CUSTOM KERNEL SCHEDULER WITH EBPF FOR CONCURRENCY FUZZING

by

HAN XING YI
(B.Eng. (Hons.), SUTD)

A THESIS SUBMITTED FOR THE DEGREE OF

MASTER OF COMPUTING (INFOCOMM SECURITY SPECIALISATION)

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2024

Supervisor:
Professor Abhik ROYCHOUDHURY

Examiners:
Assistant Professor Jialin LI

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

[illegible]

HAN Xing Yi

15 April 2024

Acknowledgments

I would like to express my sincere gratitude to Professor Abhik Roychoudhury for his unwavering guidance, invaluable insights, and constant support throughout the journey of my thesis. His expertise, encouragement, and patience have been instrumental in shaping my research and academic growth, allowing me to explore my interests daringly.

I extend my heartfelt appreciation to Assistant Professor Jialin Li for his role as my examiner and for providing valuable feedback, which significantly contributed to the refinement of my work.

Special thanks are due to Dylan J. Wolff for his collaboration, mentorship, and shared dedication to advancing our research objectives. His expertise and encouragement have been indispensable in navigating the complexities of my thesis. His innate ability to problem-solve and keen analytical skills have significantly contributed to overcoming the numerous challenges encountered throughout the research process.

I am indebted to the TSS research group for their camaraderie, intellectual exchange, and willingness to lend a helping hand whenever needed. Their collective wisdom and support have been invaluable in overcoming challenges and finding innovative solutions.

I am deeply grateful to my friends and family for their unwavering love, encouragement, and understanding throughout this journey. Their belief in me has been a constant source of motivation, and their presence has provided much-needed solace during challenging times.

Finally, I would like to express my appreciation to all those who have contributed to this thesis in ways seen and unseen. Your support has been instrumental in bringing this work to fruition.

Thank you all for being an integral part of this significant milestone in my academic and personal development.

Contents

Abstract

Custom Kernel Scheduler with eBPF
for Concurrency Fuzzing

by

HAN Xing Yi

Master of Computing (Infocomm Security Specialisation)

National University of Singapore

Concurrency bugs are notoriously difficult to detect and reproduce, often requiring a specific combination of thread interleavings and inputs. Existing concurrency fuzzing works are heavy-weight, usually requiring specialised hypervisors to control thread execution, and are limited by heavy instrumentation or scalability issues.

This paper introduces `SECT`, a lightweight approach to concurrency fuzzing by leveraging eBPF to implement a custom scheduler. `SECT` enables highly deterministic control over thread execution by serialising thread schedules, and uses concurrency testing algorithms such as PCT to explore the interleaving space. `SECT`'s integration with Syzkaller enables it to explore both the input and schedule spaces to detect concurrency bugs effectively.

We demonstrate `SECT`'s capability to identify concurrency issues with minimal performance overhead. Our approach not only reveals several bugs but also showcases its potential to schedule both userspace and kernel programs. `SECT`'s compatibility with existing data race detectors and ease of deployment opens new avenues for future concurrency fuzzing works.

Keywords: Concurrency Fuzzing, Concurrency Testing, eBPF, Linux Kernel, Thread Scheduling, Bug Detection.

List of Tables

List of Figures

Chapter 1

Introduction

In 21st-century computing, concurrency has become a cornerstone of efficient system design. The end of Moore's Law has led to a shift towards increasingly parallel and distributed software systems [17, 69, 70]. Leveraging concurrency allows applications to handle complex, data-intensive tasks more effectively. From real-time analytics [13, 56, 57] and distributed database systems [4, 41, 53], to the current AI boom [20, 21, 33, 77], concurrency has become essential to satisfy the demand for speed in modern-day applications.

This is especially so for low-level, performance-sensitive systems such as the Linux kernel [44]. Operating system kernels must handle multiple user-space processes, device interrupts, and delayed code execution via workqueues and tasklets [10]. To ensure correctness, these developers must perform fine-grained locking through a large number of diverse synchronisation primitives. These include more traditional mechanisms such as spin-locks [47], reader/writer semaphores [46, 55] and memory barriers [45], as well as more esoteric optimistic synchronisation primitives such as Read-Copy-Update (RCU) [75], sequence locks [63], and lock-free data structures [39, 48, 68] and algorithms [5, 8]. While these primitives have allowed developers to write a scalable and performant kernel that is the core of the most widely used operating system in the world, the complexity of concurrency mechanisms in the Linux kernel contributes to a large number of concurrency bugs [26]; Indeed, automated fuzzing tools such as syzbot have already discovered hundreds of concurrency bugs in released versions of the Linux kernel [71].

Many of these concurrency bugs also have severe consequences due to the nature of kernel software. Deadlocks can cause kernel panics or system crashes, reducing system availability. Data-races, the most common concurrency errors, may result in data corruption that could lead to security vulnerabilities. One prominent example is the Dirty COW bug [12], where a race condition exploited a flaw in the handling of the copy-on-write mechanism in the kernel's memory management system. This allowed an attacker to gain write access to *read-only* memory mappings, enabling local privilege escalation. Thus, finding concurrency bugs is essential in ensuring a safe and reliable kernel.

Concurrency bugs are also often difficult to discover and fix, representing a higher proportion of latent bugs compared to their sequential counterparts [1]. This difficulty stems from the non-deterministic behaviour of concurrent operations, where certain bugs are only triggered with two important ingredients: (i) a specific schedule, and (ii) specific inputs. Finding these bugs, even for sequentially consistent programs, involves exploring the thread interleaving space which can be exponential in the number of instructions for a single test. The complexity is further compounded in the Linux kernel due to its vast code base [40] and its support for diverse hardware configurations [66].

In response to these challenges, the field has seen the development of dynamic race detection and prediction tools, as well as deadlock detection techniques, that have proven successful across a range of contexts. For user-space programs, tools like FastTrack [18] and ThreadSanitizer [64] are widely deployed in both research and commercial applications, and employ lockset analysis and partial-order-based techniques to detect races in polynomial time. However, in the Linux kernel, a race detector using similar techniques has proved difficult to implement efficiently given the large number of synchronisation primitives [38]. Even if such tools are available, detecting potential races from an execution trace is NP-Hard in general [51] and many concurrency bugs, such as atomicity violations, are not detectable with trace-based dynamic analysis alone.

Thus, in practice, developers often turn to stress-testing [3, 28, 52, 65] to uncover concurrency bugs. While subjecting the system to intensive workloads can sometimes increase the probability of triggering synchronisation issues or deadlocks, it requires manual effort to generate handwritten test suites. This is impractical given the vast code base of the Linux kernel [40]. Existing coverage-guided fuzzing tools [23, 43, 67], can automatically navigate the input search space to reveal hard-to-find bugs in uncommon execution paths. However, these fuzzing tools do not handle exploration of the *thread interleaving* space and thus are limited in their ability to uncover concurrency bugs.

As a complementary approach, concurrency testing techniques have emerged to search the space of these thread interleavings by proactively controlling the schedule. Concurrency testing algorithms such as PCT [6] expose data races by running a program repeatedly through a randomised scheduler to explore various schedules. Concurrency testing frameworks such as Snowboard [22, 29] utilise PCT to generate schedules and enforce these schedules by fully controlling thread execution, but also perform prior static analysis to identify potential data race targets which helps to narrow the search space. Some tools [30, 78] further incorporate testing frameworks into a fuzzing pipeline, proposing metrics that would guide fuzzing towards *inputs* likely to exercise interesting thread interleavings. However, these existing approaches for concurrency testing generally require heavy-weight custom hypervisors [19, 29, 30], have only very weak control over the schedule [78], incur high performance overhead [22], and/or have other severe limitations (c.f. Chapter 6).

In this paper, we propose an alternative, lightweight approach to testing for concurrency bugs in kernel software using eBPF. As the core of our approach, we introduce a new scheduler — the `sched_ext` concurrency tester (`SECT`). Our scheduler is an eBPF program, built on top of the `sched_ext` infrastructure [25], that serialises thread execution, exercising *full* control over the threads of specified programs by only allowing one thread to run at a time. We have implemented several scheduling algorithms in `SECT` for concurrency testing, and `SECT` can be extended with additional algorithms in the future. To attain more fine-grained exploration, we additionally include a mechanism for dynamically inserting additional preemptions at key events during execution via kprobes [37] and tracepoints [73].

Crucially, our scheduler can be loaded at runtime within seconds, and does not require *any* modification of the host kernel or hypervisor, even with this additional instrumentation. It can also be readily combined with existing kernel bug-detection and fuzzing tools such as KCSAN [36], KASAN [35] and Syzkaller [23]. We additionally extend Syzkaller [23] — a state-of-the-art fuzzer for kernel software — to more effectively generate system call sequences that exercise concurrent behaviour. Together, Syzkaller and `SECT` can be integrated into a fuzzing pipeline that simultaneously searches *both* the input space and the schedule space for concurrency bugs.

We evaluated `SECT` by looking for bugs in the latest[1] version of the Linux kernel in parallel with its development. Over the course of two months, we found several real-world kernel bugs, some of which have already been accepted and patched by developers. We also conducted a series of controlled experiments to understand the performance overhead of `SECT`, its level of deterministic control and its bug-finding effectiveness. Concretely, the research questions that our work aims to explore are:

RQ1. *What is the bug-finding effectiveness of our customised scheduler in a simple concurrency-fuzzing approach for kernel code?*

We compare `SECT`'s integration with Syzkaller against the base Syzkaller (without our modifications) and other configurations. We quantify the coverage obtained and the number of unique bugs found in the v6.8-rc6 Linux kernel during 24-hour trials.

RQ2. *What is the performance overhead incurred by serialising the schedule with eBPF?*

We create a set of programs containing sequences of system calls and evaluate the time it takes for `SECT` to execute the program to completion against running the program natively on multi-threaded and single-threaded instances. We also compare against another `sched_ext` scheduler, scx_qmap, as well as an existing state-of-the-art concurrency testing tool, Snowboard [22].

[1] v6.8-rc6

RQ3. *What impact do different scheduling algorithms have on bug-finding effectiveness?*

We compare the bug-finding ability of several randomized scheduling algorithms implemented in `SECT` on a widely-used user-space benchmarks from the PERIOD paper [74].

RQ4. *How deterministic is our serialised scheduler?*

Determinism is useful for bug reproduction, especially in concurrent operations. Since the scheduling algorithms employed by `SECT` are randomised algorithms, it is important to understand how deterministic `SECT` is on the same program given the same starting seed.

In addition, we identify the limitations of our current work and provide guidelines for future work. In summary, we make the following contributions:

- **Serialised scheduler.** We create and make available the first serialising scheduler utilising eBPF, `SECT`, which is suitable for lightweight concurrency testing of both kernel space and user-space software.
- **Integration with Syzkaller.** We modify a popular kernel fuzzer, Syzkaller, to fuzz for concurrency bugs with `SECT`.
- **Evaluation.** We evaluate `SECT`'s bug-finding ability and performance overhead relative to prior work [22] and native execution. We also assess the effectiveness of various concurrency testing algorithms implemented in `SECT`.

Chapter 2

Background & Motivation

In this chapter, we first take a look at how concurrency bugs are triggered due to the specific scheduling of two different threads through some examples. We then go on to look at the approaches of existing state-of-the-art concurrency fuzzers and testers to exercise control over the specific scheduling of threads, and notice that most of these solutions are heavy-weight and difficult to apply efficiently in practice. Next, we show how eBPF has become an important tool for kernel observability in recent years, and how it can be used to insert custom schedulers into the kernel.

