# Thunderkaller: Profiling and Improving the Performance of Syzkaller

Yang Lan[1,2], Di Jin*[3], Zhun Wang[1], Wende Tan[1], Zheyu Ma[1], and Chao Zhang†[1,2]

[1]Institute for Network Sciences and Cyberspace & BNRist, Tsinghua University
[2]Zhongguancun Laboratory, Beijing, China,
{lanyang0908, wz906234737, twd2.me}@gmail.com, mzy20@mails.tsinghua.edu.cn, chaoz@tsinghua.edu.cn
[3]Brown University, Providence, USA, di_jin@brown.edu

***Abstract*—Fuzzing is widely adopted to discover vulnerabilities in software, including the kernel. One of the most popular and state-of-the-art fuzzers for kernels is Syzkaller. However, Syzkaller has a much lower testing throughput compared to other user-space fuzzers, which affects the efficiency of both Syzkaller and other Syzkaller-based fuzzers. In this paper, we profiled the performance of Syzkaller, recognized that the major cost comes from program isolation and kernel instrumentation, and then proposed kernel image duplication and three optimization techniques to mitigate such overheads and present the solution Thunderkaller. Our solution does not change or depend on the fuzzing algorithm in any way, orthogonal to other refinements to Syzkaller. Our evaluation shows that, in 24 hours, Thunderkaller speeds up 2.8× compared to vanilla Syzkaller, achieves 25.8% more basic block coverage, detects 21 more unique bugs, and triggers the common bugs 6.3× faster. In a long time of fuzzing, we have found 6 unique Linux kernel bugs and obtained a CVE.**

***Index Terms*—Fuzzing, OS kernel, Measurement, Performance**

## I. Introduction

Fuzzing is an effective software testing technique for vulnerability detection. It works by generating or mutating test cases and feeding them to programs under test (PUTs) to trigger potential runtime security violations. Compared with other vulnerability detection techniques, fuzzing in general has high throughput and does not need too much manual effort or prior knowledge of the PUTs, which makes it an attractive and dominant vulnerability discovery solution. Depending on the target programs, the test cases generated by fuzzers are in different forms, e.g., documents, network packets, etc.. For OS kernels, a test case in general consists of a sequence of system calls, called a *program* by Syzkaller [8].

Syzkaller is a grey-box kernel fuzzer that uses both code coverage information and input grammar knowledge. It has detected thousands of bugs across multiple operating systems [7]. To record kernel code coverage, Syzkaller uses KCOV [19] to instrument kernel codes at compile-time and applies an evolutionary algorithm to maximize the achieved code coverage. To detect security violations at runtime more precisely, it instruments the kernel with KASAN [18]. Due to its excellent capability and usability, Syzkaller becomes one of the most popular kernel fuzzers. Recently years, a number of newly proposed kernel fuzzers [40], [30], [37], [36], [13] are based on Syzkaller. However, Syzkaller has a much lower testing throughput than other fuzzers targeting user-space programs, e.g., AFL [26], which affects the efficiency of Syzkaller and other Syzkaller-based fuzzers.

To demystify the high overheads of Syzkaller, we first build a measurement pipeline to dissect its cost, regardless of its fuzzing algorithm. Our measurement shows that the execution of the system calls accounts for about 67.4% of the whole fuzzing time, and the remaining 32.6% are spent on inter-program isolation mechanisms such as process management, namespace isolation, and file-system-level isolation. We also show that KCOV and KASAN have a major impact on the throughput. It is still very slow to run programs instrumented with them outside the environment of Syzkaller.

Inspired by the measurement, we propose Kernel Image Duplication (KID) and three optimization techniques to reduce overheads and improve the efficiency of Syzkaller.

**KID:** To eliminate the cost introduced by unnecessary instrumentation, we propose KID. It duplicates the kernel image with different instrumentation during the booting process and dispatches different syscalls to the specific copy (§IV-B).

**Optimization 1:** Skipping Blocking Syscalls. We find that some syscalls would be blocked and stop the entire program until timeout. This would harm the performance of Syzkaller. Therefore, we design a smart policy to automatically skip blocking syscalls depending on the thread state and the blocking history of the syscalls in the same kernel context (§IV-C1).

**Optimization 2:** Debloating Kernel Coverage. We find that the coverage tracking feature `CONFIG_KCOV` can contribute to about 50% of the running time under certain workloads. But coverage tracking is only necessary when running test programs rather than other Syzkaller's activities such as RPC or process management. To eliminate such overheads, we use KID to debloat KCOV (§IV-C2).

