



저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:



저작자표시. 귀하는 원저작자를 표시하여야 합니다.



비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 [이용허락규약\(Legal Code\)](#)을 이해하기 쉽게 요약한 것입니다.

[Disclaimer](#)

M.S. THESIS

# Reproducible Kernel Fuzzer for Concurrency Bugs through Deterministic Scheduling

결정론적 스케줄링을 이용하여 동시성 버그를 재현하는  
커널 퍼져 제작

BY

LEE Jin Woo

AUGUST 2022

DEPARTMENT OF ELECTRICAL AND  
COMPUTER ENGINEERING  
SEOUL NATIONAL UNIVERSITY

M.S. THESIS

# Reproducible Kernel Fuzzer for Concurrency Bugs through Deterministic Scheduling

결정론적 스케줄링을 이용하여 동시성 버그를 재현하는  
커널 퍼져 제작

BY

LEE Jin Woo

AUGUST 2022

DEPARTMENT OF ELECTRICAL AND  
COMPUTER ENGINEERING  
SEOUL NATIONAL UNIVERSITY

# Reproducible Kernel Fuzzer for Concurrency Bugs through Deterministic Scheduling

결정론적 스케줄링을 이용하여 동시성 버그를 재현하는  
커널 퍼져 제작

지도교수 이 병 영  
이 논문을 공학석사 학위논문으로 제출함

2022년 5월

서울대학교 대학원

전기·정보 공학부

이 진 우

이진우의 공학석사 학위 논문을 인준함

2022년 6월

위 원 장:	김 장 우	(인)
부위원장:	이 병 영	(인)
위 원:	심 재 응	(인)

# Abstract

As kernel fuzzer has been studied and becomes better for years, the number of reported bugs from the fuzzer increased. Since kernel developers could not analyze all of the bugs, the situation emphasizes the importance of bug reproduce which can provide debug information. Unfortunately, the fuzzer often fails to reproduce bug, and one of the most difficult bug type is concurrency bug. The concurrency bug requires certain conditions between several threads, and non-deterministic thread interleavings from kernel scheduling prevent reproducing. As a result, some concurrency bugs are left unpatched after failures of reproduce.

In this thesis, we presents REPFUZZER which enables deterministic reproduce for bugs found by fuzzer. It solves the problem with selective thread tracing and deterministic scheduler. With selective thread tracing, REPFUZZER can focus on only interesting thread in fuzzing context. Then REPFUZZER schedules the selected threads with deterministic scheduler and produces deterministic thread interleavings. Using REPFUZZER's scheduling at both fuzzing and reproduce phase, the fuzzer can reproduce bugs found at fuzzing phase. As a result, REPFUZZER shows its effectiveness for reproducing concurrency bugs with 15 real-world bugs, and some of the bugs require enormous times to be reproduced with non-deterministic kernel scheduling.

**keywords:** concurrency bug, kernel, fuzzing

**student number:** 2020-27942

# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>ii</b>
<b>List of Tables</b>	<b>iv</b>
<b>List of Figures</b>	<b>v</b>
<b>1 INTRODUCTION</b>	<b>1</b>
<b>2 BACKGROUND</b>	<b>5</b>
2.1 Fuzzing . . . . .	5
2.2 Concurrency Bug . . . . .	6
<b>3 DESIGN</b>	<b>7</b>
3.1 Design . . . . .	7
3.1.1 Selective Thread Tracing . . . . .	9
3.1.2 Deterministic Scheduler . . . . .	12
3.2 Implementation . . . . .	18
<b>4 EVALUATION</b>	<b>20</b>
4.1 Effectiveness of Deterministic Scheduler . . . . .	20
4.2 Statistics of Deterministic Scheduler . . . . .	23
4.3 Overhead of Deterministic Scheduler . . . . .	25

<b>5 DISCUSSION</b>	<b>27</b>
<b>6 RELATED WORKS</b>	<b>29</b>
6.1 Kernel Concurrency Bugs . . . . .	29
6.2 Reproducing Bugs . . . . .	30
<b>7 CONCLUSION</b>	<b>32</b>
<b>Abstract (In Korean)</b>	<b>37</b>

# List of Tables

4.1	Reproduce result for 15 concurrency bugs . . . . .	21
4.2	Comparison of thread interleavings found by deterministic scheduler and kernel scheduler . . . . .	24
4.3	Execution numbers with each fuzzer for 4 hours . . . . .	25



# List of Figures

3.1	Overall work flow of REPFUZZER. . . . .	7
3.2	Examples of operations with $SET_{\text{sched}}$ . . . . .	9
3.3	Example of how kernel threads are scheduled with thread order. . . .	15
4.1	Figure of CVE-2020-25656 [2], example of non-reproducible concurrency bugs. . . . .	23
4.2	Accumulated number of unique thread interleavings over execution. . .	24
4.3	Histogram of number of appearance during execution. . . . .	24

# Chapter 1

## INTRODUCTION

Reproduce of found bugs by kernel fuzzers becomes important for analyzing and patching the bugs. As the kernel fuzzing has been developed for years, more bugs have been discovered and reported by fuzzers. But the number of reported bugs exceeds the capability of developers' manual efforts. So, reproducing bug, which can provide rich information for analyzing bugs, becomes more important.

Therefore, kernel fuzzers normally consists with fuzzing phase and reproduce phase [17]. In the fuzzing phase, a fuzzer generates an input program, i.e., the syscall sequence, and executes it to explore kernel codes. When a bug occurs during the fuzzing phase, the fuzzer extracts a crash report and a fuzzing log, i.e., list of executed inputs. In the reproduce phase, the fuzzer tries to turn the fuzzing log into a reproducer, which is minimized inputs that can trigger the bug.

With stable reproducer, developers can utilize debugger for dynamic analyzing. Certain information, e.g., stack trace and memory state when crash occurs, can be offered with crash report and sanitizers. However, more diverse information, e.g., full trace of execution and memory states of any point, are offered by dynamic debugging. As a result, the reproduce phase enables dynamic debugging and produces richer information.

So, without the reproducing, the bug consumes much time to be patched or even

can be classified as an invalid bug. Because of the kernel's complexity, kernel bugs require enormous time to be analyzed without enough information. Furthermore, manual effort of developers is not enough for analyzing the bugs. In SyzScope [1], the authors said that the number of reporting bugs is much higher than patching the bugs. In the circumstance, the importance of the reproducing stands out.

What is worse, without a reproducer produced by reproduce phase, concurrency bugs can be impractical to be analyzed for two reasons. First, for many cases, the concurrency bug itself does not make any troubles. The bug is detected after the buggy state from the bug is used and makes harmful outcome, so the root cause of the bug is hard to be found. Second, the number of threads in kernel compounds the difficulty of analyzing concurrency bugs. In addition to syscall threads executed by a fuzzer, in kernel system, background threads are running as non-syscall context for asynchronous computation. The problem is these threads also hold the possibility to produce concurrency bugs [28]. Hence, search space for concurrency bug is excessive to be analyzed manually.

As a result, some concurrency bugs are ignored after a failure of bug analysis. For instance, the authors of RAProducer [21] reported CVE-2020-25656 [2] which was already found by Syzkaller [17], i.e., one of the most popular on-the-fly kernel fuzzers. But Syzkaller failed to make a reproducer for the bug and, therefore, developers failed to analyze the bug. So, the bug was regarded as an invalid bug until the authors reported the PoC(Proof-of-Concept) program to the developers.

But certain concurrency bugs are impractical to be reproduced because of non-deterministic scheduling of kernel. Thread interleavings produced by kernel scheduling is non-deterministic because of varying states. Since a kernel state when a thread is executed varies every time, kernel schedules threads differently depending on the state. Also, kernel is a layer which handles hardware directly, its thread scheduling is easily affected by hardware states, e.g., hardware interrupts. By varying kernel and hardware states, kernel scheduler produces different thread interleavings for same threads.