2.1 Triggering Concurrency Bugs

Concurrency bugs can generally be classified into two main categories: deadlock bugs and non-deadlock bugs. Deadlock bugs occur when two or more threads hold resources and each thread waits for another to release their resource, leading to a standstill where none of the threads can proceed and causes the system to become unresponsive. Figure 2.1 shows a deadlock bug discovered by `SECT` in the `sched_ext` subsystem of a fork of the Linux kernel [25]. 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: CPU 0 is currently holding the lock `pernet_ops_rwsem`, and requires the `cpu_hotplug_lock`. However, CPU 1 already holds `cpu_hotplug_lock`, and requires the `pernet_ops_rwsem` lock that CPU 0 holds. Since both CPUs are

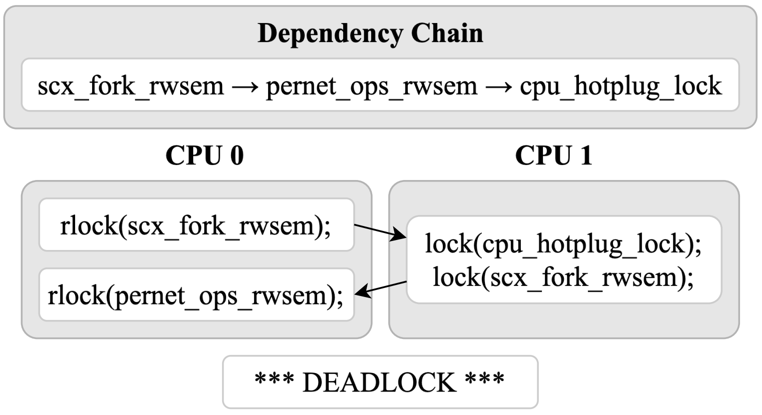
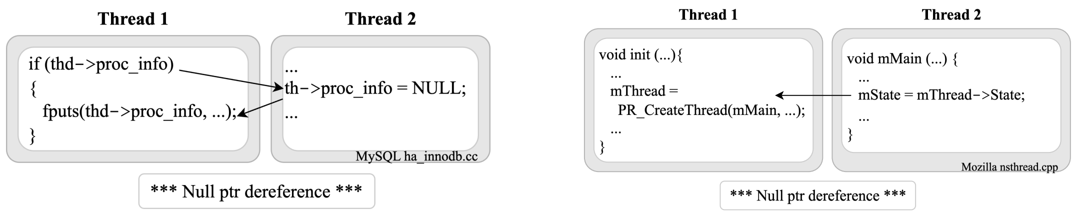


Figure 2.1: Deadlock bug in `sched_ext` found by `SECT`.

waiting on each other and have no way to preempt the resource, a deadlock scenario is reached.

On the other hand, non-deadlock bugs, which are more common and varied, include data races, atomicity violations, and order violations. Data races happen when two concurrent operations access the same variable without proper synchronisation and at least one of them is a write. Atomicity violations occur when a sequence of operations that should be executed atomically is interrupted, leading to inconsistent states. Order violations arise when the execution order of operations deviates from the expected sequence, potentially causing erratic behaviour or system crashes.

Figure 2.2 shows the two of the most frequent bug patterns observed in non-deadlock concurrency bugs [49]. For 2.2a, the variable `thd->proc_info` is checked in the if-statement to be non-null by Thread 1. Before Thread 1 is able to execute the statement that uses the variable, the OS scheduler schedules Thread 2, which preempts Thread 1 and sets the variable to null. This would then lead to a null pointer dereference when Thread 1 resumes its execution to write to the variable. This interleaving pattern usually results in a time-of-check to time-of-use (TOCTOU) race. For 2.2b, the code assumes that Thread 1 will always be run before Thread 2, which may not be the case. The OS scheduler may choose Thread 2 to run before Thread 1, which means that the variable `mThread` has not yet been created, causing a null pointer dereference.



```
if (thd->proc_info)
{
  fputs(thd->proc_info, ...);
}
```

```
...
th->proc_info = NULL;
...
```

```
void init (...){
  ...
  mThread =
    PR_CreateThread(mMain, ...);
  ...
}
```

```
void mMain (...) {
  ...
  mState = mThread->State;
  ...
}
```

(a) Atomicity violation **(b)** Order violation

Figure 2.2: Examples of non-deadlock concurrency bugs [49]

For each of these examples, the program must be executed with a specific interleaving of threads for the bug to manifest. During normal execution, the interleaving is determined by the OS scheduler, which determines which thread is executed next based on a particular strategy. For the Linux kernel, most processes are scheduled by the Completely Fair Scheduler (CFS), which tries to allocate CPU time to the processes as evenly as possible [7]. However, as its primary goal is fairness (rather than schedule exploration), the OS scheduler may only trigger a concurrency bug with very low probability, possibly only under particular workloads or on specific hardware. Thus, the ability to manipulate the scheduling behaviour can be a powerful strategy to find and reproduce concurrency bugs.

2.2 State-of-the-Art Kernel Concurrency Fuzzers

Since its development in 2015, Syzkaller [23] has gained traction in the fuzzing community and is arguably the most popular kernel fuzzer today. It has identified over 5,000 bugs in the upstream Linux kernel since its inception and inspired several research works [11, 24, 42, 54, 59, 72]. Syzkaller is designed primarily to uncover memory-related bugs and typically generates inputs meant for sequential execution; Of the bugs found by Syzkaller, only roughly 10% are concurrency-related issues. Nevertheless, Syzkaller does provide limited support for concurrency bug-finding via two simple, ad-hoc strategies of doubly executing a program and duplicating certain system calls, which have shown effectiveness in discovering some shallow concurrency bugs [71]. However, their strategies have *no control* over the thread interleavings, and this greatly restricts the fuzzer's ability to explore the scheduling space to discover deeper bugs.

To close this gap, researchers have developed several other fuzzers specifically tailored towards "concurrency fuzzing". Krace [78] introduces a new coverage metric, alias coverage, which is implemented in their own fuzzing architecture to guide fuzzing towards executing new and uncommon interleavings. To scale to handle kernel background threads, Krace makes a trade-off and adopts a *weak* form of scheduling control by injecting random delays at memory access points. This presents two major shortcomings: 1) delay injection alone is likely to miss many uncommon thread interleavings during fuzzing [31], and 2) Krace is unable to deterministically replay any execution.

Razzer [29] employs static analysis techniques on the kernel source code to identify potential race pair candidates, and guides fuzzing towards generating inputs containing these candidates. To control the schedule, Razzer modifies the hypervisor to include hypercalls that, when called by a kernel thread, would install hardware breakpoints on the vCPU. These hardware breakpoints are installed at memory reads and writes, which are pre-determined during static analysis. To trigger a race, Razzer does a single step on the specified vCPU before resuming execution on all vCPUs. Razzer also performs Virtual Machine Introspection (VMI) to ascertain the kernel thread context and ensures that the calling thread does not belong to an unrelated task. However, since Razzer can only enforce a certain thread interleaving at the breakpoints and is unable to control the interleavings of the threads before and after the breakpoints, we deem this as *intermediate* control.

SegFuzz [30], which builds on top of the Razzer architecture, improves on Razzer's limitation and achieves *full control* by enforcing a schedule sent by the fuzzer process via hypercalls. When a scheduling point is reached, the hypervisor suspends the running thread by making it call `cond_resched()` to yield the CPU. The next thread will then be scheduled to run. SKI [19] and Snowboard [22] use a similar method of sending hypercalls to suspend and resume the execution of vCPUs according to a schedule. 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, hence achieving full control over the thread interleaving. One major restriction of these works is that in exchange for full control, they do not scale well to large programs. For Razzer and SegFuzz, only a limited number of hardware breakpoints can be installed simultaneously, around 4 on Intel CPUs, thus restricting their ability to discover more complex concurrency bugs. For SKI and Snowboard, the one-to-one mapping of threads to vCPUs sets a physical constraint on the number of threads it can control to the underlying hardware. In addition, to infer liveness, complex heuristics are usually required to detect blocking operations in the system to avoid accidental deadlocks.

To sum up, the current state-of-the-art works in concurrency fuzzing must balance between the level of control over the thread interleaving and the scale of the program. Works that wish to scale to handle all threads in a system can only maintain *weak control* over the thread schedules. Works that exercise *full control* generally utilise a custom hypervisor to achieve instruction-level control over the thread schedules. However, the complexity of enforcing this control through a hypervisor leads to fragile implementations that either sacrifice performance or come with very strict limitations, or both. Moreover, they are mostly incompatible with existing bug detection technologies such as KCSAN [36], DataCollider [16] and many more [2, 32, 58, 60, 80]. These works also specifically target the kernel, making it difficult to reuse the same architecture for other user-space programs. Additionally, the high complexity of these tools coupled with the need to recompile a custom hypervisor with every change make them extremely difficult to adapt and modify with new algorithms or other improvements. As a result, these concurrency fuzzing tools are difficult to use and apply in practice.

Given these constraints, we aim to develop a lightweight alternative to controlling the thread schedule that has the following implementation goals (referred to as Goals G1-G3 in this paper):

G1 Easy to implement and test new scheduling algorithms efficiently, without requiring any modification of the hypervisor or large changes to compile-time instrumentation.

G2 Applicable for both user-space and kernel space processes, and scalable to large programs.

G3 Compatible with existing technologies like Syzkaller and KCSAN without excessive modifications.

2.3 Advantages of eBPF for Kernel Programming

To conceptualise a simpler approach for controlling thread schedules, we first make an observation: controlling threads at the machine-instruction level is often not necessary to trigger concurrency bugs; many times, controlling threads only at the source-statement level is sufficient. Hence, it is a reasonable assumption and trade-off to make that relaxing the scheduling granularity to whole statements would still be able to achieve the motive of manifesting concurrency bugs. Finally, rather than choosing to modify the hypervisor, we should instead attempt to directly control the source of non-determinism that causes concurrency bugs in the first place — the OS scheduler.

However, directly modifying the OS scheduler and injecting a new scheduling policy is not so straightforward. The Linux kernel is an incredibly complex, monolithic kernel [40], of which the scheduler is an integral part of ensuring that all processes are able to run smoothly. Coupled with the fact that kernel programming is inherently challenging, direct modification of the scheduler is not only error-prone, but also requires recompilation and installation of the new kernel on each change, which does not fulfil Goal **G1**. Any errors not caught at compile time could also lead to system crashes or kernel panics. Recognising these challenges, it becomes evident that a solution requiring minimal changes to existing infrastructure while providing control over thread scheduling would be highly advantageous.

Since it became fully available in the Linux kernel v4.4, the extended Berkeley Packet Filter (eBPF) technology has been at the forefront of kernel observability, networking and security. It offers a new way of kernel programming: developers can write programs that can be loaded at runtime and run in a safe, sandboxed environment within the Linux kernel, without directly modifying the kernel or loading a kernel module. To ensure that the inserted program does not crash the system, the eBPF verifier performs validation on the program's size, use of variables and memory accesses, and checks that it runs to completion. eBPF is a powerful technology that grants developers unprecedented access to extend the capabilities of the kernel.

One such capability is exemplified by `sched_ext` [25]. First released in Nov 2022, `sched_ext` brings about the concept of a "pluggable" scheduler for customisation, experimentation, and rapid deployment of scheduling strategies without the need for invasive kernel changes or recompilation. This dovetails well with our needs for a safe and simple approach to thread control (Goal **G1**). Since this "pluggable" scheduler essentially replaces the OS scheduler, it has the ability to schedule both user and kernel threads, allowing us to test both user-space and kernel space programs (Goal **G2**). Last but not least, `sched_ext` is built on top of the Linux kernel, which means that all functionalities of the Linux kernel are available to `sched_ext` (Goal **G3**).

Implementing `SECT` at the OS kernel level, rather than at the hypervisor level, also provides several advantages. Firstly, we have full and direct control over how the program threads are being scheduled, without having to perform sophisticated manoeuvres of suspending and resuming a vCPU. Next, we benefit from software abstractions that the kernel provides atop the hypervisor level, allowing us to directly access and modify thread information stored in the thread structure. This includes the state that the thread is currently in (e.g. blocked or running), the time slice that it has been executing for, and whether it is a kernel thread. Finally, we can perform scheduling at a lower latency, as we no longer need to incur the overhead of hypercalls for each preemption.

Chapter 3

Design & Implementation

In this chapter, we introduce the overall architecture of SECT, including its integration with the Syzkaller [23] fuzzing pipeline. We additionally go into further detail on the implementation of SECT within the eBPF sched_ext [25] infrastructure as well as the corresponding design considerations. We discuss how SECT can leverage additional instrumentation to obtain more fine-grained control over the thread interleaving schedule without introducing heavy-weight recompilation of the kernel or hypervisor. Finally, we explain in depth how concurrency fuzzing works in Syzkaller and the modifications that we have made to integrate SECT with Syzkaller.