**Optimization 3:** Debloating Kernel Sanitizer. Similarly to KCOV, KASAN also harms the performance of software, which has an overhead up to 2.0× in our experiments. As our measurement, removing the minimization and triage phases during the fuzzing period will not undermine the detection ability of KASAN. Hence, we also use KID to eliminate

*Di Jin is the co-first author.
†Chao Zhang is the corresponding author.

KASAN's security checks when the process is not executing the test programs or the test program is under triage or minimization phases (§IV-C3).

Based on these optimizations, we implement a prototype tool Thunderkaller and evaluate its performance in throughput, code coverage, and bug finding. When compared to Syzkaller, during 24 hours, Thunderkaller speeds up about 2.8× (§VI-B), achieves about 25.8% more basic block coverage (§VI-C), discovers 21 more unique bugs, and detects common bugs 6.3× faster (§VI-D). Besides, we found 6 unique kernel bugs and got a CVE.

In this paper, we make the following contributions:

- **Profiling Syzkaller.** We conduct a measurement and dissect the cost of Syzkaller, which gives a systematic understanding of the major overheads.
- **Optimizing Syzkaller.** According to our measurement, we propose KID to reduce unnecessary instrumentation, come up with three optimizations, and implement a tool dubbed Thunderkaller to improve the performance of Syzkaller.
- **Evaluation.** We evaluate Thunderkaller from four aspects, showing that Thunderkaller can significantly improve the performance of the vanilla Syzkaller.

## II. Background

### A. Syzkaller

Syzkaller [8] is the most popular kernel fuzzer at present. It detects kernel bugs via executing test programs consisting of a sequence of system calls. Syzkaller provides manually crafted templates for system call sequence generation and also utilizes mutation-based fuzzing to yield new sequences from existing sequences. Syzkaller has several major components as follows:

- **Syz-manager.** This component is the control center of Syzkaller, responsible for initializing the basic works, storing fuzzing information at a local database, communicating with other components, etc. For example, it would spawn a couple of virtual machines and copy other components into VMs to test the target OS kernel. It communicates with VMs via RPC to retrieve the coverage and crashes information during fuzzing.
- **Syz-fuzzer.** It is the brain of the fuzzer, responsible for generating and mutating test programs. It stores the test programs in a working queue, selects them one by one via a certain strategy from the queue, then sends them to the *syz-executor* component for testing. Specifically, each test program could be used in different working modes, including *triage*, *minimization*, *hints*, *smash*, etc. The life cycle of *syz-fuzzer* is the same as VMs.
- **Syz-executor.** This component accepts test programs from *syz-fuzzer*, then interprets and executes them. It is launched by *syz-fuzzer* and has a transient life cycle. It can use the shared memory to communicate with *syz-fuzzer*.

Given a test program selected from the working queue, (1) if it is in *hints* or *smash* working modes, then the fuzzer will mutate syscalls' arguments using the comparison operands collected by Syzkaller or perform a series of random mutations to yield a set of test programs and send them for the executor to execute. During the testing, if a test program triggers new code, it will be added to the working queue again and set with the *triage* mode. (2) if it is in *triage* mode, the test program will get executed several times to verify whether it indeed contributes to code coverage, eliminating unexpected noises in coverage increment, e.g., special OS state, concurrency, and interaction behavior in multi-thread mode and non-determinism operations in fuzzing [40]. If a test program indeed explores new code, then it will be set with the *minimization* mode, otherwise aborting triage. (3) if it is in *minimization* mode, the fuzzer will try to minimize the test program by removing some syscalls or shortening their arguments while preserving the coverage. After minimization, the test program will be added to the local corpus for further mutation.

### B. KCOV: Kernel Code Coverage

KCOV [19] is a code coverage tester for the system kernel. It can provide access to the code coverage information during fuzzing kernels. KCOV instruments the kernel codes with the sanitizer coverage of compilers and places the coverage recording functions (which we will call coverage hooks) in front of each basic block (shown as line 3 in Figure 6(a)). If enabling KCOV on a kernel task basis, the kernel process can obtain the precise code coverage of each system call. Moreover, the kernel adds the coverage collection mode in the structure of kernel task `struct task_struct` to indicate whether KCOV is enabled. Users can access the code coverage data through the file `/sys/kernel/fs/debug/kcov`.

Although the original KCOV implementation has already skipped coverage recording outside the test program execution, in every basic block, the instrumented function calls used to invoke the coverage hooks are not eliminated, and still introduce overheads.

### C. KASAN: Kernel Address Sanitizer

Kernel address sanitizer (KASAN) is a dynamic memory error detector for Linux kernels, e.g., use-after-free and out-of-bounds bugs. KASAN inserts instrumentation into kernel codes and replaces certain memory allocators in the Linux kernel at compile time to detect memory errors at runtime.