For reproducing concurrency bugs, deterministic scheduling is required since some bugs require uncommon thread interleavings to be triggered. Concurrency bugs can be triggered by subset of all possible thread interleavings. Depending on concurrency bug's race window, the size of buggy thread interleavings can be shrunk. Furthermore, certain concurrency bugs require combination of thread interleavings, and some combinations are impractical to be produced. Hence, even if a fuzzer finds bug with high computation despite of the difficulty, the buggy thread interleavings are hardly reproduced. ExpRace [20] shows that, without any proposed method in the thesis, some concurrency bugs cannot be exploited over 24 hours.

In this thesis, we propose REPFUZZER, fuzzing framework which can reproduce found bugs easily. The key idea of REPFUZZER is to make deterministic scheduler in the kernel and provide same scheduling with both fuzzing phase and reproduce phase. First, REPFUZZER inserts scheduling points for kernel codes so that the executions of kernel threads can be controlled. Second, REPFUZZER makes scheduling policy, e.g., duration of executing thread and next thread to be executed. Lastly, using above scheduling points and policy, REPFUZZER enforces serial execution for kernel threads.

However, some challenges derive from the contexts where REPFUZZER resides in, i.e., kernel and fuzzing. First, in kernel context, scheduling background threads, created to service asynchronous events, can cause huge performance overhead. Large number of threads are created by kernel and they could trigger concurrency bugs by sharing resources with other threads [28]. Second, as a fuzzer, REPFUZZER should be possible to find all the bugs. If REPFUZZER schedules the threads with limited policy, REPFUZZER could miss some concurrency bugs.

To solve the challenges, following features are added into design of REPFUZZER. First, REPFUZZER schedules background threads selectively so that it can not only reproduce concurrency bugs but also reduce the overhead. Fuzzers usually focuses on bugs which are triggered by fuzzing inputs. So, REPFUZZER also focuses on the threads which originates from fuzzing inputs, i.e., input syscalls. During execution of

input syscalls, REPFUZZER traces kernel threads created from the input syscalls and schedules within the threads.

Second, REPFUZZER schedules kernel threads based on scheduling token so that it can test diverse schedulings. REPFUZZER makes scheduling policy depending on given scheduling token and schedules the thread with it. Then a fuzzer can explore diverse schedulings by mutating scheduling token. Furthermore, REPFUZZER still generates deterministic scheduling by providing same scheduling token.

We implemented deterministic scheduler in Linux kernel and instrumented kernel codes to call our scheduling process. We also implemented ioctl interface and modified Syzkaller [17] to utilize our deterministic scheduler. We evaluated REPFUZZER with 15 real world concurrency bugs in Linux kernel system. REPFUZZER could reproduce bugs faster than Syzkaller with all target bugs. Furthermore, with many bugs Syzkaller could not reproduce over 24 hours, REPFUZZER reproduced within few seconds.

In summary, this work makes contributions as follow:

- We propose deterministic scheduling to reproduce concurrency bugs by providing same scheduling with same inputs, i.e., scheduling token and input syscalls.
- We propose selective thread tracing to reduce overheads from scheduling all the threads in kernel space.
- We propose scheduling token which enables diverse scheduling but still deterministic scheduling.
- We implemented REPFUZZER with Linux kernel and Syzkaller, and evaluated reproducing of 15 real world concurrency bugs. Even with bugs kernel scheduler could not reproduce, deterministic scheduler easily reproduced them.

## Chapter 2

### BACKGROUND

#### 2.1 Fuzzing

Fuzzing is practical testing technique to find bugs in software. In common, the fuzzing processes as follows: (i) generating random input for target program; (ii) executing the program with generated input; and (iii) checking the result of the program and repeating processes. The base idea of fuzzing is simple, but many researches proposed several techniques to improve fuzzing. One of the most general approach is coverage-guided fuzzing. Coverage-guided fuzzing prioritizes the input exploring new coverage so that it can explore diverse coverages and find more bugs.

Kernel fuzzing usually utilizes the syscalls as an fuzzing input [17]. The fuzzing input targets to explore code space of target program. In kernel, the easiest way to execute kernel code is calling syscall provided by the kernel. However, since some components, e.g., scheduling, are not handled by the fuzzer, some bugs are failed to be reproduced.

## 2.2 Concurrency Bug

Concurrency bug occurs between multiple threads or sometimes single thread with other context, e.g., signal handler or interrupt. Since the threads in concurrent program share same resources, e.g., memory, the states of resources can be different by how the threads use them. Unfortunately, certain states can be unintended by programmer and cause harmful results, e.g., Denial-of-Service (DoS), Remote Code Execution (RCE) and Local Privilege Escalation (LPE) [26, 28, 29, 30].

For concurrency bugs, thread interleaving determines whether the bugs to be triggered or not [30, 31]. The states of shared resources are determined by the order of sharing resources in several threads. Since the concurrency bugs are derived from the resource states, the interleavings of threads' executions decide the bug to be triggered. But scheduling methods commonly used in systems do not guarantee the deterministic replay of thread interleavings. As a result, concurrency bugs normally have challenges to be reproduced.

# Chapter 3

## DESIGN

### 3.1 Design

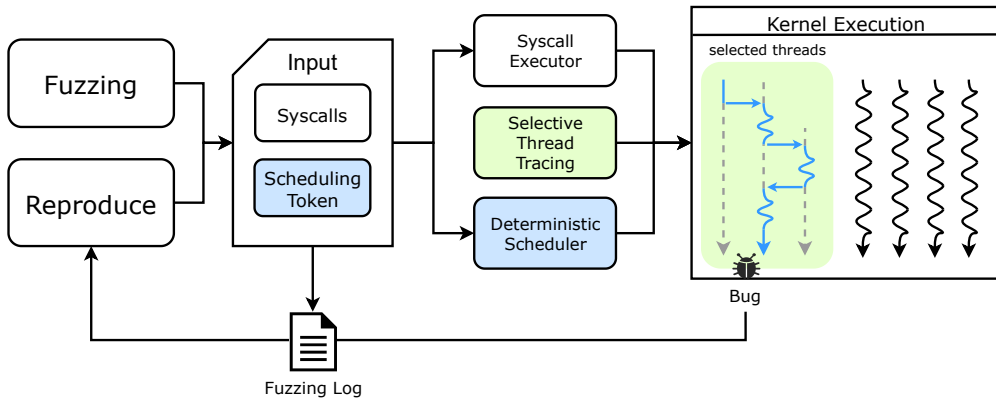


Figure 3.1: Overall work flow of REPFUZZER.

In this section, we describe high-level design and key points of REPFUZZER. To replay concurrency bugs, our system focuses on reproducing thread interleavings occurred during fuzzing. The overall workflow of REPFUZZER is shown in Figure 3.1. REPFUZZER provides a framework which selects kernel threads to be scheduled and enforces deterministic scheduling for selected threads. With both fuzzing and reproduce phase, a fuzzer executes an input upon the framework, so that the thread inter-



leaving of fuzzing phase can be replayed at reproduce phase.

**Selective Thread Tracing.** In the kernel, there are background threads running with non-syscall context [28] and REPFUZZER also schedules these threads but in a selective way. In common cases, it is hard to choose which background threads to trace. However, in fuzzing context, we only have interest on results from inputs, so we can set policy to select which threads to schedule.

**Deterministic Scheduler.** The key component of REPFUZZER is deterministic scheduler. The deterministic scheduler interferes kernel threads and enforces customized scheduling policy. So, with the deterministic scheduler, thread interleavings can be easily replayed. Also, not to limit possible thread interleavings, REPFUZZER's scheduler receives scheduling token as an input. Scheduling policy varies with the given token but the same policy is produced under the same token.