3.1 Concurrency Fuzzing Architecture

The architecture of SECT for concurrency fuzzing includes its integration within the Syzkaller framework. For a full fuzzing pipeline, we insert SECT into the kernel under test to replace the OS scheduler during fuzzing, as shown in Figure 3.1. Components in blue correspond to existing Syzkaller modules, some of which we modified (c.f. § 3.3). Components in purple are the SECT scheduler. To begin a concurrency fuzzing campaign, we first start a syz-manager process on the host machine with a standard Syzkaller configuration file as input. Syz-manager spawns multiple virtual machine (VM) instances, each with its own syz-fuzzer process. We modified the Syzkaller fuzzer to generate sequences of system calls ("syscall programs") that are more suitable for manifesting

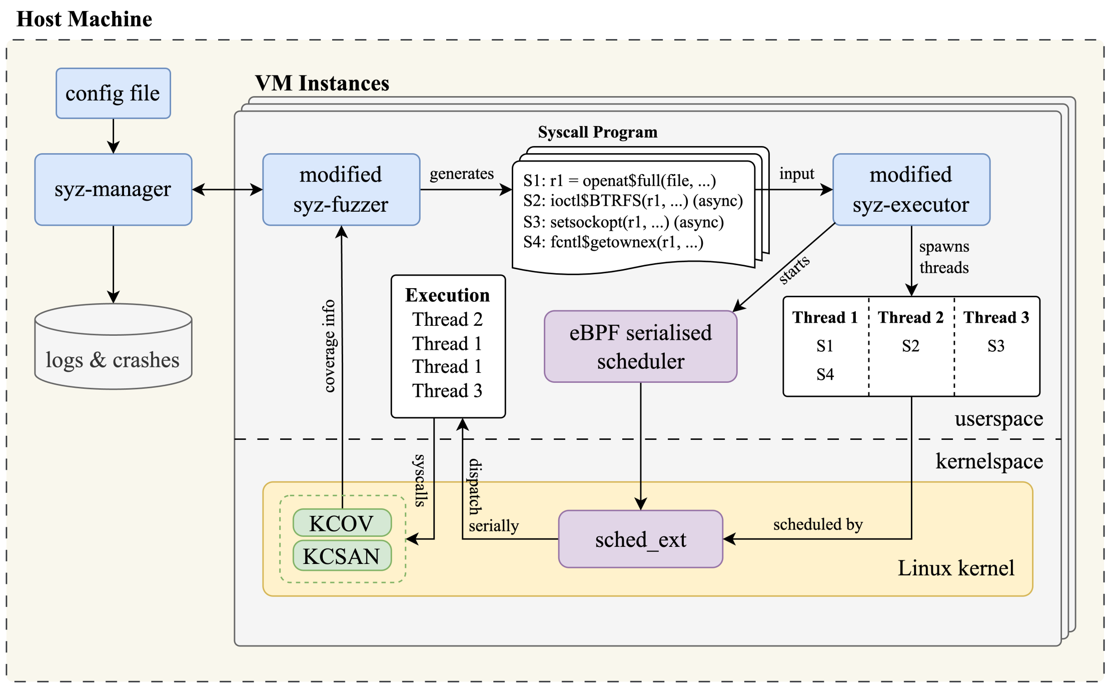


Figure 3.1: Overall architecture of SECT with Syzkaller.

concurrency bugs (c.f. § 3.3). These syscall programs are sent over to the syz-executor in the same guest VM instance, which has been modified to 1) insert SECT into the kernel, and 2) execute the syscall program with all spawned threads set to use the SECT scheduling policy.

These threads are then scheduled by SECT, which dispatches only one thread at a time (§ 3.2). By default, the threads yield the CPU after executing each system call. When a thread yields the CPU, or when the kernel natively calls the `schedule()` function, SECT picks the next thread to execute from all runnable tasks. This choice is determined by one of several scheduling algorithms implemented in SECT.

During execution, kernel sanitisers such as KCSAN [36] or KASAN [35] check for data races and other classes of bugs. Any bug reports produced by sanitisers are reported back to syz-manager, which stores the crash logs and restarts the VM instance to ensure the next fuzzing process starts on a clean slate. Otherwise, after each execution, code-coverage from KCOV [34] is collected by syz-fuzzer to guide subsequent syscall program generations and mutations.

3.2 Serialising Scheduler Implementation

3.2.1 The Kernel Scheduling Infrastructure

We build `SECT` upon the `sched_ext` [25] infrastructure, which introduces a new scheduling class that exports a full scheduling interface for eBPF programs. This scheduling interface mirrors that of other scheduling classes within the kernel, allowing any arbitrary scheduler to be implemented on top of their interface. A typical scheduler involves overseeing multiple dispatch queues, each containing runnable tasks awaiting CPU allocation [9]. The eBPF scheduler can be loaded and unloaded dynamically at runtime, allowing for flexibility (Goal **G1**) and easy integration with various tools that the kernel provides (Goal **G3**). The `sched_ext` infrastructure, coupled with the eBPF verifier, also puts in place safety features to ensure that system integrity is maintained even with a volatile scheduler, creating a safe environment for development and exploration.

We note that in kernel terminology, threads and processes (which are main threads) are both commonly referred to and scheduled as *tasks*, so these terms are used interchangeably. Any occasional distinction between threads and the main thread will be made explicit.

The main states of a task are:

- `runnable`: A task is runnable on a CPU when it is 1) waking up, 2) moving in from another CPU, or 3) being restored after temporarily being taken off the queue for an attribute change.
- `running`: A task is starting to run.
- `stopping`: A task is stopping execution. `running` and `stopping` come in one or more pairs during a thread's lifetime.
- `quiescent`: A task is not runnable on a CPU because it is either 1) going to sleep, 2) moving out to another CPU, 3) being temporarily taken off the queue for an attribute change.

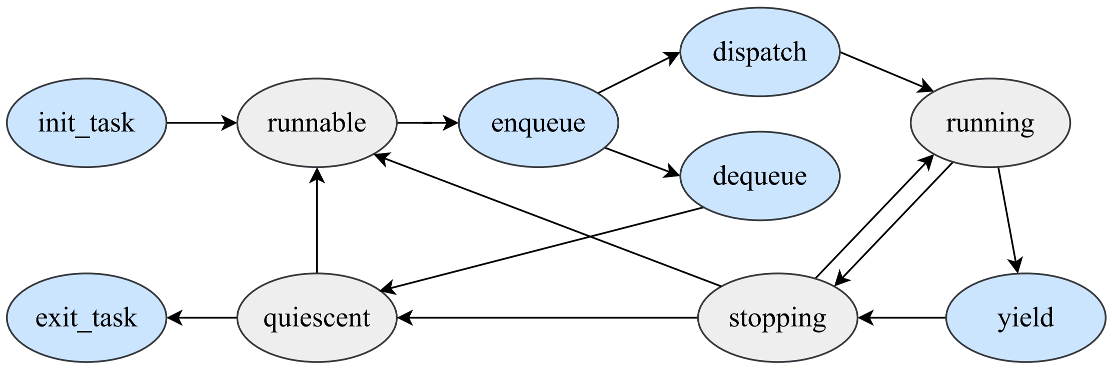


Figure 3.2: Thread lifecycle within the `sched_ext` infrastructure. Grey ovals represent the states that a thread can be in, and blue ovals represent actions that a thread can undergo.

The main actions that a thread in `SECT` will undergo are:

- `enqueue()`: Transfers control of the thread to the eBPF scheduler, and queues it at the scheduler in a dispatch queue.
- `dequeue()`: Takes back control of the thread from the eBPF scheduler, usually to update scheduling properties such as its priority.
- `dispatch()`: The eBPF scheduler can send a thread to the CPU for execution. This function is called when the CPU's local queue is empty.
- `yield()`: The thread has yielded the CPU from the program side, usually by the `sched_yield()` system call.

Figure 3.2 depicts the lifecycle of a task from its creation to its exit within the kernel, and the states that a thread can undergo within `sched_ext`. `init_task` marks the creation of a new thread. After creation, the thread transitions to the runnable state. At this point, the thread will call `enqueue()`, making it available to the scheduler. Here, the scheduler can make a decision on whether to dispatch this thread immediately, or store the thread in a dispatch queue for future dispatch. After a thread has been dispatched, it transitions to the running state, and goes through one or more running-stopping pairs until it either (i) voluntary yields, (ii) exhausted its time slice, or (iii) is blocked. A blocking thread will enter the quiescent state until it is unblocked and runnable again. Otherwise, it directly becomes runnable and is re-enqueued for further scheduling. Once the thread has finished execution, it enters the quiescent state before clean-up is performed and the thread exits.

The new scheduling class, `SCHED_EXT` (7), can be enabled via the kernel configuration file. Scheduling classes in the Linux kernel are ordered by priority, and `SCHED_EXT` is positioned in the priority hierarchy below the `SCHED_NORMAL` class, which uses the Completely Fair Scheduler (CFS), but above the `SCHED_IDLE` class, which is of the lowest scheduling priority. This design choice ensures that any scheduler operating under the `SCHED_EXT` class is unable to cause system-wide harm as usual processes scheduled by the CFS can still proceed normally.

Apart from the eBPF verifier ensuring the safety of the custom scheduler, `sched_ext` has three main safety triggers that would restore a system to its default scheduling behaviour to prevent system freezes or crashes. Firstly, any runtime error detected would immediately trigger a deregistration of the eBPF scheduler. Second, `sched_ext` implements a watchdog function that tracks each thread that is handed over to the custom scheduler. If the thread is not dispatched for at least thirty seconds, this is deemed as a "runnable task stall" and the watchdog process kills the custom scheduler. Threads then resume scheduling by the CFS. Third, invoking the SysRq key sequence `SysRq-S` would restore the usual scheduling behaviour regardless of the system state, allowing the system to recover from potential freezes.

For a thread to come under the control of `SECT`, the program that creates the threads needs to explicitly set the scheduling policy of the thread to the `SCHED_EXT` class. This can be done by instrumenting the program under test to use the `sched_setscheduler()` system call before spawning the threads in the `main` function. We preface that the `main` function will start off being scheduled by the CFS, and control will only be transferred to `SECT` after the `sched_setscheduler()` system call is invoked. Thus, a small limitation presents itself: prior to setting the scheduling policy, we have no control over the main thread. This is, fortunately, not the case for subsequent thread creations, as they will inherit the current scheduling properties of the main thread, hence giving SECT full control from thread creation.

3.2.2 Exercising Thread Control

To fully control the order of events during execution, SECT maintains that only one thread can be in the running state at any one time. It enforces this constraint by only dispatching a runnable task after verifying that all other threads eligible for scheduling, including the main thread, are either quiescent or successfully re-enqueued (and awaiting dispatch). After dispatching a given task, SECT will wait for the task to either become quiescent or be enqueued again before dispatching another task. SECT repeats this procedure until all threads under the specified scheduling policy have exited.

While the sched_ext infrastructure provides a priority queue interface for dispatching tasks, this interface does not allow for any updates to thread attributes after it has been added. Unfortunately, using this interface would preclude making dynamic scheduling decisions — once a task has been put into a dispatch queue, it priority is fixed and cannot be adjusted. However, even basic concurrency testing scheduling algorithms such as PCT [6] require priorities to be updated at runtime based on the dynamic state of the various tasks in the program (e.g. after a certain number of events have occurred).

Thus, we avoid using the dispatch queue for scheduling in SECT; Instead, we separately create and manage our own unique data-structure for each task that encapsulates its dynamic priority (as dictated by a scheduling algorithm), enqueued status, and other related meta-data. We choose to use `enqueue()` as our initial interaction point with the thread instead of the `runnable` state or `init_task`, since they might include threads that are outside of our custom scheduling policy and we would have to perform additional checks. When a thread transitions into a quiescent state, it is deemed as ineligible for scheduling and we remove its corresponding context. The thread is not considered for scheduling at the quiescent state, rather than in `exit_task`, because the thread might be blocking. Without removing its eligibility earlier, other threads cannot be dispatched until the blocking thread wakes up again, which may lead to the starvation of the runnable threads awaiting dispatch, leading to runnable task stalls. This custom architecture allows us to access any thread that has not yet been dispatched and update its attributes accordingly.

3.2.3 Inserting Additional Instrumentation

When `SECT` dispatches a task to run, that task will continue to run until it hits a blocking operation or its allotted time-slice expires. In the case of `SECT`, a hard interrupt will be sent to the CPU every three seconds to ensure that every task will eventually be rescheduled. However, preempting only on blocking operations may not be fine-grained enough to trigger many concurrency issues, which may depend on the relative ordering of function calls (c.f. Figure 2.2) or even individual load and store instructions from shared memory locations. To attain this fine-grained control, additional, voluntary preemptions can be injected into a system under test using *instrumentation*.

Instrumenting User-space Programs. User-space programs can be instrumented with the `sched_yield()` system call, which causes the calling thread to relinquish the CPU. These `sched_yield()` calls can be inserted manually into the source code of a client program before events of interest. For example, we use this instrumentation to ensure that the thread yields the CPU and returns to `SECT` before the execution of each system call in a program when testing with Syzkaller (§ 3.3).

Alternatively, we provide a script for the binary rewriting tool, e9patch [14], which instruments all of the individual memory reads and writes of a compiled program. In this case, prior to any memory access, the running thread will return to `SECT`'s control prior to the access, giving us fine-grained control over how threads are scheduled at critical locations.

Instrumenting the Kernel. To attain fine-grained scheduling with `SECT` in the kernel, we would want the CPU to reschedule just before an event of interest (e.g. memory access). To do so, we could manually inject calls to invoke `SECT` before instructions of interest at compile time. However, static instrumentation of the entire kernel to constantly trigger rescheduling would not only incur significant performance overhead [78], but is also likely to cause instability in the kernel itself. Furthermore, any changes to the set of events instrumented or instrumentation itself would require rebooting the system and recompiling the kernel, which contradicts Goal **G1** of a lightweight, easy-to-use scheduler.

