoC Satellite Event).

[61] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 344–358, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Herb Sutter. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[63] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005.

[64] Paul Thomson, Alastair F Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing (TOPC)*, 2(4):1–37, 2016.

[65] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758. USENIX Association, August 2021.

[66] Dylan Wolff, Shi Zheng, Gregory Duck, Umang Mathur, and Abhik Roychoudhury. Greybox fuzzing for concurrency testing. 2024.

[67] 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, 2020.

[68] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 560–573, New York, NY, USA, 2025. Association for Computing Machinery.

[69] Kenichi Yasukata and Kenta Ishiguro. Developing process scheduling policies in user space with common os features. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '24, page 38–44, New York, NY, USA, 2024. Association for Computing Machinery.

[70] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 317–335. Springer, 2018.

[71] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, Boston, MA, August 2022. USENIX Association.

[72] Huan Zhao, Dylan Wolff, Umang Mathur, and Abhik Roychoudhury. Selectively uniform concurrency testing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 1003–1019, 2025.