**Fuzzing Phase.** Using the scheduler, a fuzzer can execute syscalls with deterministic scheduling. The fuzzer generates an input which consists of syscalls and scheduling tokens. The syscalls are usually used as an input in kernel fuzzing to explore kernel codes and find bugs. The tokens are handed to the deterministic scheduler and used for scheduling kernel threads. Both syscalls and tokens are delivered to syscall executor and deterministic scheduler. With the given input, an execution result including thread interleavings is produced.

**Repro Phase.** When a bug occurs during fuzzing, a reproducer receives a fuzzing log and reproduces the bug. The log contains inputs which our fuzzer has executed, including scheduling tokens. So, the reproducer parses a buggy input from the log and executes the input. As the fuzzer does, syscalls are sent to a syscall executor and tokens are sent to a deterministic scheduler. With the same syscalls and scheduling tokens, the same execution result is produced and, as a result, the bug occurs again.

In following paragraphs, we describe key designs of above framework. Section 3.1.1 describes how REPFUZZER selects background threads to be scheduled. Next Section 3.1.2 shows how deterministic scheduler is designed to reproduce thread in-

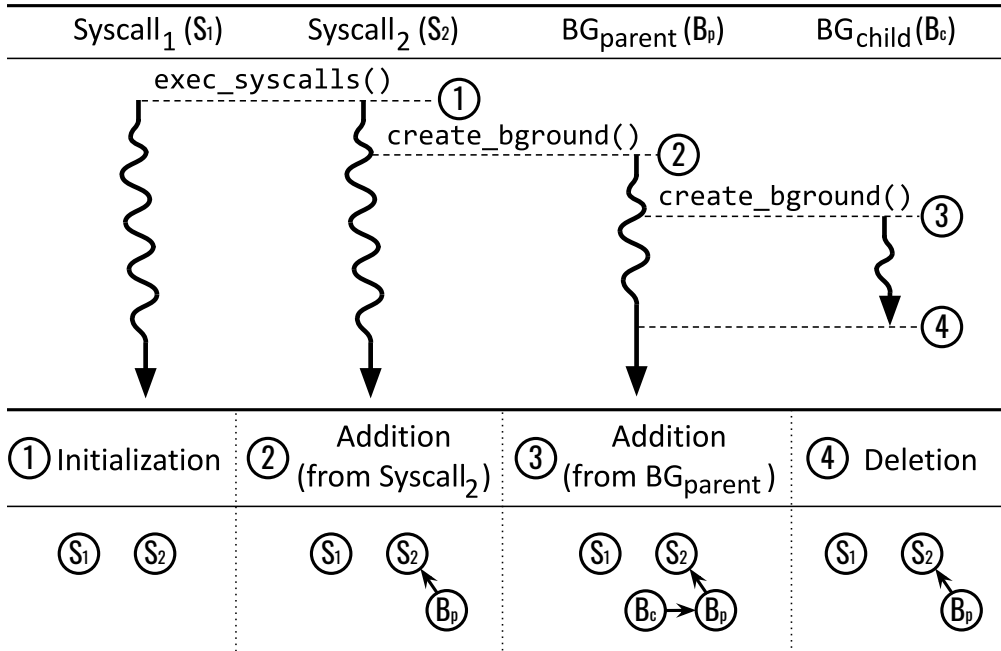


Figure 3.2: Examples of operations with `SETsched`.

terleavings with the chosen threads.

### 3.1.1 Selective Thread Tracing

**Challenge: Difficult to Trace All Threads.** A bug can occur in background threads [28] and `REPFUZZER` needs to control these threads. In kernel, there are many threads that run in background, i.e., non-syscall context. Most of the background threads exist to process works asynchronously, e.g., `kthread`, `workqueue` and `rcu`. So, other threads invoke background threads by registering a job to be processed, e.g., start routine of new thread or destructor for `rcu`. During processing jobs, background threads run concurrently and share memory with other kernel threads. The challenge is scheduling all of background threads brings huge overhead.

**Tracing Threads Originating from Syscalls.** To solve this challenge, `REPFUZZER` schedules only selected threads. For common cases, it is hard to choose which threads

to select, since bug can exist in anywhere. However, fuzzer's only interest is bugs caused from its inputs. Fuzzer focuses to find input that can be driven to bug, so it usually ignores bugs triggered by unknown events. In this regard, we can select threads that originate from input, i.e., syscall thread.

To schedule threads selectively, REPFUZZER classifies threads into two types according to the thread where they originate. If they originate from input syscall thread, they are classified as  $Thr_{interest}$ , and the rest are referred as  $Thr_{non-interest}$ . At the beginning, REPFUZZER holds input syscall threads as  $Thr_{interest}$ . Whenever a background thread is invoked, REPFUZZER checks whether the thread originates from input syscall and selects all  $Thr_{interest}$ . With above classification, REPFUZZER can focus on  $Thr_{interest}$  and reduce overhead of scheduling.

**Rules with Background Thread Selection.** Deciding whether a thread originates from input cannot be simply processed by checking a parent thread with input syscall threads. There can be a case that a background thread invokes other background thread, e.g., registering new rcu callback in a workqueue thread. Therefore,  $Thr_{interest}$  can be recursively derived from other  $Thr_{interest}$ .

For the reason, REPFUZZER traces  $Thr_{interest}$  and maintains  $SET_{sched}$  which consists of current  $Thr_{interest}$ . When a background thread is invoked, REPFUZZER checks whether its parent thread belongs to  $SET_{sched}$ , i.e., set of currently active  $Thr_{interest}$ . Recursively, if the parent thread is derived from input syscall, the child is also derived from input syscall. So, the new background thread is added into  $SET_{sched}$  as a child. Thus,  $SET_{sched}$  can be understood as a directed graph composed with  $Thr_{interest}$  as a node.

A directed graph  $SET_{sched}$  is managed with following three operations. Figure 3.2 depicts how each operation is processed.

- **Initialization:**  $SET_{sched}$  starts with threads executing input syscalls.
- **Addition:** When a thread in  $SET_{sched}$ , i.e.,  $Thr_{interest}$ , creates a new background

thread, the thread is added to  $SET_{\text{sched}}$ . The thread is added as a leaf node and its outgoing edge points its parent thread.

- **Deletion:** When  $Thr_{\text{interest}}$  finishes its job and exits,  $Thr_{\text{interest}}$  is removed from  $SET_{\text{sched}}$ . Deletion is needed since REPFUZZER schedules the threads in  $SET_{\text{sched}}$ . The existence of finished thread affects REPFUZZER's scheduling and, therefore,  $SET_{\text{sched}}$  contains only active  $Thr_{\text{interest}}$ .

**Determinism of Thread Selection.** Thread selection is deterministic from fuzzing input for two reasons. First, background threads added to  $SET_{\text{sched}}$  are deterministic from input syscalls. With same input syscalls executed, same background threads are created by input syscall threads. Also, from same background threads, same child background threads are created. As a result, the components of  $SET_{\text{sched}}$  are deterministic.

Second, additions and deletions of  $SET_{\text{sched}}$  are deterministically place in scheduling, as both are executed by  $Thr_{\text{interest}}$ . The creation of the background thread occurs during execution of parent thread. Since parent thread of  $Thr_{\text{interest}}$  is also  $Thr_{\text{interest}}$ , the addition of a thread into  $SET_{\text{sched}}$  occurs in  $Thr_{\text{interest}}$ . Also,  $Thr_{\text{interest}}$  is removed from  $SET_{\text{sched}}$  when its execution is over, i.e., deletion of  $Thr_{\text{interest}}$  occurs in  $Thr_{\text{interest}}$ . Hence, both operations are executed by  $Thr_{\text{interest}}$  and deterministically scheduled by REPFUZZER. In conclusion, since both components and operations are deterministic,  $SET_{\text{sched}}$  is deterministic from input syscalls.