Instead, we opt for dynamic instrumentation using existing mechanisms within the kernel. The eBPF infrastructure provides a wide array of hook points, including those for system calls, network events, and kernel tracepoints. Functions defined in eBPF programs, or *hooks* in short, are lightweight and can be dynamically loaded and executed whenever a hook point is reached. In particular, kprobes [37], primarily used as a debugging tool, provides an interface to hook onto virtually any exported kernel function, except itself. By prompting the thread to yield directly inside the hook, we can trigger a preemption every time the hook function is called during an execution. In addition to kprobes, we also use tracepoints [73], which are pre-determined hook points inserted into the kernel source code. Using tracepoints, we added hooks for memory-related functions `kmalloc()` and `kfree()`, as well as the locking-related functions `lock_acquire()` and `lock_release()`.

However, as the `sched_yield()` is a system call, it is not applicable in the kernel space. Hence, we cannot directly reuse the method for user-space programs in the kernel. Underneath the hood, `sched_yield()` invokes a call to the kernel function `schedule()`, which is responsible for performing context switches. Unfortunately, `schedule()` is also not exposed to eBPF programs because it could potentially cause unsafe behaviour. Hence `schedule()` cannot be called directly from our scheduler. Instead, we use a `sched_ext` function, `scx_bpf_kick_cpu()`, which triggers rescheduling on a busy CPU by sending a hard interrupt. While this does allow us to trigger a preemption, the preemption may not occur immediately before the next event, as the sent interrupt is racing with the executing program. We explore the impact of this limitation in practice in § 4.1.3. Additionally, `scx_bpf_kick_cpu()` can only be called from `sched_ext` functions and not kprobes or tracepoints. To circumvent this, we created a new BPF syscall, `bpf_schedule()`, that wraps `scx_bpf_kick_cpu()`. BPF system calls are callable from any function in an eBPF program, thus making it suitable for our use case. Although this requires a small modification to the kernel source code, we have raised the issue to `sched_ext` developers and they are actively working on allowing for manual preemptions natively within their framework.

Memory Operation Instrumentation While eBPF hooks allow us to trigger preemption before many important events, such as function calls or network events, these hooks are typically not available at the granularity of individual memory operations. To insert scheduling points dynamically close to memory accesses, we leverage existing instrumentation used by KCSAN, the kernel data race detector. KCSAN relies on compile-time instrumentation of the kernel to identify potentially dangerous memory accesses. During runtime, KCSAN samples from a set of memory accesses and installs watchpoints at these locations. To detect races, KCSAN delays a read instruction for a certain number of microseconds using the `udelay()` function, and checks back after the delay to see if the value was modified via a non-instrumented access. We use kprobes to hook to `udelay()`, and subsequently trigger a preemption within the kprobe hook function. This allows `SECT` to preempt at the same granularity as KCSAN without additional compile-time instrumentation and infrastructure. Unfortunately, `udelay()` is called ubiquitously when KCSAN is enabled. Triggering a context switch via a hard-interrupt on every `udelay()` call quickly becomes prohibitively expensive. To alleviate this issue, we adopt a similar sampling approach to KCSAN and only invoke the scheduler with a fixed probability whenever `udelay()` has been called.

3.2.4 Scheduling Algorithms

To this point, we have described how `SECT` serialises the execution of a subset of tasks using the `sched_ext` infrastructure and inserts additional preemptions for more fine-grained scheduling. However, the actual ordering of events executed by the program is determined by a scheduling algorithm. We have implemented three randomised scheduling algorithms in the initial implementation of SECT: PCT [6], Priority Walk and Random Walk. These three algorithms do not require any information nor communicate with the program under test, and adjust thread priorities solely based on the number of events observable by SECT. This design choice allows the scheduling algorithms to operate independently of the specific behaviours of the program under test, providing a flexible and generic approach to thread management.

PCT. Probabilistic Concurrency Testing (PCT) [6] is a popular and efficient algorithm developed for concurrency testing in user-space programs, and provides a lower bound on the probability of detecting a bug. The PCT algorithm assigns random priorities to each thread and schedules them from highest to lowest priority. Upon reaching one of $d - 1$ designated change points, PCT lowers the priority of the corresponding thread. For a single execution of a program with at most n threads and at most k events, the PCT algorithm finds a concurrency bug of depth d with probability at least $1/nk^{d-1}$. Here, depth d signifies the minimum number of scheduling constraints required to find a concurrency bug, while the number of events k typically refers to the number of preemptible events in the execution. In our implementation of PCT, we approximate the event count based on the number of enqueues in a single run of execution of a system call program. We also dynamically update this event count if we observe large differences in the number of enqueues, unlike the fixed event number assumed in PCT.

One limitation of PCT is that it only changes the priorities of the threads at designated change points. This may lead to starvation of the low-priority threads in the system. For instance, a high-priority thread in a spinlock might significantly increase the scheduler's observed event count. If spinlock is encountered after the last change point, the spinlock-engaged thread will continue to be prioritised by PCT forever, and all lower-priority threads will be unable to run. In SECT, this usually results in runnable task stalls and causes the scheduler to be deregistered by the safety mechanisms of the `sched_ext` infrastructure. To mitigate this issue, we adapt the PCT algorithm to be stratified over subsets of

Algorithm 1: Stratified PCT

```
Input: $n$ threads, $d - 1$ change_points, xorshift32 $rng$, thread_priorities
1: Function init_scheduler():
2:     max_events ← 0;
3:     current_events ← 0;
4:     shufflePriorities($rng$);
5:     chooseChangePoints($rng$);
6: Function enqueue():
7:     current_events += 1;
8:     if change_point then
9:         if current_events > max_events × (strata + 1) then
10:            strata += 1;
11:        update thread's priority to S32_MAX - ((strata+2) ×$n$) + $i + 1$
12:    mark thread as enqueued
13: Function dispatch():
14:    if threads enqueued = threads alive then
15:        dispatch highest priority thread
16: Function exit_task():
17:    if current_events > max_events then
18:        max_events = current_events;
19:    current_events = 0;
20:    strata = 0;
21:    shufflePriorities($rng$);
22:    chooseChangePoints($rng$);
```

k events, completely restarting the algorithm after the first k events have been observed. This means that a spin-locked thread will always encounter another priority change point within k events, and thus can be switched away from before a task stall occurs. In practice, this modification greatly reduces the number of task stalls without compromising the guarantees of PCT within the first k events.

The full implementation of PCT in SECT can be found in Algorithm 1. To set up PCT, we first construct an array to contain the thread priorities, ranging from d to $d + n$. This array is then randomised utilising the Fisher-Yates shuffle algorithm [15]. Next, we randomly choose the change points based on an estimated event count. This set-up requires randomness to ensure that PCT

Algorithm 2: Priority Walk

```
   Input: thread_priorities
1: Function enqueue():
2:     update thread's priority with random()
3: Function dispatch():
4:     if threads enqueued = threads alive then
5:         dispatch highest priority thread
```

can explore various thread interleavings. To introduce determinism to aid in reproducing thread schedules, we need to seed the pseudo-random number generator (PRNG). Unfortunately, the eBPF PRNG helper uses its own pseudo-random internal state and does not allow for a seed to be provided. Thus, we implement a custom PRNG using the Xorshift algorithm [50]. We explore the determinism of SECT further in § 4.4.

Priority Walk This simple algorithm, shown in Algorithm 2, randomly assigns a priority to each thread that enters `enqueue()`. Threads that were previously enqueued retain their assigned priorities until they are dispatched. Upon being re-enqueued, the task will receive a new priority, which may change the previous dispatch ordering. This is in contrast to PCT, where the thread maintains the same assigned priority until a change point is reached. As a result, Priority Walk generally results in more priority changes than PCT, but far fewer context switches as compared to a true random walk.

Random Walk This scheduling algorithm does a simple random walk. At each scheduling point, we randomly select the next task to run from the pool of enqueued tasks. This algorithm is fair, in that threads 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 be run.

3.3 Integrating with Syzkaller

While Syzkaller [23] is primarily concerned with generating sequential system call programs to fuzz the kernel, it still employs two simple, ad-hoc strategies for triggering concurrency bugs. First, Syzkaller does a "double execution collision" by appending the system call program to itself and requiring that the appended portion only references resource values from the original program. All system calls in the appended portion are executed concurrently. Second, Syzkaller also implements a more granular duplication of just certain system calls in a program and marks them as asynchronous. During execution, asynchronous system calls are run concurrently on a separate thread, allowing races to be provoked between these duplicated system calls and the system calls in the original program. Another important property is rerun, which are assigned to consecutive pairs of system calls where the first system call is asynchronous. This allows the pair of system calls to be run multiple times in succession to trigger races between them.

Modifying syz-fuzzer. To increase the bug-finding ability of `SECT` in combination with Syzkaller, we modify Syzkaller to improve the chances of generating an interesting concurrent execution for `SECT`. This involves a few key modifications to syz-fuzzer, which is responsible for generating these programs. The first modification is based on the observation that syz-fuzzer typically only generates short system call programs of around 30 system calls. Passing this through Syzkaller's mutation engine quickly reduces the size of the program to just 3-5 system calls. This short length of programs is often insufficient to trigger non-trivial concurrent interaction between threads. Thus, we increase the length of the program to generate at least 100 system calls and modify the Syzkaller mutation engine to preserve the original length of the program.

After increasing the length of the programs, we also increased the number of asynchronous system calls within each program. System calls can be assigned the `async` property by the syz-fuzzer if i) they don't produce a resource, or ii) they produce a resource, and the subsequent system call consuming the resource is distanced at least one system call away from it. We increase the maximum number of total asynchronous system calls, as well as raise the chance for the `async` property to be assigned to 80%. We do not want to assign all system calls to be asynchronous system calls, as this may cause shallow segmentation faults in user-space, which prevents exploration of the kernel-space executions.

Many concurrency testing algorithms such as PCT assume some fixed knowledge about the execution being scheduled, and thus work best when the same program is run repeatedly within a loop. However, Syzkaller only runs the same program twice by default. We increase the number of collision iterations to 100 in syz-fuzzer to give PCT a higher chance of discovering concurrency bugs. [1]

Modifying syz-executor. To insert `SECT` into the kernel at runtime, we additionally modify syz-executor to set the scheduling policy of its main thread to the `SCHED_EXT` policy. To avoid throttling the fuzzing throughput by serialising the fuzzer process itself, we only set the scheduling policy in the looping thread of syz-executor, which communicates with syz-fuzzer to receive programs and send back the results obtained. The looping thread internally spawns worker threads to execute system calls, which inherits the scheduling properties of the looping thread. Thus, our scheduler will serialise the thread schedules of all worker threads executing the syscall program. In the worker thread, we also insert a call to `sched_yield()` before the execution of each system call. In this way, we can guarantee scheduling granularity at the system call level.

[1]We could further increase the number of collision iterations to 1000, which have also yielded decent results in our internal experiments. However, we must be careful not to spend too much time colliding potentially short system call programs that are introduced by the initial seed corpus. For shorter programs with three or less system calls, we reduce the collision iterations to 30 instead.

Chapter 4

Evaluation

In this chapter, we evaluate various aspects of SECT in comparison to other approaches [22, 23]. We aim to find out the bug-finding effectiveness of SECT integrated with Syzkaller (§ 4.1), as well as the performance overhead incurred by scheduling with eBPF (§ 4.2). We also evaluate the scheduling algorithms implemented by their effectiveness at finding bugs in userspace benchmarks (§ 4.3). Last but not least, we test the determinism of SECT (§ 4.4).

4.1 RQ1: Bug Finding Ability

While controlled scheduling for a single execution can be useful for reproducing a specific bug, finding new bugs at scale typically requires a highly-automated input fuzzer to generate many such executions. To assess SECT's bug finding effectiveness in combination with an input-fuzzer at scale, we conducted a long-running experiment on the latest version of the Linux kernel.