KASAN utilizes the shadow memory to record whether the memory address is safe to access. In detail, KASAN inserts instrumentation in front of each memory access instruction. It utilizes a run-time library to manage the shadow memory and implements the specific memory allocators to replace the default ones, such as `malloc` and `free`. Besides, KASAN allocates poisoned redzones around each global and stack object to detect memory bugs.

## III. Motivating Study

To study the performance of Syzkaller, we build a measurement pipeline to discover its performance bottleneck, including the measurement of the performance of Syzkaller with different configurations and analyzing the root cause of the long-running test programs.

TABLE I
LINUX KERNELS WITH DIFFERENT CONFIGURATIONS.

| Version | Kernel Configuration |
|---|---|
| Kernel$^{\times}_{\times}$ | the `defconfig` |
| Kernel$^{\text{kcov}}_{\times}$ | the `defconfig` with `CONFIG_KCOV` enabled |
| Kernel$^{\times}_{\text{kasan}}$ | the `defconfig` with `CONFIG_KASAN` enabled |
| Kernel$^{\text{kcov}}_{\text{kasan}}$ | the `defconfig` both with `CONFIG_KCOV` and `CONFIG_KASAN` enabled |
| Kernel-Syz$^{\text{kcov}}_{\text{kasan}}$ | default conf of Syzkaller with `CONFIG_KCOV` and `CONFIG_KASAN` enabled |
| Kernel-Syz$^{\times}_{\times}$ | default conf of Syzkaller with `CONFIG_KCOV` and `CONFIG_KASAN` disabled |

## A. Measurement Pipeline

We run the modified Syzkaller in one fuzzer VM configured with two cores, one process, and 2GB memory, testing on the Linux kernel 5.10 with the default kernel configuration [9]. Our measurement consists of the following steps:

► **Step 1: Mitigating performance fluctuation.** We aim to improve the overall performance of Syzkaller, regardless of its fuzzing algorithm. In addition, to measure the performance of a specific function in Syzkaller, we modify it to get several variants of Syzkaller. To avoid the effects of random mutation and generation strategies of fuzzing, we build the variant **Syzkaller-gen**, which disables the default fuzzing algorithm of Syzkaller. The second variant we built is **Syzkaller-corpus**, which executes all the test programs in a corpus once without mutation and generation phases. To catch test programs that time out and record the execution time of each system call, we build the variant **Syzkaller-time**.

As we know, the performance of the kernel will fluctuate if the type of syscalls tested and the kernel configurations have varied. Therefore, we test Syzkaller-gen with various kernel configurations and different groups of syscalls to mitigate such fluctuations. We divide syscalls into five sets according to their execution time: less than 5,000, 10,000, 20,000, 50,000, and 100,000 microseconds, denoted as *Syscall-5ms* to *Syscall-100ms*, respectively. *Syscall-all* means enabling all syscalls. Moreover, we have compiled six versions of the Linux kernel with different configurations (shown in Table I).

► **Step 2: Breaking Down the Time Cost of Syzkaller.** As §II-A describes, the execution process of Syzkaller includes several working modes. To begin with, we break down the time cost of working modes in a fuzzing campaign to figure out the overall time distribution of Syzkaller and which working modes are the most time-consuming. Then, we break down the time cost of running an individual test program in *syz-executor* during a fuzzing campaign. Consequently, we can find which execution phases of a test program take the most time.

In detail, we instrument Syzkaller-gen with several timing points to analyze the cost of each phase as follows and set up a control pipe between *syz-executor* and *syz-fuzzer* to share the timing points.

- **Setup:** creating a temporary directory for the test programs and receiving data from *syz-fuzzer* in the parent process.
- **Fork Child:** time between forking in the parent process and returning from the fork in the child process.
- **Child Setup:** doing architecture-specific setup (network, file system, etc.) in the child process.

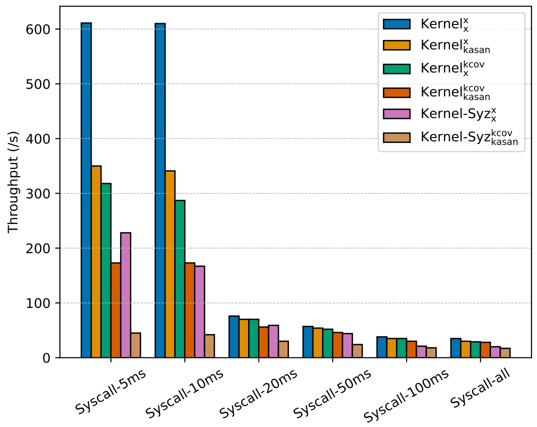


Fig. 1. Throughput on different workloads and kernel configurations.