**Handling Different Types of Background Thread.** In kernel system, various kinds of background threads are serviced. When a thread invokes a background thread, the parent thread creates and sends a job which indicates a work to be processed, e.g., a routine executed by a new thread. According to the number of handled jobs, the background threads can be categorized into two types. First type is  $BG_{\text{single}}$  which spawned by other threads with a single job, e.g.,  $kthread$ .  $BG_{\text{single}}$  processes a single job and exits when the job is done. Therefore, once the thread is decided as either  $Thr_{\text{interest}}$  or  $Thr_{\text{non-interest}}$ , it would not be changed. As a result, REPFUZZER can

handle the thread without any difficulty.

Another type is  $BG_{\text{multi}}$  which executes as daemon thread and keeps receiving the jobs to execute, e.g., workqueue and rcu. In  $BG_{\text{multi}}$ , a single thread can receive various jobs from various threads, including both  $Thr_{\text{interest}}$  and  $Thr_{\text{non-interest}}$ . Therefore, the thread cannot be decided between  $Thr_{\text{interest}}$  and  $Thr_{\text{non-interest}}$ , which makes a challenge in handling  $BG_{\text{multi}}$ . If the thread is treated as  $Thr_{\text{non-interest}}$ , REPFUZZER misses executions originate from input syscall. It can produce a case that REPFUZZER fails to replay a bug deterministically.

Meanwhile, if REPFUZZER handles the thread as  $Thr_{\text{interest}}$ , it occurs not only extra overhead but also non-determinism to the scheduling. When a job is created by  $Thr_{\text{interest}}$  whose execution is controlled by REPFUZZER, the creation places deterministically in scheduling. On the other hand,  $Thr_{\text{non-interest}}$  executes independently with REPFUZZER's scheduling and the creation of a job happens non-deterministically for REPFUZZER. Therefore, the order of processing jobs can vary even with same scheduling token.

As a solution, REPFUZZER considers  $BG_{\text{multi}}$  as a set of individual threads. Although  $BG_{\text{multi}}$  handles multiple jobs, each job is processed sequentially. So,  $BG_{\text{multi}}$  cannot be both  $Thr_{\text{interest}}$  and  $Thr_{\text{non-interest}}$  at same time. Hence, REPFUZZER can divide processes of jobs and regard each one as an individual thread. Then each thread can be classified by the job currently dealing with and only  $Thr_{\text{interest}}$  can be scheduled by REPFUZZER.

### 3.1.2 Deterministic Scheduler

**Challenge: Non-deterministic scheduling.** The main contribution of REPFUZZER is helping a fuzzer to reproduce concurrency bugs deterministically. The main reason why concurrency bug is hard to be reproduced is that kernel's scheduling is non-deterministic. Even if same input syscalls are executed, thread interleavings from scheduler differs every execution. As a result, when reproducing concurrency bugs, the

fuzzer has to expect the buggy thread interleaving to be occurred again. Unfortunately, some concurrency bugs require uncommon interleaving to be triggered. Therefore, REPFUZZER resolves non-determinism of kernel thread scheduling by adopt deterministic scheduler.

To enable deterministic scheduling, REPFUZZER applies two changes for kernel threads. First, REPFUZZER inserts scheduling point into kernel codes so that kernel threads can be controlled by our deterministic scheduler. Second, it enforces serial execution for target kernel threads, i.e., threads in  $SET_{\text{sched}}$ .

With the changes, the deterministic scheduler produces thread interleavings deterministic from a fuzzer's input. When the fuzzer executes its input, i.e., scheduling token and syscall, the deterministic scheduler receives the token and generates streams used as a scheduling policy. During scheduling, whenever a scheduled thread meets scheduling point, the scheduling policy decides whether to keep executing or wake other thread.

**Scheduling Point.** REPFUZZER interferes and schedules threads in  $SET_{\text{sched}}$  by inserting scheduling point into kernel codes. At the scheduling point, REPFUZZER checks whether a current thread is in  $SET_{\text{sched}}$  or not, so it can enable  $Thr_{\text{non-interest}}$  to execute independently. Therefore, if the thread is not included in  $SET_{\text{sched}}$ , it returns immediately from the scheduling point. Meanwhile, for threads in  $SET_{\text{sched}}$ , the deterministic scheduler can continue the current thread or hand over scheduling to other thread. When the scheduler hands over the scheduling, a thread interleaving occurs at scheduling point.

REPFUZZER inserts scheduling point at every memory instructions to minimize overhead without missing concurrency bugs. As many scheduling points being inserted, the overhead of scheduling is increased since extra computation is executed at scheduling point. Meanwhile, if REPFUZZER targets too small set of instructions, some thread interleavings, including buggy interleavings, can be neglected. Therefore, REPFUZZER inserts scheduling points at every memory instructions, so thread inter-

leavings between all memory accesses can be tested. Memory access is important for concurrency bug since it affects shared resource between threads. In this regard, stack memory is not shared between threads, so REPFUZZER ignores stack memory access.

**Serial Execution.** For deterministic scheduling, REPFUZZER enforces serial execution for threads in  $SET_{\text{sched}}$  using scheduling point. At the beginning, REPFUZZER executes a single syscall thread in input syscall threads and makes other threads to wait for REPFUZZER's scheduling. Whenever a running thread in  $SET_{\text{sched}}$  hands over a scheduling, the thread waits for next scheduling rather than keeps executing. As a result, threads in  $SET_{\text{sched}}$  are executed as if they are running in a single thread. It enables deterministic commits of thread interleaving under same scheduling policy.

In common systems, executing threads serially can be defect since it fails to utilize the merit of multi-core system. However, from fuzzer's perspective, performance of single instance could not be important, if it uses less resources. The fuzzer can increase the number of fuzzing instances within same resource limitation. Therefore, what matter for the fuzzer is the performance with same resource utilization.

**Token-based Scheduling Policy.** The challenge in deterministic scheduling is that it can limit possible thread interleavings that fuzzer can explore. For fuzzer, one of the most important objective is exploring as many states to find bugs. However, if deterministic scheduler restricts possible thread interleavings with same scheduling policy, some concurrency bugs cannot be found. Although helping fuzzer to find more bugs is not REPFUZZER's goal, it shouldn't disturb the fuzzer.

So, deterministic scheduler gets scheduling token as an input. It generates different scheduling policies depending on given token. So, with same syscalls, a fuzzer can explore diverse states of thread interleavings by differing the token. Meanwhile, the token is part of fuzzing input, so REPFUZZER still generates same execution result from same input, which is most important to offer deterministic reproduce of bug.