4.1.1 Experimental Set-up

For all evaluations, we start off the fuzzing process with an empty corpus to prevent unfair advantages of discovering more bugs arising from faster executions. We compare an unmodified Syzkaller (referred to as *base*), a modified Syzkaller without SECT (referred to as *mod*), SECT integrated with Syzkaller using PCT (pct-300) and Random Walk (rw2-300). The suffix '-300' represents the

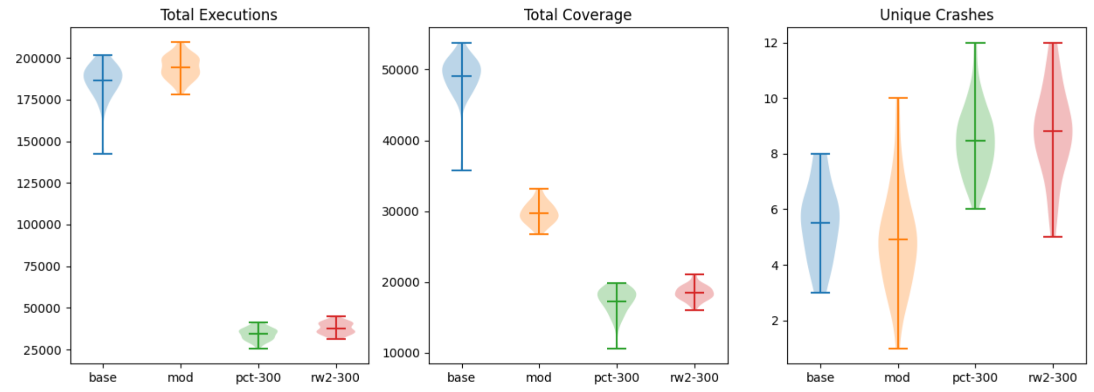


Figure 4.1: Total executions **(left)**, total coverage **(middle)** and unique crashes **(right)** found over 40 fuzzing trials of 24 hours each.

upper bound on the number of random `udelay()`s to skip before a preemption occurs, and is further explained in § 4.1.3. We measure total executions, total coverage obtained and unique crashes found for a single trial, averaged over 40 trials. Each bug-finding experiment is run for 40 trials, with each trial being run for 24 hours.

4.1.2 Results

Impact of modifications to Syzkaller. Comparing between base and mod, we find that mod achieves only roughly 60% of the coverage of base (Figure 4.1 (middle)), despite reaching a similar throughput (Figure 4.1 (left)). This is because we modified Syzkaller to increase the number of system calls in a program from 30 to 100, and we also increased the number of times that Syzkaller repeats the same program from 2 to 100. These modifications will lead to a high amount of redundancy in terms of coverage as they are explicitly repeating similar code paths within the same execution on different threads. In terms of unique crashes found, however, we see that base found a similar number of unique crashes on average, while mod has a larger variation in the number of unique bugs found. This larger variation can be attributed to the changes made to the mutation engine to preserve length, which involves multiple iterations of mutating the original program. In contrast, the baseline Syzkaller only mutates the original

Table 4.1: Type of bugs found by SECT using the PCT algorithm during RQ1 experiments that base and mod did not find.

Type of bug	Count
data race	9
task hung	10
warning	2
starvation	1
assertion error	1

program mostly once. Overall, while our modifications to Syzkaller decrease coverage significantly, they do not appear to impede Syzkaller's ability to find concurrency bugs without a controlled scheduler.

Coverage obtained vs. Unique Crashes From Figure 4.1 (middle), we can see that SECT achieves significantly lower code coverage after 24 hours than either version of Syzkaller alone, less than half that of base. Looking at Figure 4.1 (left), we see that this is likely because configuring Syzkaller with SECT results in nearly a 10x reduction in fuzzing throughput. This is caused both by the performance overhead incurred by SECT itself and the instability of the kprobes instrumentation which causes syz-manager to conduct frequent full reboots of the system. The impact and possible future work regarding these limitations are discussed in further detail in § 5.1.

In Table 4.1, we see the various types of bugs found by SECT with the PCT algorithm that base and mod did not find. In general, due to the instability of the dynamic instrumentation infrastructure, the data races found by our scheduler with KCSAN tend to be shallow race bugs and are hence benign. On the other hand, the bugs that result in “WARNING”s and assertion errors tend to have greater impact, as these arise from source lines of code placed by kernel developers to alert of a potential issue.

However, we notice that a disproportionately large number of crashes in Table 4.1 are related to task hungs, which occur when the task has been blocked for more than 120 seconds. Task hungs occur quite frequently with SECT. While we have tried to debug the source of the task hungs, we are, sadly, unable to identify the root cause of the issue. We suspect that this is likely a real bug in the

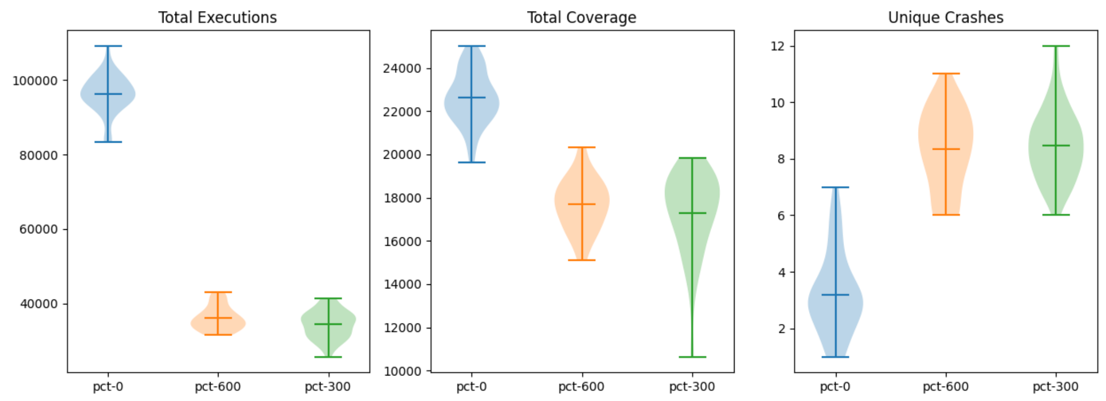


Figure 4.2: Comparison of how increasing the random skips of `udelay()` impacts total executions (**left**), coverage (**middle**) and unique bugs found (**right**).

kprobe infrastructure that arises because triggering a reschedule from within a kprobe function conflicts with the trampolining mechanism that kprobe uses for instrumentation. These task hungs are extremely expensive, because each one takes 2 minutes for a bug report to appear and for the VM to restart, resulting in a significant performance penalty during fuzzing. This is a major fuzz blocker that greatly impedes the performance and progress of `SECT` during fuzzing.

While the addition of `SECT` is costly in terms of code coverage and throughput, Figure 4.1 shows that `SECT` is able to uncover more unique crashes within the 24 hours. This result is statistically significant [1]. Thus, in the context of discovering concurrency bugs, the trade-off in execution speed, and correspondingly code coverage, in using `SECT` appears to be favourable.

4.1.3 `udelay()` Preemption Frequency Sensitivity Analysis

One aspect that greatly impacts the bug-finding ability of `SECT` is its interaction with the primary race detection tool for the kernel, KCSAN. As KCSAN relies on `udelay()` stalling the thread before a memory read to detect a race, it is important to understand how our scheduler interacts with this mechanism. As a recap, we hook to the `udelay()` function using a kprobe, and when the function

[1]We used the Mann-Whitney U-test with alternative="greater". Comparing pct-300 and base gives a U-statistic of 1509 and a p-value of 2.70e-12. Comparing pct-300 and mod gives a U-statistic of 1500 and a p-value of 5.60e-12.

is called, we trigger a reschedule using the BPF system call that internally calls `scx_bpf_kick_cpu()` (c.f. § 3.2.3).

However, `udelay()` is used ubiquitously throughout the kernel when KCSAN is enabled, and thus it is too costly to preempt at every `udelay()` call. As a result, we conduct our experiments with `SECT` preempting on only a random sample of the `udelay()`s in a given execution. Concretely, we compared no udelay preemptions (pct-0), randomly preempting `udelay()`s with probability $\frac{1}{300}$ (pct-300) and randomly preempting with probability $\frac{1}{600}$ (pct-600). We show the results of this sensitivity analysis in Figure 4.2.

Here, we see that no additional preemptions in Figure 4.2 (left) leads to the highest throughput, but there are a statistically significant number of crashes [2] than cannot be found without these additional preemptions as shown in Figure 4.2 (right). Furthermore, it seems that increasing the preemption frequency to $\frac{1}{300}$ does not significantly decrease the throughput as compared to a preemption frequency of $\frac{1}{600}$ `udelay()`s. While a more detailed parameter selection process would likely find a more optimal frequency, for our experiments involving KCSAN, we choose a preemption frequency of every $\frac{1}{300}$ `udelay`s as a reasonable number.

4.2 RQ2: Performance Overhead

Serialising schedulers can incur large overheads as it attempts to make a multi-threaded environment perform at the level of a single-threaded execution. There are also overheads incurred by the scheduling algorithm to compute and update priorities. We evaluate the performance of `SECT` against other execution types such as native execution and Snowboard [22] to understand the impacts of serialising the scheduler at the kernel level.

[2]We used the Mann-Whitney U-test with alternative="greater". Comparing pct-300 and pct-0 gives a U-statistic of 1593.5 and a p-value of 6.83e-15. Comparing pct-600 and pct-0 gives a U-statistic of 1582.5 and a p-value of 1.65e-14.

4.2.1 Experimental Set-up

We compare `SECT` against Syzkaller's native multi-threaded and native single-threaded execution, another `sched_ext` scheduler, and Snowboard [22]. We evaluate performance based on the average time taken to execute a benchmark of programs generated by syz-fuzzer.

The programs are generated by running an instance of Syzkaller and extracting the program logs. For each execution, one program log is output to stdout. We minimise duplicate programs in the benchmark by focusing on the new programs generated by syz-fuzzer, and we prioritise programs with asynchronous calls and reruns to mimic a multi-threaded environment. In total, we collected 100 programs each for 300 and 30 system calls respectively.

To run the experiments with Syzkaller, we first load the test programs into a VM. Then, we use a bash script to load `SECT` into the kernel and start syz-execprog to execute the syscall program for a total of 70 iterations. We modified syz-executor to output to a file the time taken to execute the entire program, which includes a small wait period that syz-executor does to wait for asynchronous threads to complete.

For Snowboard, running the experiment is a two-stage process. We first convert the syscall program into its in-memory representation using a modified version of syz-execprog. Given the output binary file, static analysis is conducted to identify potential memory communication points. To measure the time taken, we modify the hypervisor to start timing in `ski_start_test()` and stop timing in `ski_last_in_test`. These events are called when the TEST_ENTER and TEST_EXIT hypercalls are invoked respectively. The results are output to Snowboard's trace file, which we then extract for analysis.

4.2.2 Results

30 system calls. From Figure 4.3 (sorted by native single-threaded timings), we see that Snowboard takes a significantly longer time to run an execution

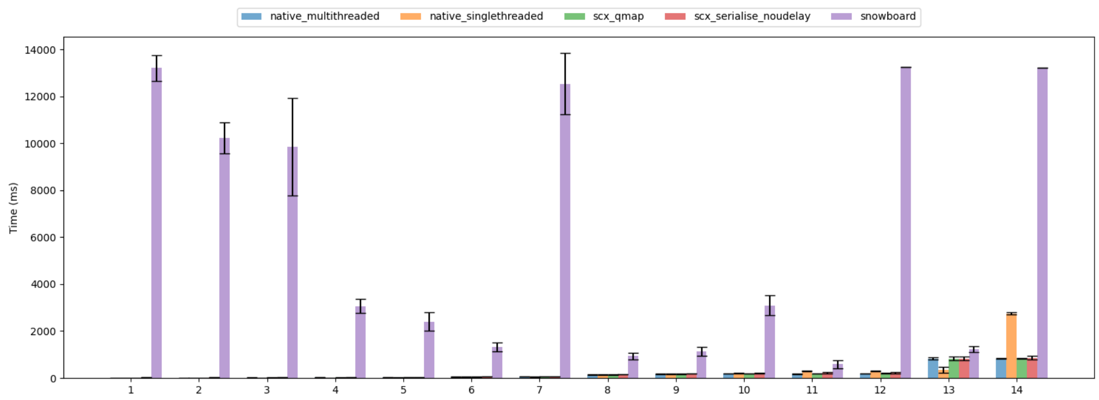


Figure 4.3: Performance of native multi-threaded, native single-threaded, another `sched_ext` scheduler, `SECT`, and Snowboard on programs of 30 system calls.

as compared to native multi-threaded, single-threaded, scx_qmap and `SECT` for programs of 30 system calls. While certain runs of Snowboard take a shorter time (about 1 to 3 seconds) as compared to other runs which take above 10 seconds, Snowboard generally incurs a huge performance overhead across the board. We have contacted Snowboard's authors for comment, and we received a response that certain system calls such as `mount()` are heavy, and they would take a longer time as compared to other system calls. While this does not directly address the enormous performance overhead given that the other four experiments perform significantly better, the authors did not mention that the timings incurred are erroneous.

In addition, Snowboard does not run all programs to completion. If an error such as a segmentation fault occurs during the program execution, Snowboard exits and moves on to the next program. For early exits, we are unable to retrieve the timings that Snowboard took to run. Figure 4.3 shows the programs that Snowboard was able to run to completion for, which is only 14 out of 100.

300 system calls. We also ran with larger programs of 300 system calls to show the scalability of `SECT`. Figure 4.4 shows the performance results for native multi-threaded, native single-threaded, scx_qmap and `SECT`'s executions on programs of 300 system calls, sorted according to native single-threaded. In general, we observe that native multi-threaded and scx_qmap perform similarly and

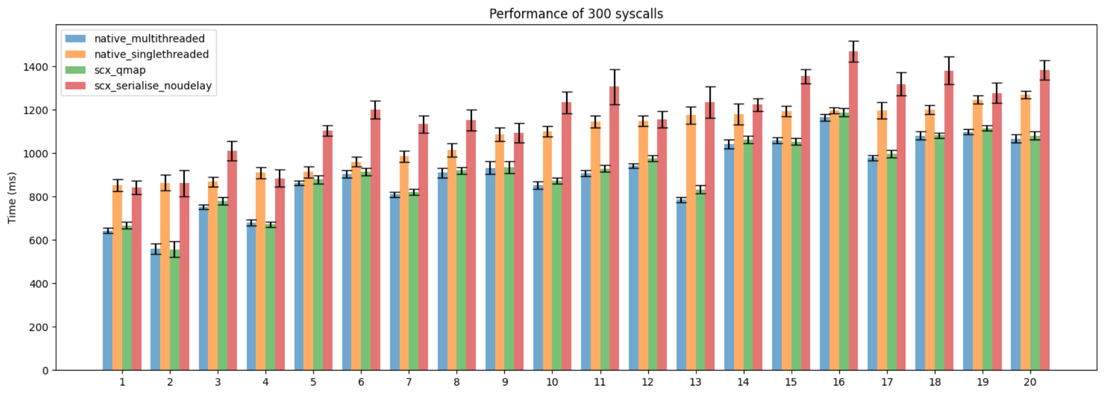


Figure 4.4: Performance of native multi-threaded, native single-threaded, another `sched_ext` scheduler and `SECT` on programs of 300 system calls.

consistently lower than native single-threaded and `SECT`. This is because native multi-threaded and scx_qmap both operate in multi-threaded environments, leading to shorter execution times. While the timings are similar, scx_qmap has a slightly higher performance overhead over native multi-threaded. This is due to the overhead incurred by `sched_ext` to transfer control over to scheduling functions of the eBPF scheduler, instead of having it implemented directly in kernel space.

For native single-threaded and `SECT`, we notice that their performance are also quite similar, although `SECT` usually takes about 200-400ms (roughly 20%-30%) more per program. While `SECT` operates in a multi-threaded environment like native multi-threaded and scx_qmap, `SECT` schedules these threads serially, effectively making the entire execution akin to single-threaded. The additional overhead occurs due to computations performed by the scheduler as well as the eBPF overhead. We also notice that there are certain cases where native single-threaded performs slightly worse than `SECT`, and this could potentially be due to blocking system calls. In a single-threaded execution, Syzkaller explicitly waits for the system call to return before executing another system call. For `SECT`, if the system call is blocking, the thread becomes quiescent and a subsequent thread is dispatched.

Snowboard is not included in this graph because Snowboard places a hard limit on the size of the program's in-memory representation to be less than 1MB.

This restriction is because Snowboard loads the entire program onto the stack in the hypervisor. In-memory representations of programs with 300 system calls have a large size of around 4MB to 8MB. To work around the size restriction, we have tried to modify Snowboard's hypervisor to allocate memory on the heap for loading the program instead of the stack, but we were unsuccessful. We have reached out to Snowboard's authors for comment, but did not receive any input on this matter.

Other concurrency fuzzers. We were unable to run other concurrency fuzzers for experiments due to time constraints. As such, we will perform a simple comparison based off what is in previously reported data. For Krace [78], the authors mention a performance overhead of 47.6% just for loading the kernel file system module without any system calls. Krace also offlines its data race checker, as it takes 2-7 minutes for a seed with 0 to 30 system calls even after optimisation. For SegFuzz [30], their throughput is reported to be 54% that of Syzkaller's. In general, current state-of-the-art concurrency fuzzers incur a large performance penalty due to the complexity of managing thread interleavings via hypervisor-kernel communication and the time required to perform computations required for their metrics.

4.3 RQ3: Impact of Various Scheduling Algorithms on Bug-Finding Effectiveness

For `SECT`, we have implemented three scheduling algorithms that explores the thread interleaving space in different ways. To understand how these algorithms impact bug-finding, we conduct experiments on known buggy programs.

4.3.1 Experimental Set-up

The workflow is similar to § 4.2, but instead of using syz-execprog to execute system call programs, we work on userspace programs that are instrumented with e9patch [14]. We pick 4 programs from the PERIOD paper [74] and evaluate

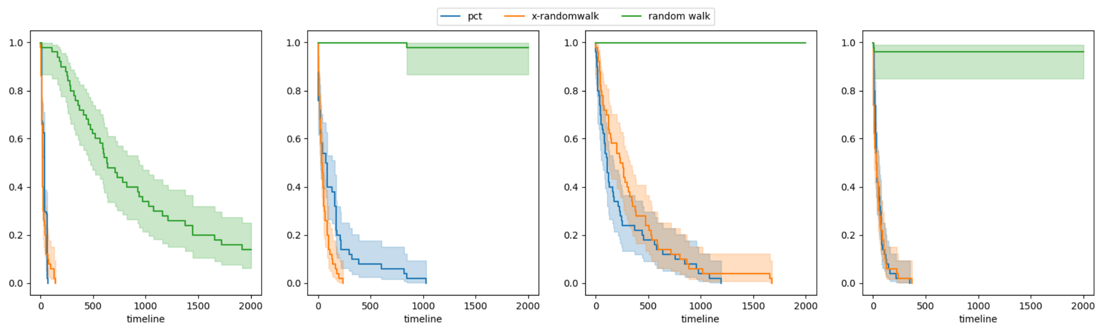


Figure 4.5: Number of iterations taken by PCT, Priority Walk and Random Walk to find a bug in the PERIOD benchmark. From left to right: CS/twostage_bad, CS/reorder_3_bad, CHESS/InterlockedWorkStealQueue, CHESS/Interlocked-WorkStealQueueWithState.

the bug-finding effectiveness of each algorithm based on the average number of iterations to each bug. Each scheduling algorithm is given a total of 2000 iterations to find a bug, and we perform 50 trials of 2000 iterations each to obtain an average.

4.3.2 Results

`SECT` is able to run effectively on user-space programs, and managed to find bugs in all tested programs with PCT and Priority Walk.

Bug-finding effectiveness. From Figure 4.5, we see that Random Walk performs the worst in finding bugs. Random Walk is only able to find bugs consistently in CS/twostage_bad (leftmost), but there are still some trials where it is unable to find the bug. While PCT and Priority Walk are both able to find the bugs within the first 200 iterations over 50 trials, Random Walk takes about 5 times longer to find the bug. For the remaining three programs, Random Walk also performs poorly. Random Walk was not able to find the bug for CHESS/InterlockedWork-StealQueue given 2000 iterations, and was only able to find the bug in 1-2 trials for CS/reorder_3_bad and CHESS/InterlockedWorkStealQueueWithState.

PCT and Priority Walk are generally more capable at finding bugs, and they manage to find the bugs within a similar number of iterations for CS/twostage_bad (leftmost) and CHESS/InterlockedWorkStealQueueWithState (rightmost). For CS/reorder_3_bad (second from left), Priority Walk shows a better bug-finding ability with only a maximum of 250 iterations required to sieve out the bug and converges much faster than PCT, which requires up to 1000 iterations to find the bug. For CHESS/InterlockedWorkStealQueue (second from right), PCT performs better at finding bugs than Priority Walk. Thus, PCT and Priority Walk both show similar and reasonable performance in being able to find concurrency bugs.

4.4 RQ4: Determinism of Scheduler

Determinism in concurrency fuzzing tools is useful for bug reproducibility, which is essential for kernel developers to understand the underlying problem. We assess the determinism of the scheduler by monitoring the interleavings explored within a short userspace program given the same starting random seed.

4.4.1 Experimental Set-up

To measure the determinism of the scheduler, we use a program which creates two threads that each perform arithmetic operations on a shared variable. After each operation, `sched_yield()` is called to allow `SECT` to choose the next thread to execute. At the end of the program execution, the shared variable can take on a wide range of possible values depending on the thread interleaving. The metric used to measure determinism would be the variety of output from the program, with less variety implying more determinism.

Determinism is measured for the PCT algorithm, because it allows for a seeded random number generator. The current Priority Walk and Random Walk implementations rely on the BPF pseudorandom number generator, and as such, we cannot seed the starting state. This is not a fundamental limitation, but rather a simplifying design decision. To perform our experiments, we i) insert `SECT` with the same seed, ii) execute the multi-threaded program, and iii) deregister `SECT`. We repeat these steps for a total of 1000 iterations.

Number of vCPUs	Determinism Achieved
1	100%
3	46.6%
6	73.3%

Table 4.2: Comparison of determinism obtained by running SECT with the same seed across an increasing number of vCPUs.

4.4.2 Results

In Table 4.2, for 1 vCPU, we see that the same result is obtained for all 1000 iterations, meaning that 100% determinism is achieved when 1 vCPU is used. For 3 vCPUs, SECT performs less deterministically and generally hovers between two interleavings, with the most popular interleaving having a count of 466. This number is increased to 733 for 6 vCPUs, which means that using SECT with 6 vCPUs presents more deterministic results than 3 vCPUs.

The source of non-determinism arises because the threads are allowed to run on separate vCPUs. Even with full serialisation of the threads by the scheduler, the determinism is not guaranteed in a multi-CPU environment due to factors such as cache coherence protocols, memory access patterns and hardware level optimisations for SMP systems. Additionally external interrupts that are serviced in parallel on a different CPU could also influence the determinism achieved. Performing the experiment again on VMs with multiple CPUs but only allowing the program to run on one vCPU (using `taskset`) also achieves 100% determinism. As such, if one wishes to deterministically replay bugs found by SECT, it is best to use a single vCPU for fuzzing to ensure that the same interleaving can be replicated again with high probability.

4.5 Case Studies

Overall, we have submitted a total of six bug reports, of which five are to `sched_ext` and one to the mainline kernel. We have received responses for all the reports, and five of them are currently fixed. Of the fixed bugs, one is a potential deadlock scenario (see Figure 2.1), two are "WARNING"s relating to

Thread 1 – xa_get_order		Thread 2 – xas_store	
1	// read slots pointer if (!xa_is_sibling(xas.xa_node->slots[slot])) ...		

2	udelay(180);	2	// write to slots pointer
			rcu_assign_pointer(*slot, entry);

3	// read slots pointer again if (!xa_is_sibling(xas.xa_node->slots[slot])) ...		

Figure 4.6: Data race in `xa_get_order` and `xas_store`. The numbers at the side represent the coarse timestamps of the operations.

improper locking and out-of-sync variables, one is a memory leak and the last one is a data race in the mainline kernel.

4.5.1 Bug #1: Kernel RCU race

This is a data race in xa_get_order / xas_store of the Xarray library. The conflicting access occurs at the following read and write operations shown in Figure 4.6. The data race occurs because while RCU is used to write to the pointer, the read does not use RCU to properly dereference the pointer, leading to value change from 0x0000000000000000 -> 0xffffea0000488f00. This race is benign since the write uses RCU, so the written pointer will either take on the old value or the new value, and no data corruption will occur. However, it could cause a wrong `order` to be computed and returned by the function `xa_get_order`.

4.5.2 Bug #2: Unbalanced depth in `scx_ops_bypass`

This is not exactly a data race, but rather an edge case in `sched_ext` that was not accounted for, leading to a variable `depth` going out-of-sync. The WARNING is placed by `sched_ext` developers to warn them if the variable `depth` goes below 0.

This occurs when a `sched_ext` scheduler is loaded while a device driver callback handler, `scx_pm_handler()`, is in progress. If `scx_pm_handler()` starts without a loaded `sched_ext` scheduler, it skips calling `scx_ops_bypass()`. When the scheduler is subsequently loaded, it results in the value of `depth` not being updated properly in `scx_ops_bypass()`, leading to the warning. Such a scenario can be triggered easily by opening `/dev/snapshot`.

To fix this, the developers made sure that `scx_pm_handler()` always calls `scx_ops_bypass()`, and uses a mutex in `scx_ops_bypass()` to synchronise the function with the status of whether a `sched_ext` scheduler is loaded into the kernel.

4.6 Experimental Hardware and Set-up

RQ1. Experiments are performed on an Ubuntu 22.04 server with Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz and 192GB of RAM. We perform bug finding on a release candidate version of the Linux kernel, v6.8-rc6, in the workflow shown in Figure 3.1. We launch 6 VMs running in parallel, one for each configuration with each VM allocated 2 vCPUs and 4GB of RAM.

RQ2. Experiments are performed machines with Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz and 64GB of RAM. For Syzkaller and `SECT`, we run experiments on an Ubuntu 22.04 server. For Snowboard, we run experiments on an Ubuntu 20.04 server, although the authors recommended running Snowboard on Ubuntu 18.04 instead.