For generating scheduling policy depending on token, REPFUZZER uses stream generator which generates number stream depending on given token. REPFUZZER

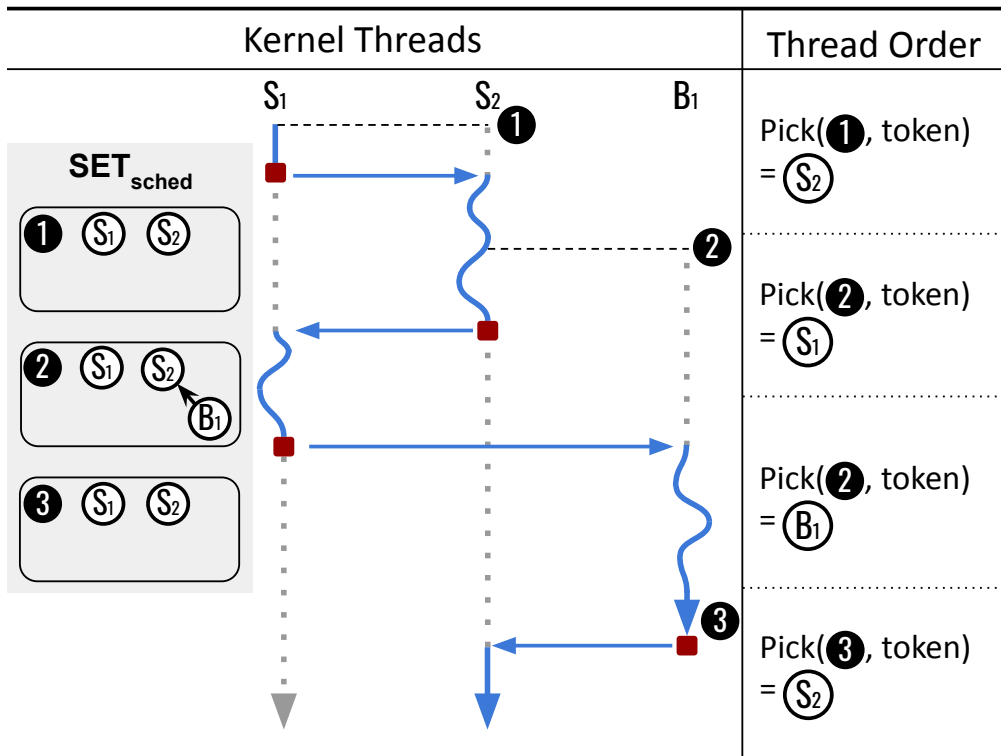


Figure 3.3: Example of how kernel threads are scheduled with thread order.



composes scheduling policy with execute counts and thread orders. The execute count indicates how many scheduling points the thread should execute and the thread order indicates the next thread to be scheduled. Thus, by generating streams of execute count and thread order, REPFUZZER is able to make scheduling policy depending on the token.

The execute count is assigned to the thread and determines its progress. When a thread is scheduled by deterministic scheduler, the execute count is newly assigned to the thread. During execution, if the thread meets scheduling point, the assigned count is decreased one by one. At some point, the count becomes zero and the thread hands over scheduling to other thread.

When a thread hands over scheduling to other thread, thread order is used to decide target thread. As discussed at Section 3.1.1, the threads in  $SET_{\text{sched}}$  keep changing as execution continues. For example, in Figure 3.3,  $SET_{\text{sched}}$  differs through ①, ② and ③.  $SET_{\text{sched}}$  is initialized at ① with two syscall threads, i.e.,  $S_1$  and  $S_2$ , and new background thread  $B_1$  is added at ②. Finally, at ③,  $B_1$  is removed from  $SET_{\text{sched}}$ , since it finishes its execution and exits. As  $SET_{\text{sched}}$  changing, thread order is picked using current  $SET_{\text{sched}}$  and scheduling token.

During execution, deterministic scheduler schedules new thread in two cases: (i) when current thread's execute count becomes zero; and (ii) when current thread finishes its execution. In Figure 3.3, red squares indicate the point where the change of current thread occurs and show both cases of thread change. All red squares, except last one, show when thread's execute count becomes zero. With the last case, thread  $B_1$  exits, so new thread, i.e.  $S_2$ , is picked and scheduled by deterministic scheduler.

To sum up, the work flow of deterministic scheduling is as follows: (i) when input syscalls are called, the scheduler executes one syscall thread and assigns execute count for the thread; (ii) whenever the thread meets scheduling point, it decrease its execute count; (iii) if the execute count becomes zero or the thread exits, the scheduler picks next thread in  $SET_{\text{sched}}$ ; (iv) the scheduler assigns new execute count for the thread; and

(v) steps (ii) through (iv) are repeated until  $SET_{\text{sched}}$  becomes empty, i.e., all threads are finished.

**Timeout Handling.** During scheduling, currently scheduled thread can become non-runnable state because of dependency for other threads in  $SET_{\text{sched}}$ . For synchronization or atomicity, threads often wait for certain conditions, e.g., callbacks or mutex. If current thread waits for the condition which can be solved only by other thread in  $SET_{\text{sched}}$ , current thread cannot continue its execution. But dependencies are present in everywhere in large kernel space. Furthermore, some are implemented manually by programmer. As a result, it is impractical to handle all the cases.

To practically prevent halting, REPFUZZER invokes timeout handler when it do not schedule other threads for a while. When timeout handler is called, REPFUZZER first makes current thread's execute count to zero, preventing parallel execution. Then it picks next thread to be scheduled and continues its scheduling. The cause of timeout is dependency between threads in  $SET_{\text{sched}}$  and, since the threads are controlled by deterministic scheduler, the timeout is also invoked deterministically.

**Thread Interleavings and Scheduling Policy.** Theoretically, all possible thread interleavings between memory instructions can be made. In REPFUZZER, thread interleaving is made by transfers between scheduling points of two threads. The scheduling points where the transfer occurs are determined by values of execute count and thread order. Since the values are not fixed, if scheduling token generates the required values, all possible transfer between scheduling points can occur.

The distribution of execute count and thread order is important since it affects the possibilities of thread interleavings. REPFUZZER picks both execute counts and thread orders within uniform distribution. For thread orders, REPFUZZER picks the thread uniformly in  $SET_{\text{sched}}$ . For execute counts, REPFUZZER makes a range by picking maximum and minimum number and assigns the range for each threads. During execution, execute count is picked from current thread's assigned range.

REPFUZZER assigns the range of execute count differently for each thread, to even

the possibilities of thread interleavings. In REPFUZZER’s scheduling, the thread interleaving is made between the progress of two threads when a transfer between threads occurs. The progress of a thread means the sum of execute counts uniformly picked from the range. If execute count is picked from same range for all threads, each thread’s distribution of the sum, i.e., the progress of thread, will correlate with others. It means the possibilities of certain interleavings are much higher than others. As a result, to reduce correlation of the threads, REPFUZZER assigns different execute count range for each thread.

## 3.2 Implementation

The implementation of REPFUZZER is composed with kernel level and user level implementations. We implemented our scheduling framework, e.g., selective thread tracing and deterministic scheduler, at kernel space. Since the design of REPFUZZER requires little changes with fuzzer program, we implemented the fuzzer at user level.

**Kernel Level.** We implemented REPFUZZER based on Linux 5.15.0-rc4. We built our scheduling framework based on KCSAN’s [18] implementation. KCSAN instruments memory accesses by applying thread sanitizer [19] at compile time and overwriting its functions, e.g., `__tsan_read4`. So, instead of building instrumentation framework from the beginning, we implemented REPFUZZER modifying KCSAN. For selective thread tracing, we implemented operations of `SETsched`, e.g., addition and deletion, and inserted the operations into the points where background threads are handled.

We also implemented `ioctl` interface for a fuzzer to use our scheduling framework. It provides following APIs: `SCHED_START`; `SCHED_TRACE`; `SCHED_UNTRACE`; and `SCHED_END`. `SCHED_START` is called at the beginning, sending scheduling token and the number of syscalls to be called for our scheduler. `SCHED_TRACE` is called right before execution of syscall so that our scheduler can trace input syscall thread. After the syscall execution finishes, `SCHED_UNTRACE` informs the end of syscall. Finally,

SCHED\_END notifies the end of scheduling and cleans up states of the scheduler.

**User Level.** For a kernel fuzzer, we implemented it based on Syzkaller [17], most commonly used on-th-fly kernel fuzzer. For our design, we added scheduling token as a fuzzing input so that it can be managed and used by the fuzzer. We modified syscall executor of Syzkaller so that it can schedule syscalls to be executed concurrently. Lastly, we added calling ioctl provided by our scheduling framework.

## Chapter 4

### EVALUATION

In this section, we evaluate REPFUZZER with three aspects. First, we show effectiveness of REPFUZZER with the real world bugs (Section 4.1). Second, we estimate capability of REPFUZZER’s scheduling policy compared to kernel scheduling (Section 4.2). Lastly, we measure performance overhead of REPFUZZER (Section 4.3).

**Experiment Setup.** All the evaluations were performed on Intel(R) Xeon(R) Silver 4214R CPU @ 2.40GHz with 512 GB of RAM. As a host operating system, we used Ubuntu 20.04.1 LTS with kernel version 5.4.0. For running guest kernel, Qemu 5.2.0 was used as hypervisor.

#### 4.1 Effectiveness of Deterministic Scheduler

To evaluate effectiveness of REPFUZZER, we tested 15 concurrency bugs in Linux kernel. We first searched patching commits for the target bugs. With the patch, we built different versions of kernels so that each kernel contains each vulnerability. The patch was applied on the kernel where we implemented REPFUZZER. Then we ran a fuzzer to find inputs for the target bugs, since REPFUZZER requires scheduling token for reproducing the bugs. With fuzzing, we enabled for subset of syscalls that can trigger the bugs, since the evaluation is not about fuzzing ability.

Table 4.1: Reproduce result for 15 concurrency bugs

Vulnerability	Reproduce Time	
	Deterministic Scheduling	Kernel Scheduling
CVE-2020-25656 [2]	✓ (< 4 sec)	× (> 24 hours)
CVE-2019-6974 [3]	✓ (< 27 sec)	× (> 24 hours)
CVE-2019-1999 [4]	✓ (< 12 sec)	✓ (< 10 hours)
CVE-2017-2636 [5]	✓ (< 2 sec)	× (> 24 hours)
CVE-2017-15265 [6]	✓ (< 1 sec)	× (> 24 hours)
20f2e4c2 [7]	✓ (< 1 sec)	✓ (< 7 min)
32d3182c [8]	✓ (< 8 sec)	× (> 24 hours)
4842e98f [9]	✓ (< 1 sec)	× (> 24 hours)
4b848f20 [10]	✓ (< 11 sec)	✓ (< 20 sec)
6c605f83 [11]	✓ (< 1 sec)	× (> 24 hours)
6cd1ed50 [12]	✓ (< 4 sec)	✓ (< 24 sec)
7311d665 [13]	✓ (< 1 sec)	× (> 24 hours)
a6361f0c [14]	✓ (< 2 sec)	× (> 24 hours)
da1b9564 [15]	✓ (< 9 sec)	× (< 12 hours)
e20a2e9c [16]	✓ (< 1 sec)	× (> 24 hours)

After the input was found, we executed the input repeatedly and measured the spent time until the bug was reproduced. The input was executed on two circumstances: (i) with REPFUZZER's deterministic scheduling; and (ii) with kernel scheduling.

Table 4.1 shows the result of 15 concurrency bugs. For kernel scheduler, most of the bugs require much time to be reproduced. In contrast, with deterministic scheduling, the bugs were reproduced within few seconds. As a result, REPFUZZER effectively helps bug reproduce once the bug is found by a fuzzer.

Especially, 10 bugs cannot be reproduced over 24 hours without our scheduler, which can be regarded as non-reproducible bugs. Using REPFUZZER is gain for those bugs, despite of overhead in fuzzing phase. REPFUZZER holds overhead by scheduling the kernel threads at fuzzing phase and, meanwhile, it can reproduce bugs with minimal executions. If all concurrency bugs can be reproduced within several minutes, the bugs are better to be fuzzed without REPFUZZER. However, many bugs are failed to be reproduced over 24 hours and REPFUZZER is extremely helpful for those bugs. Without reproduce, developers cannot be provided enough information to analyze the bug. CVE-2020-25656 [2] is the example of the bug which was found by Syzkaller [17] but failed to be patched since the bug cannot be reproduced.

**Analysis of Non-reproducible Concurrency Bugs.** While certain bugs were easily reproduced without deterministic scheduler, the others demanded much time or even failed over 24 hours. We looked into those bugs and found the reason of non-reproducibility. Concurrency bugs require certain buggy thread interleavings and, according to required thread interleavings, we can group the bugs into two types. The bugs require only single thread interleaving (type 1) or combination of some thread interleavings (type 2), and non-reproducible concurrency bugs belong to the latter.

In type 2 bugs, the bugs become non-reproducible bugs with following conditions: (i) short race window; and (ii) equal or larger number of inclusive instructions. CVE-2020-25656 [2] (in Figure 4.1) is one of non-reproducible concurrency bugs. In the figure, the numbers (① ~④ ) indicates the buggy execution order. So, ① and ④

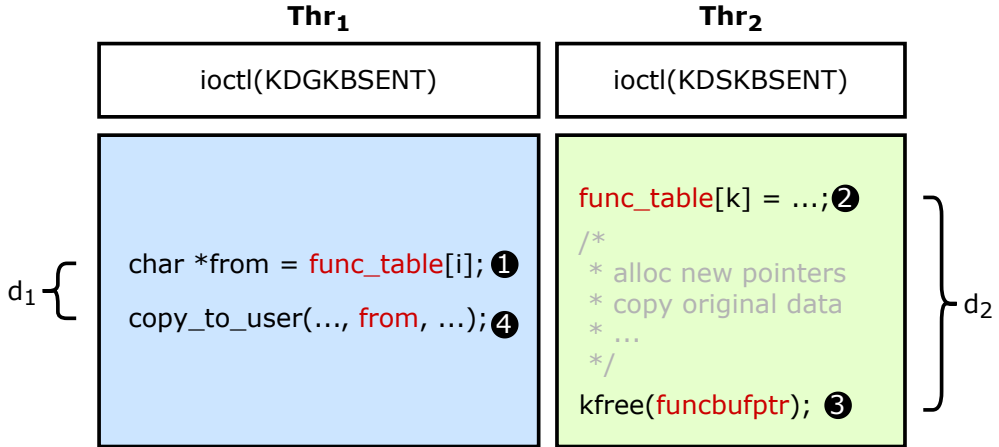


Figure 4.1: Figure of CVE-2020-25656 [2], example of non-reproducible concurrency bugs.

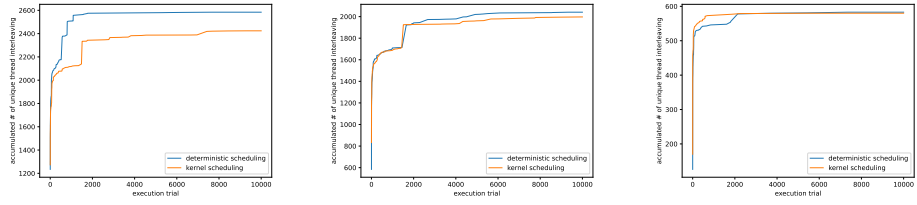
consist the race window and instructions from ② to ③ become inclusive instructions. The length of race window, i.e.,  $d_1$ , is short and the length of inclusive instructions, i.e.,  $d_2$  is much larger than race window. With the case, the buggy execution order is impractical to be produced by kernel scheduler. Hence, the bug is hardly reproduced with the kernel scheduling.

## 4.2 Statistics of Deterministic Scheduler

To estimate REPFUZZER’s scheduling ability, we ran syscalls with different scheduling token and collected thread interleaving. In Section 3.1.2, we emphasized the importance of scheduling ability. Since REPFUZZER is designed in the fuzzing context, REPFUZZER shouldn’t disturb the fuzzer for finding bugs. Hence, if REPFUZZER produce insufficient thread interleavings, some concurrency bugs cannot be found by the fuzzer.