RQ3 & RQ4. Experiments are performed on an Ubuntu 22.04 server with Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz and 32GB of RAM.

Chapter 5

Limitations & Future Work

While SECT introduces a novel methodology in the domain of concurrency testing, it also presents areas for enhancement and further research. We discuss the limitations and possible future work in this chapter.

5.1 Scheduling Granularity

The single most important limitation of SECT is that it is not currently able to inject fine-grained preemptions into kernel code being tested. The following section discusses this limitation and its secondary effects on SECT's design.

Injecting Additional Preemptions. The ***crucial*** limitation of SECT for testing kernel-space code at the time of writing this thesis is that it cannot accurately trigger preemptions before operations of interest. While we can insert preemptions in *user-space* code by adding additional calls to `sched_yield`, this is not straightforward in the context of OS kernel code, as discussed in Section 3.2.3. The current `sched_ext` framework does not allow us to trigger preemptions from within dynamically instrumented code. Without this capability, SECT cannot reliably perform fine-grained thread switching between events of interest. For example, the scheduler may not be invoked in-between consecutive memory accesses, causing an intervening data race to be missed. We note that this is not a fundamental limitation, but is just a missing feature in the evolving `sched_ext` framework used by `SECT`. However, at the time of writing, `sched_ext` is still a relatively new framework under active development. Currently, the `sched_ext` developers are working on adding new features that would aid the implementation of task preemption [1]. This may provide a more stable solution as compared to invoking our custom BPF system call in a kprobe hook function. Another option would be to look towards static instrumentation of the kernel [78], but we warn that such an approach is likely to incur a large performance overhead.

Unfortunately, this one key limitation has several additional constraints it imposes on other aspects of our approach as well. For example, many algorithms for concurrency testing such as POS [79] require some semantic knowledge about the events being executed, such as which lock or memory location they are accessing. Without the ability to inject preemption points at these events of interest, we also cannot leverage this semantic information to build more sophisticated scheduling algorithms. Additionally, prior work [22, 29] has used static analysis to identify key events of interest; again, without the ability to inject preemptions at the points identified by static analysis, `SECT` cannot adequately utilise such information.

Interaction with KCSAN The inability to trigger preemptions also greatly reduces the efficacy of KCSAN when used with `SECT`. To detect bugs, KCSAN randomly samples for memory read operations during a thread's execution. When a memory read has occurred, KCSAN then delays the currently running thread before executing another read operation on the same memory location. It then checks if the read data after the delay is the same as before the delay, reporting a race if the data has changed due to an intervening write. However, under `SECT` naively, *there can be no such intervening write*, as only one task is allowed to run at a time. Thus, for KCSAN to be able to detect bugs with `SECT`, we must manually trigger a context switch to allow another thread to run in between the beginning and end of the KCSAN check. This requirement, combined

[1]Based on prior discussion from `sched_ext` developers [61], they have newly implemented a hookable function `ops.tick()` that is called on every CPU tick cycle [62], from which `scx_bpf_kick_cpu()` can be called. They have also implemented support for iteration in a dispatch queue [27].

with our inability to inject preemptions (above) lead us to the workaround using `scx_bpf_kick_cpu()` to trigger approximate preemptions during random invocations of `udelay()` as described in Section 3.2.3.

`scx_bpf_kick_cpu()` Approximate Preemptions. To have a chance at triggering preemptions in KCSAN instrumented code, we used a kprobe hook to call the `scx_bpf_kick_cpu()` function, which sends an interrupt that reschedules the current task. However, these preemptions are only *approximate* in that the interrupt sent from `scx_bpf_kick_cpu()` is only serviced after a small delay of a few microseconds, which may exceed the `udelay()` time that KCSAN delays the read thread. In addition to being only approximate, hooking into a frequently called functions adds significant performance overhead. Also, because kprobes are dynamically inserted into the kernel at runtime via a trampoline, we have observed that its instability can cause task hungs within the system when the function invokes `scx_bpf_kick_cpu()`. During fuzzing, this causes the VM to reboot multiple times due to hung tasks, resulting in fuzz blockers that greatly reduce fuzzing throughput. Tracepoints, on the other hand, are more stable since the hook points are predefined in the kernel source code, but the hook locations provided by tracepoints remain limited and thus could not be used for KCSAN specifically.

5.2 Fuzzing Methodologies

This section discusses various improvements that can be made to essential concurrency fuzzing components: the scheduling algorithms, coverage metric and data race detector. By incorporating better fuzzing methods, we hope that future work can leverage on our scheduler and improve its bug-finding abilities.

Scheduling Algorithms. Currently, the implemented scheduling algorithms are all stateless and require no semantic information about the kernel or program being executed aside from an estimated number of preemptible events. These limitations are not fundamental to `SECT`, but reflect an important design decision and tradeoff. More complex algorithms that understand the semantics of the events being executed can be much more effective, but require heavy instrumentation of the system under test. We note that such an effort would first require more fine-grained and accurate preemptions as discussed in § 5.1. Additionally, the constrained programming environment of eBPF increases the complexity of communicating with the kernel.

Our current implementation of `SECT` in eBPF is oblivious to the system and program state, as we are unable to retrieve any semantic information about the program dynamically during its execution. This limits our choice of scheduling algorithms to essentially black-box ones. We acknowledge that a more intelligent scheduling algorithm could look at providing our scheduler with a feedback loop from the program under test and adjust subsequent thread schedules based on the feedback [76]. This is likely to better explore the thread interleaving space and lead to more fruitful bug-finding. Future work can explore incorporating more complex algorithms, as well as stable communication methods between eBPF and the kernel.

Integration with Syzkaller. Syzkaller is a large, real-world project that is actively maintained and has been constantly improving since its creation in 2015. Due to time constraints, we focused on the simplest modifications to make Syzkaller invoke our scheduler and generate a reasonable set of concurrent system call programs. Future work can explore further modification of syz-fuzzer to improve system call program generation to produce more complex concurrent system call programs. This could include adding various coverage metrics such as alias coverage [78], interleaving segment coverage [30], or even creating a new concurrency aware coverage metric. Modifications to syz-executor to detect task hungs and recover the system without rebooting would also aid in the fuzzing process with `SECT`.

Another limitation is that integrating with well-known kernel fuzzers might not be optimal for finding concurrency bugs. Syzkaller operates on `kcov`, which guides fuzzing towards unexplored paths, but not new thread interleavings. Without exploring the thread interleaving space, this significantly decreases the probability of triggering a concurrency bug. To tackle this, a new coverage metric may be required to place more emphasis on exploring uncommon thread interleavings.

Finding Data Races. While KCSAN does not generally incur false positive reports (i.e. most races are true races), it does report many benign races. Benign races are races that can be tolerated by the kernel and does not affect overall system correctness. Such races are typically ignored by kernel developers as fixing them might not be worth both the implementation effort and the performance overhead. Differentiating between a benign and harmful data race usually requires manual labour to verify against the kernel source code, which is time-consuming [30, 78]. To work around this, we identify sources of benign races during our experiments, such as timer and statistic related functions, and configure Syzkaller to ignore such races during fuzzing.

Furthermore, while KCSAN is a useful data race detector, its watchpoint-based sampling approach assumes a multi-threaded environment and does not generally tie in well with a serialised schedule. Under a serialised schedule, the thread that has performed the read and entered `udelay()` is highly likely to be scheduled to resume execution under the PCT algorithm. This occurs because we do not preempt the currently running thread at every invocation of `udelay()` due to large performance penalties. As a result, there is a high likelihood that KCSAN will conduct its check before the other thread is scheduled to *write* to the memory location, causing a potential race to go undetected. Coupled with a sampling approach that is also likely to miss out on potential bugs, this may lead to many false negatives. To prevent this, a custom data race checker may be required to trace the read and write operations to *each* memory location throughout the program's execution and perform a race checking step offline.

5.3 Other Scheduler Enhancements

Handling Synchronisation Primitives. Kernel synchronisation primitives such as spinlocks are used frequently throughout the kernel when a thread is waiting for a resource. Currently, we do not have a way to detect if a thread is in a synchronisation primitive via eBPF alone. This would require instrumentation of the kernel to send signals to our scheduler. While we can preempt a thread when it is stuck busy waiting in a synchronisation primitive, this is not of much use if we are using PCT as a scheduling algorithm. The PCT algorithm would continue scheduling this thread of highest priority until a change point is reached, which could potentially starve lower-priority threads. As such, we need a way to handle such synchronisation primitives to ensure liveness of the program.

Exercising Thread Control. While `SECT` has to ability to schedule all threads, including kernel and background threads, within the system, we choose to only schedule the threads initiated by a program under test (e.g. threads executing a sequence of system calls). This choice reduces scheduling complexity, and ensure that critical kernel threads are prioritised to prevent them from potentially slowing down the entire system if they are blocked for too long (e.g., ksoftirqd/N, rcuop/N). Nevertheless, it is trivial for future work leveraging `SECT` to explore scheduling all tasks in the system which allows for more deterministic control over the triggering of concurrency bugs involving background kernel threads.

Chapter 6

Related Work

Kernel Fuzzing. Over the years, coverage-guided fuzzing tools have achieved much success in discovering kernel bugs. However, despite the large volume of work in this area [11, 23, 24, 42, 54, 59, 67, 72], these tools are primarily tailored to handle inputs for sequential execution, and only consist of naive approaches for exposing concurrency bugs. These works are focused on exploring uncommon paths in the system under test, rather than exploring the thread interleaving space, which limits their potential for finding concurrency bugs.

Kernel Concurrency Testing. Specific to the kernel, testing frameworks like Razzer [29] and Snowboard [22] usually rely on heuristics to narrow the search space for potential data races. These works first perform static and dynamic analysis to identify the memory location of potential race candidates before systematically interleaving the threads to expose the bug.

These approaches have seen success by uncovering many kernel concurrency bugs, but they are not without their limitations. They usually require specialised hypervisors [19, 22, 29] which are heavy-weight, and incur a large performance overhead. Despite the involved development of such tools, they only apply to kernels and cannot extend to user-space programs. In addition, Razzer only serialises a small subset of memory read/write operations, which restricts its ability to reproduce bugs. Snowboard can only test short system call programs, greatly limiting its scalability.

In contrast, SECT proposes a lightweight approach that replaces the kernel scheduler via eBPF. With dynamic runtime loading and unloading, the testing process becomes much simpler to deploy and development time is shortened (Goal **G1**). While other controlled concurrency testing approaches need to explicitly model blocking operations to avoid false deadlocks [76] or indirectly detect them via an instrumented hypervisor [19], SECT is integrated into the operating system itself and thus can natively detect when a thread is blocking.

Since the kernel scheduler is responsible for handling all threads in the system, SECT extends its input to both user-space and kernel programs (Goal **G2**). SECT provides a high level of determinism when running with 1 vCPU, which helps the bug reproduction process. SECT can also scale well to arbitrarily large programs, allowing real-world programs to be tested efficiently (Goal **G2**).

Kernel Concurrency Fuzzing. Concurrency fuzzing in the kernel combines the methods of concurrency testing to explore the thread interleaving space, and the methods of conventional fuzzing to explore various test inputs. Krace [78] and SegFuzz [30] explore uncommon thread interleavings by guiding fuzzers with new coverage metrics and mutation-based strategies. However, they also inherit the limitations of concurrency testing of being heavy-weight and restrictive in terms of performance and scale. Furthermore, Krace uses random delay injections to uncover new interleavings and lacks fine-grained control over the threads. SECT, on the other hand, provides full control over the scheduling of threads.

Data Race Detection. There are numerous data race detection works [2, 16, 32, 36, 58, 60, 80] that aim to detect concurrency bugs through a variety of methods. These include KCSAN's watchpoint-based sampling approach and probabilistic lockset analysis. In general, SECT relies on existing data race detectors to provide bug detection capabilities. As such, SECT's design allows for data race detectors compatible with the Linux kernel to be integrated easily into its fuzzing architecture (Goal **G3**).

Chapter 7

Conclusion

This work presents SECT, a lightweight approach to concurrency fuzzing. By leveraging eBPF, we develop a kernel scheduler that serialises thread execution to search for concurrency bugs. Unlike traditional methods that require modifications to the hypervisor or kernel recompilation, our approach ensures ease of deployment and flexibility in testing both user-space and kernel programs. Implementing SECT at the kernel level ensures that it can be compatible with existing works [23, 36]. Our evaluations of SECT show that it is a promising approach with lower performance overhead, high determinism and reasonable bug-finding ability.

Looking ahead, SECT helps to lower the barrier of entry to concurrency fuzzing and enables efficient development and testing of scheduling algorithms. Possible future work includes i) developing a more intelligent scheduling algorithm to effectively explore the thread interleaving space, which may include but is not limited to machine learning techniques, ii) exploring alternatives for preemption to improve the scheduling granularity for more precise control over the thread interleavings, as well as iii) refining the fuzzing pipeline with more sophisticated modifications to Syzkaller to guide mutations towards generating better concurrency-related syscall programs.