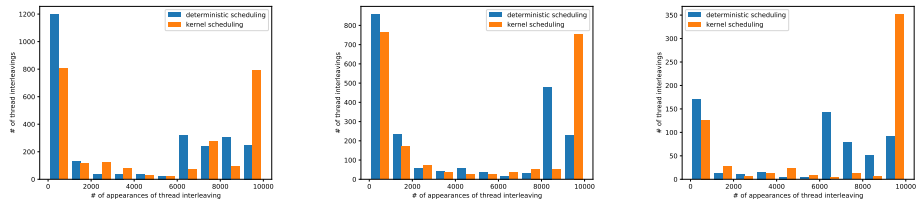
For testing program, we chose PoCs of 3 bugs evaluated in Table 4.1. With PoCs, we ran each program for 10,000 times with both deterministic scheduling and kernel





(a) PoC of CVE-2019-6974 [3]      (b) PoC of 4b848f20 [10]      (c) PoC of CVE-2020-25656 [2]

Figure 4.2: Accumulated number of unique thread interleavings over execution.



(a) PoC of CVE-2019-6974 [3]      (b) PoC of 4b848f20 [10]      (c) PoC of CVE-2020-25656 [2]

Figure 4.3: Histogram of number of appearance during execution.

Table 4.2: Comparison of thread interleavings found by deterministic scheduler and kernel scheduler

Vulnerability	Number of thread interleavings		
	Common	Unique in Deterministic Scheduling	Unique in Kernel Scheduling
CVE-2019-6974 [3]	2421	163	3
4b848f20 [10]	1977	64	20
CVE-2020-25656 [2]	580	3	0

scheduling. For deterministic scheduler, we changed scheduling token every time so that diverse thread interleavings can be produced. For collecting thread interleavings, we modified instrumenting codes to record instruction pointer, thread id and accessing memory address. With the information, we extracted thread interleavings between memory accesses for each execution. For scalability issue, we only collected thread interleavings of memory instructions which access same memory.

Figure 4.2 shows statistics of the result thread interleavings. Graphs illustrate the accumulated number of unique thread interleavings explored by each scheduler as execution maintains. In all cases, deterministic scheduler explores more unique interleavings than kernel scheduler. Also, deterministic scheduler covers most of the thread interleavings explored kernel scheduler. In Table 4.2, deterministic scheduler covers most of thread interleavings explored by kernel scheduler. Especially, with CVE-2020-25656 [2], it covers all kernel scheduler’s interleavings.

Furthermore, deterministic scheduler explored diverse thread interleavings with more even distributions than kernel scheduler. Figure 4.3 is a histogram of number of appearance while PoCs were executed. With kernel scheduler, i.e., orange bars, last bin shows high data. It means that many thread interleavings were appeared in most execution, i.e., same interleavings were repeatedly explored. Meanwhile, deterministic scheduler produced lower data in last bin of histograms.

### 4.3 Overhead of Deterministic Scheduler

Table 4.3: Execution numbers with each fuzzer for 4 hours

	Syzkaller [17]	REPFUZZER	
		Syscall	Syscall + Background
Execution	357743 ( $\times 1.00$ )	296475 ( $\times 1.20$ )	61755 ( $\times 5.78$ )

We checked overhead of deterministic scheduler with fuzzing throughput under

same resource limitation. We ran Syzkaller [17] and REPFUZZER with 8 instances of VMs with guest kernel where deterministic scheduling was implemented. Since REPFUZZER serializes traced threads' executions, it uses less CPU resource than kernel scheduler. Thus, for fair comparison, we limited fuzzers to use only 4 cores using cgroups.

Table 4.3 shows execution numbers with 4 hours of fuzzing. When REPFUZZER scheduled syscall threads only, it showed  $\times 1.2$  of fuzzing throughput overhead. As we limited CPU resource and maximized utilization of it, we could remove overhead from serial execution. The overhead appeared in the table was from instrumented scheduling codes, e.g., managing execute count and picking next thread.

When it comes to REPFUZZER's selective thread tracing, the throughput was 5.78 times slower than Syzkaller. The overhead was mainly from dependency between jobs in  $BG_{\text{multi}}$ . For  $BG_{\text{multi}}$ , REPFUZZER regards each job as individual thread. But since jobs cannot run until completion of other jobs enqueued in same queue earlier, the timeout handler of deterministic scheduler occurs every time non-runnable job is scheduled.

## Chapter 5

### DISCUSSION

**Unstable Kernel Execution Coverage.** In kernel system, certain branch executes differently by system states and, as a result, execution coverage of kernel thread could differ with each execution. Since the kernel services without refreshing its memory states, the execution coverage could differ even if same syscall is called. Also, kernel is a layer which directly communicates with hardware. So, the execution coverage can be affected by hardware states which is hard to be controlled.

The varying execution coverage can disturb REPFUZZER from reproducing concurrency bugs. REPFUZZER produces deterministic scheduling policy and applies it to incoming executions of target kernel threads. But if executions are changed, the result thread interleavings produced by REPFUZZER's policy can be affected. As a result, if the buggy interleaving disappears because of different execution, the concurrency bug could fail to be triggered.

However, we decided that the issue is not necessary to be handled. In evaluation, REPFUZZER could reproduce the target bugs within few executions and required only dozens of executions with most severe case. Since bugs can be reproduced without severe problem, REPFUZZER do not have to hold extra overhead, handling unstable kernel execution coverage, at run time.

**Unhandled Kernel Components.** Although REPFUZZER handles diverse non-syscall

context executions, some kernel components are not handled by REPFUZZER. The components handled by REPFUZZER are explicitly created in parent thread so that REPFUZZER can determine whether it should trace. However, with some components, they are invoked implicitly and cannot be decided whether to be traced. For instance, if kernel thread uses hardware, hardware interrupt context will be called. But the context cannot be decided to be traced, since the processes until interrupt is called occur in external from the kernel. But, if REPFUZZER can be assisted from hypervisor, more components can be handled by REPFUZZER.

## Chapter 6

### RELATED WORKS

#### 6.1 Kernel Concurrency Bugs

**Bug Testing.** KRACE [28] is fuzzing framework which targets data race in file system. KRACE introduces new metrics, referred as alias-coverage, to guide fuzzer to race bugs. Also, it injects random delays in memory instructions to manage kernel thread scheduling. Unfortunately, as the authors said, KRACE fails to reproduce found bugs deterministically since it cannot handle all kernel components because of its overhead. In contrast, REPFUZZER handles kernel threads with selective way to solve overhead issues so that it can reproduce found bugs deterministically.

Razzer [29] and Snowboard [30] are kernel fuzzers targeting concurrency bugs. Razzer applies static analysis to find possible racy instruction pairs and fuzzing input syscalls that can trigger the pairs. Snowboard clusters instructions sharing memory region during runtime, and tests diverse thread interleavings with the information. Razzer and Snowboard can reproduce found concurrency bugs easily, but both works are highly restricted to concurrency bugs. Also, they commonly holds high overhead from gathering information and testing thread interleavings under same input syscalls. Such design makes it inefficient for finding other types of bugs. Meanwhile, REPFUZZER does not require any restriction to a fuzzer, so it can support for general

bugs.

SKI [26] and RASProducer [21] are frameworks stress testing given program with different kernel thread interleavings. SKI pins each kernel thread to logical CPU in hypervisor so that it can serialize and schedule the execution of the CPU, i.e., kernel thread, with PCT algorithm [27]. Meanwhile, to reproduce data race from given program, RASProducer gathers candidate race pairs from dynamic analysis of the program and tests each pair with inserting breakpoint. The works can be used to reproduce concurrency bugs from a fuzzer, but some problems exist. First, fuzzer's crash log contains syscalls irrelevant to found bugs. SKI and RASProducer is based on stress testing and the required execution number increases by the size of crash logs. Second, the tools cannot decide whether the bug can be reproduced with the tool. In kernel, the issues other than concurrency can make unstable reproduce of bugs, e.g., external hardware states. But the tools cannot know why the program finished without any crash and, therefore, need to keep executing the program.

## 6.2 Reproducing Bugs

**Record and Replay.** For reproducing bugs, one of the most significant system is record and replay [22, 24, 25]. Record and replay system records non-deterministic events at runtime and, when a bug occurs, replays the bug with the recorded information. Some previous works [23] tried to implement record and replay system covering for whole system, including kernel. But because of its high overhead in both performance and space, it is hard to be used in context of kernel fuzzing. Especially, a fuzzer executes high throughput of syscalls, so the space overhead can be critical for the fuzzer.

Among the record and replay systems, tsan11rec [22], which targets for user programs, shares similarity with REPFUZZER. For recording thread interleavings, it instruments atomic instructions in program and applies between two strategies. It can

simply records the order of instrumented instructions or schedules the program based on seed number and replays with the number, which is similar to our approach. But the design of tsan11rec cannot provide deterministic reproduce of data race, so it was implemented upon race detector which makes huge overhead of both performance and memory. Also, unlike REPFUZZER, tsan11rec's scheduling policy is too simple to explore diverse thread interleavings. Since tsan11rec is based on record and replay system, exploring diverse thread interleavings is not tsan11rec's concern. Meanwhile, REPFUZZER is built for a fuzzer and should be design not to disturb the fuzzer, including possible thread interleavings.



## **Chapter 7**

### **CONCLUSION**

Concurrency bug is one of the most challenging bug types to be reproduced and some concurrency bugs are failed to be reproduced after fuzzer finds them. This thesis proposed REPFUZZER, a kernel fuzzing framework to enable successful reproduce of concurrency bugs once they are found by the fuzzer. It applies deterministic scheduling for kernel threads so that the buggy scheduling of fuzzing phase can be replayed at reproduce phase. Also, with selective thread tracing and token based scheduling, it can solve the challenges with the contexts of kernel and fuzzing. With the evaluation, REPFUZZER shows its ability to reproduce the found bugs in fuzzing phase and to provide stable environment to analyze the bugs.

# Bibliography

- [1] X. Zou and G. Li, “SyzScope: Revealing High-Risk security impacts of Fuzzer-Exposed bugs in linux kernel,” in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, 2022.
- [2] CVE-2020-25656, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-25656>
- [3] CVE-2019-6974, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6974>
- [4] CVE-2019-1999, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1999>
- [5] CVE-2017-2636, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2636>
- [6] CVE-2017-15265, <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15265>
- [7] 20f2e4c2, <https://github.com/torvalds/linux/commit/20f2e4c2>
- [8] 32d3182c, <https://github.com/torvalds/linux/commit/32d3182c>
- [9] 4842e98f, <https://github.com/torvalds/linux/commit/4842e98f>
- [10] 4b848f20, <https://github.com/torvalds/linux/commit/4b848f20>

- [11] 6c605f83, <https://github.com/torvalds/linux/commit/6c605f83>
- [12] 6cd1ed50, <https://github.com/torvalds/linux/commit/6cd1ed50>
- [13] 7311d665, <https://github.com/torvalds/linux/commit/7311d665>
- [14] a6361f0c, <https://github.com/torvalds/linux/commit/a6361f0c>
- [15] da1b9564, <https://github.com/torvalds/linux/commit/da1b9564>
- [16] e20a2e9c, <https://github.com/torvalds/linux/commit/e20a2e9c>
- [17] Syzkaller, <https://github.com/google/syzkaller>
- [18] Kcsan, <https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html>
- [19] Thread Sanitizer,  
<https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual>
- [20] Y. Lee, C. Min, and B. Lee. ExpRace: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380. USENIX Association, Aug. 2021.
- [21] M. Yuan, Y. Lee, C. Zhang, Y. Li, Y. Cai, and B. Zhao. *RAProducer: Efficiently Diagnose and Reproduce Data Race Bugs for Binaries via Trace Analysis*, page 593–606. Association for Computing Machinery, New York, NY, USA, 2021.
- [22] C. Lidbury and A. F. Donaldson. *Sparse Record and Replay with Controlled Scheduling*. New York, NY, USA, 2019.
- [23] Hyunmin Yoon, Shakaiba Majeed, and Minsoo Ryu. 2018. Exploring OS-based full-system deterministic replay. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18)*. Association for Computing Machinery, New York, NY, USA, 1077–1086.

- [24] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, page 193–206, New York, NY, USA, 2009. Association for Computing Machinery.
- [25] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference, ATEC '06*, page 27, USA, 2006. USENIX Association.
- [26] 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)*, pages 415–431, Broomfield, CO, Oct. 2014. USENIX Association.
- [27] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3): 167–178, mar 2010.
- [28] 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)*, pages 1643–1660, 2020.
- [29] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Rizzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.
- [30] 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, SOSP '21*, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

- [31] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII). Association for Computing Machinery, New York, NY, USA, 329–339.

# 초 록

커널 퍼져들이 최근 몇 년간 발전해오면서 퍼져에 의해 보고되는 버그들의 수가 늘어났다. 커널 개발자들이 이런 버그들을 모두 분석할 수 없기 때문에 이런 상황은 디버깅 정보를 제공해줄 수 있는 버그 재현의 중요성을 더욱 중요하게 만든다. 불행히도, 퍼져는 종종 버그 재현에 실패하는데 재현이 힘든 버그 중 하나가 바로 동시성 버그다. 동시성 버그는 여러 스레드 사이의 특정한 조건을 만족할때 발생하는데 커널 스케줄링의 비결정적인 스레드 간섭은 동시성 버그의 재현을 막는다. 그 결과, 일부 동시성 버그들은 버그의 재현에 실패한 뒤 고쳐지지 않은채 버려진다.

이 논문에서는, 퍼져에 의해 발견된 버그를 결정적으로 재현할 수 있도록 도와주는 REPFUZZER 를 소개한다. 이는 선택적 스레드 추적과 결정적 스케줄러를 통해 문제를 해결한다. 선택적 스레드 추적을 통해 REPFUZZER 는 퍼져에 있어서 흥미로운 스레드에만 집중할 수 있도록 한다. 그리고 REPFUZZER 는 결정적 스케줄러를 통해 선택된 스레드를 스케줄 해주며 결정적인 스레드 간섭을 만들어 낸다. 퍼징 단계와 재현 단계 모두에서 REPFUZZER 를 사용함으로써 퍼져는 찾아낸 버그를 재현해낼 수 있게 된다. 결과적으로 REPFUZZER 는 15개의 실제 동시성 버그를 재현하면서 효율성을 보여줬으며, 버그 중 일부는 비결정적인 커널 스케줄링으로 재현해내기 위해 엄청난 시간을 필요로 했다.

**주요어:** 동시성 버그, 커널, 퍼징

**학번:** 2020-27942

# Reproducible Kernel Fuzzer for Concurrency Bugs through Deterministic Scheduling

결정론적 스케줄링을 이용하여 동시성 버그를 재현하는  
커널 퍼져 제작

지도교수 이 병 영  
이 논문을 공학석사 학위논문으로 제출함

2022년 5월



서울대학교 대학원

전기·정보 공학부

이 진 우

이진우의 공학석사 학위 논문을 인준함

2022년 6월

위원장:	김 장 우	
부위원장:	이 병 영	
위원:	심 재 응	