Bibliography

[1] S. Abbaspour Asadollah, D. Sundmark, S. Eldh, and H. Hansson, "Concurrency bugs in open source software: A case study", *Journal of Internet Services and Applications*, vol. 8, no. 1, p. 4, Apr. 2017, ISSN: 1869-0238. [Online]. Available: https://doi.org/10.1186/s13174-017-0055-2.

[2] A. Ahmad, S. Lee, P. Fonseca, and B. 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*, ser. ASPLOS '21, Virtual, USA: Association for Computing Machinery, 2021, pp. 647–660, ISBN: 9781450383172. [Online]. Available: https://doi.org/10.1145/3445814.3446727.

[3] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. 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)*, 2006, pp. 458–465.

[4] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems", *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981, ISSN: 0360-0300. [Online]. Available: https://doi.org/10.1145/356842.356846.

[5] P. Bonzini, "An introduction to lockless algorithms", https://lwn.net/Articles/844224/, Feb. 2021.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs", *SIGPLAN Not.*, vol. 45, no. 3, pp. 167–178, Mar. 2010, ISSN: 0362-1340. [Online]. Available: https://doi.org/10.1145/1735971.1736040.

[7] "CFS scheduler", https://docs.kernel.org/scheduler/sched-design-CFS.html.

[8] J. Corbet, "Lockless algorithms for mere mortals", https://lwn.net/Articles/827180/, Jul. 2020.

[9] J. Corbet, "The extensible scheduler class", https://lwn.net/Articles/922405/, Feb. 2023.

[10] J. Corbet, A. Rubini, and G. Kroah-Hartman, "Linux device drivers, 3rd edition", [Online]. Available: https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch05.html.

[11] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2123–2138, ISBN: 9781450349468. [Online]. Available: https://doi.org/10.1145/3133956.3134069.

[12] "CVE-2016-5195." Available from NIST, CVE-ID CVE-2016-5195. Nov. 2016. [Online]. Available: https://nvd.nist.gov/vuln/detail/cve-2016-5195.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA: USENIX Association, Dec. 2004. [Online]. Available: https://www.usenix.org/conference/osdi-04/mapreduce-simplified-data-processing-large-clusters.

[14] G. Duck, "E9patch", https://github.com/GJDuck/e9patch, 2020.

[15] M. Eberl, "Fisher-yates shuffle", *Arch. Formal Proofs*, vol. 2016, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:34933560.

[16] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel", in *Operating System Design and Implementation (OSDI'10)*, USENIX, Oct. 2010. [Online]. Available: https://www.microsoft.com/en-us/research/publication/effective-data-race-detection-for-the-kernel/.

[17] D. Etiemble, “Technologies and computing paradigms: Beyond moore’s law?” *arXiv preprint arXiv:2206.03201*, 2022.

[18] C. Flanagan and S. N. Freund, “Fasttrack: Efficient and precise dynamic race detection”, in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:2696584.

[19] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, “SKI: Exposing kernel concurrency bugs through systematic schedule exploration”, in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 415–431, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca.

[20] G. Gilman and R. J. Walls, “Characterizing concurrency mechanisms for nvidia gpus under deep learning workloads”, *Performance Evaluation*, vol. 151, p. 102 234, 2021, ISSN: 0166-5316. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531621000511.

[21] G. Gilman and R. J. Walls, “Characterizing concurrency mechanisms for nvidia gpus under deep learning workloads”, *Performance Evaluation*, vol. 151, p. 102 234, 2021, ISSN: 0166-5316. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531621000511.

[22] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, “Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis”, in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21, Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 66–83, ISBN: 9781450387095. [Online]. Available: https://doi.org/10.1145/3477132.3483549.

[23] Google, “Syzkaller”, https://github.com/google/syzkaller, 2015.

[24] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. 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.

[25] T. Heo, D. Vernet, J. Don, and B. Rhoden, "Sched_ext", https://github.com/sched-ext/sched_ext, 2023.

[26] Z. Huang, S. Guo, M. Wu, and C. Wang, "Understanding concurrency vulnerabilities in linux kernel", 2022. arXiv: 2212.05438 [cs.CR].

[27] "Implement bpf dsq iterator by htejun", https://github.com/sched-ext/sched_ext/pull/185.

[28] "Jcstress", https://github.com/openjdk/jcstress.

[29] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing", in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 754–768.

[30] D. R. Jeong, B. Lee, I. Shin, and Y. Kwon, "Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing", in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2104–2121.

[31] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection",, Jan. 2022.

[32] Y. Jiang, Y. Yang, T. Xiao, T. Sheng, and W. Chen, "Drddr: A lightweight method to detect data races in linux kernel", *J. Supercomput.*, vol. 72, no. 4, pp. 1645–1659, Apr. 2016, ISSN: 0920-8542. [Online]. Available: https://doi.org/10.1007/s11227-016-1691-1.

[33] O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing gpu concurrency in heterogeneous architectures", in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 114–126.

[34] "KCOV: Code coverage for fuzzing", https://docs.kernel.org/dev-tools/kcov.html.

[35] "Kernel Address Sanitizer (KASAN)¶", https://docs.kernel.org/dev-tools/kasan.html.

[36] "Kernel Concurrency Sanitizer (KCSAN)", https://docs.kernel.org/dev-tools/kcsan.html.

[37] "Kernel Probes (kprobes)", https://docs.kernel.org/trace/kprobes.html.

[38] "Ktsan", Apr. 2024. [Online]. Available: https://github.com/google/kernel-sanitizers/blob/master/KTSAN.md.

[39] E. Ladan-Mozes and N. Shavit, "An optimistic approach to lock-free fifo queues", *Distributed Computing*, vol. 20, pp. 323–341, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:13467031.

[40] M. Larabel, "Linux 5.12 coming in at around 28.8 million lines, amdgpu driver closing in on 3 million", Mar. 2021. [Online]. Available: https://www.phoronix.com/news/Linux-5.12-rc1-Code-Size.

[41] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance concurrency control mechanisms for main-memory databases", *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 298–309, Dec. 2011, ISSN: 2150-8097. [Online]. Available: https://doi.org/10.14778/2095686.2095689.

[42] D. Li and H. Chen, "Fastsyzkaller: Improving fuzz efficiency for linux kernel fuzzing", *Journal of Physics: Conference Series*, vol. 1176, no. 2, p. 022 013, Mar. 2019. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1176/2/022013.

[43] H. Liang, Y. Chen, Z. Xie, and Z. Liang, "X-afl: A kernel fuzzer combining passive and active fuzzing", in *Proceedings of the 13th European Workshop on Systems Security*, ser. EuroSec '20, Heraklion, Greece: Association for Computing Machinery, 2020, pp. 13–18, ISBN: 9781450375238. [Online]. Available: https://doi.org/10.1145/3380786.3391400.

[44] "Linux (6.8-rc6)", https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tag/?h=v6.8-rc6, 2024.

[45] "Linux kernel memory barriers", https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[46] "Lock types and their rules - sempahore", https://www.kernel.org/doc/html/latest/locking/locktypes.html.

[47] "Locking lessons", https://docs.kernel.org/locking/spinlocks.html.

[48] "Lockless ring buffer design", https://docs.kernel.org/trace/ring-buffer-design.html.

[49] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics", in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII, Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 329–339, ISBN: 9781595939586. [Online]. Available: https://doi.org/10.1145/1346281.1346323.

[50] G. Marsaglia, "Xorshift rngs", *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 2003. [Online]. Available: https://www.jstatsoft.org/index.php/jss/article/view/v008i14.

[51] U. Mathur, A. Pavlogiannis, and M. Viswanathan, "The complexity of dynamic data race prediction", in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2020, pp. 713–727.

[52] M. Musuvathi and S. Qadeer, "Chess: Systematic stress testing of concurrent software", in *Logic-Based Program Synthesis and Transformation*, G. Puebla, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 15–16, ISBN: 978-3-540-71410-1.

[53] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems", in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 677–689, ISBN: 9781450327589. [Online]. Available: https://doi.org/10.1145/2723372.2749436.

[54] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing OS fuzzer seed selection with trace distillation", in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 729–743, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor.

[55] "Percpu rw semaphores", https://docs.kernel.org/locking/percpu-rw-semaphore.html.

[56] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki, "Sharing data and work across concurrent analytical queries", *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 637–648, Jul. 2013, ISSN: 2150-8097. [Online]. Available: https://doi.org/10.14778/2536360.2536364.

[57] I. Psaroudakis, T. Scheuer, N. May, and A. Ailamaki, "Task scheduling for highly concurrent analytical and transactional main-memory workloads",, 2013. [Online]. Available: http://infoscience.epfl.ch/record/188280.

[58] J. Roemer, K. Genç, and M. D. Bond, "Smarttrack: Efficient predictive race detection", in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 747–762, ISBN: 9781450376136. [Online]. Available: https://doi.org/10.1145/3385412.3385993.

[59] G. Ryan, A. Shah, D. She, and S. Jana, "Precise detection of kernel data races with probabilistic lockset analysis", in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2086–2103.

[60] G. Ryan, A. Shah, D. She, and S. Jana, "Precise detection of kernel data races with probabilistic lockset analysis", in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2086–2103.

[61] "Sched_ext slack discussion on preemption features", https://schedextworkspace.slack.com/archives/C05UTFPQC1H/p1712679846082769.

[62] "Scx: Implement cpufreq support by htejun", https://github.com/sched-ext/sched_ext/pull/180/commits/1d884354f24ac4a7c8690a8698ff328d14175

[63] "Sequence counters and sequential locks", https://docs.kernel.org/locking/seqlock.html.

[64] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice", in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09, New York, New York, USA: Association for

Computing Machinery, 2009, pp. 62–71, ISBN: 9781605587936. [Online]. Available: https://doi.org/10.1145/1791194.1791203.

[65] S. D. Stoller, "Testing concurrent java programs using randomized scheduling", *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4, pp. 142–157, 2002, RV'02, Runtime Verification 2002 (FLoC Satellite Event), ISSN: 1571-0661. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066104805826.

[66] Stratodesk, "The multi-kernel-os: One system, widest hardware support ever", Apr. 2014. [Online]. Available: https://www.stratodesk.com/multi-kernel-os-introduction/.

[67] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation learning guided kernel fuzzing", in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 344–358, ISBN: 9781450387095. [Online]. Available: https://doi.org/10.1145/3477132.3483547.

[68] H. Sundell and P. Tsigas, "Lock-free deques and doubly linked lists", *Journal of Parallel and Distributed Computing*, vol. 68, no. 7, pp. 1008–1020, 2008, ISSN: 0743-7315. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731508000518.

[69] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software", *Dr. Dobb's Journal*, vol. 30, no. 3, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm.

[70] H. Sutter and J. Larus, "Software and the concurrency revolution", *ACM Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[71] "Syzbot dashboard", [Online]. Available: https://syzkaller.appspot.com/upstream/fixed.

[72] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel", in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23, <conf-loc>, <city>Copenhagen</city>, <country>Denmark</country>, </conf-

loc>: Association for Computing Machinery, 2023, pp. 1630–1644. [Online]. Available: https://doi.org/10.1145/3576915.3623146.

[73] "Using the linux kernel tracepoints", https://docs.kernel.org/trace/tracepoints.html.

[74] C. Wen, M. He, B. Wu, Z. Xu, and S. Qin, "Controlled concurrency testing via periodical scheduling", in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 474–486, ISBN: 9781450392211. [Online]. Available: https://doi.org/10.1145/3510003.3510178.

[75] "What is RCU? – "Read, Copy, Update"", https://www.kernel.org/doc/html/next/RCU/whatisRCU.html.

[76] D. Wolff, S. Zheng, G. Duck, U. Mathur, and A. Roychoudhury, "Greybox fuzzing for concurrency testing",, 2024.

[77] T. Xiao, E. Jang, D. Kalashnikov, S. Levine, J. Ibarz, K. Hausman, and A. Herzog, "Thinking while moving: Deep reinforcement learning with concurrent control", *CoRR*, vol. abs/2004.06089, 2020. arXiv: 2004.06089. [Online]. Available: https://arxiv.org/abs/2004.06089.

[78] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems", in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1643–1660.

[79] X. Yuan, J. Yang, and R. 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*, Springer, 2018, pp. 317–335.

[80] T. Zhang, C. Jung, and D. Lee, "Prorace: Practical data race detection for production use", in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, Xi'an, China: Association for Computing Machinery, 2017, pp. 149–162, ISBN: 9781450344654. [Online]. Available: https://doi.org/10.1145/3037697.3037708.