- **Exec Loop:** fetching and interpreting the test program candidates in the child process.
- **Wait:** time before the child process exits to the parent process catches the child.
- **Reset:** removing the private work directory and checking memory leak in the parent process.
- **Extra Time:** time to execute the timing function calls that we instrumented.

► **Step 3: Measuring the Cost of System Calls.** In this step, we further break down the time cost of invoking system calls and take the system call as the granularity to gain a time distribution of Syzkaller. We measure the time cost of each phase of invoking system calls, including the switching cost between guest and host, the switching into kernel mode, and the cost of instrumentation.

In addition, we collect the execution time of each system call by Syzkaller-time, capture and manually analyze the test programs with long execution times.

► **Step 4: Analyzing Results.** Last, we analyze the results based on previous measurement data and pinpoint the root causes of performance overheads of Syzkaller, shedding light on our optimization techniques.

## B. Result Analysis

*1) Mitigating performance fluctuation:* Figure 1 shows the throughput of Syzkaller-gen in different settings. When testing the Linux kernel with the `defconfig` [20] and disabling those system calls that take longer than 5,000 microseconds to execute, the throughput can achieve about 611 test programs per second. This result is higher than what we realize. However, it is expected since most execution time of system calls is longer than 5,000 microseconds (as our measurement in §III-B3), resulting in only 21 system calls that could be executed in this workload. A key observation is that either `CONFIG_KCOV` or `CONFIG_KASAN` can slow down Syzkaller's throughput by about 50%, and both could decrease throughput by about 75% with Syscall-5ms and Syscall-10ms. Moreover, when turning on more kernel configurations, the longer execution time of system calls will be and the lower throughput of Syzkaller. Therefore, reducing the instrumenta-

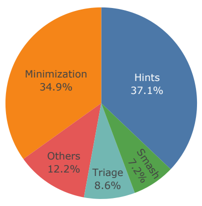


Fig. 2. Time cost of different working modes.

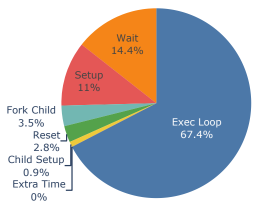


Fig. 3. Time breakdown of execution phases of a test program.

tion overhead and the time cost of system calls can enhance Syzkaller's performance.

*2) Breaking Down the Time Cost of Syzkaller:* Figure 2 shows the time cost of different working modes relative to the total execution time. We conduct this experiment on the default Syzkaller for 24 hours. It shows that the vanilla Syzkaller spends over 40% of time on the minimization and triage working modes. Besides, we also find an interesting observation that after about 12 hours, the execution time or the number of the programs in hints mode is more than in minimization mode, which is consistent with the previous study [40]. This phenomenon is caused by the trade-off between exploration and exploitation of Syzkaller's default fuzzing strategy. It also demonstrates that the optimal strategy of the vanilla Syzkaller should be dynamic and can be improved.

Another critical insight is that removing KASAN's checks under the minimization and triage working modes will not compromise KASAN's ability to detect vulnerabilities. To demonstrate that we modify the default Syzkaller, excluding these two working modes, and analyze KASAN's ability. We found that the modified Syzkaller can detect all the vulnerabilities as the vanilla Syzkaller. However, only the time-to-exposure (TTE) of vulnerabilities becomes longer. The evaluation (§VI-D) also demonstrates that it would not undermine the detection ability of KASAN. Thus, diminishing the cost of executing the minimization and triage test programs by KASAN can improve the performance of Syzkaller.

Figure 3 illustrates the breakdown of time costs during the execution phases of a test program. We collect the time cost of each phase compared to the total execution time. Our results show that fetching and interpreting the test programs is the primary cost phase, making up 67.4% on average. Hence, reducing the time cost of performing system calls is the prominent problem.

*3) Measuring the Cost of System Calls:* Our experiment shows that the average execution time of 70% system calls exceeds 10,000 microseconds, and an individual system call is about 14,451 microseconds. Thus, most system calls are slow, such as file systems' average execution time is about 34,742 microseconds and accounts for 16.7% of the total time.

Further, we measure the cost of the user-kernel mode switching and the performance differences between host and guest. Since `getpid` [17] has few operations done in the Linux kernel, it is an ideal API to evaluate the time cost. We thus test `getpid` on the Linux kernel with different configurations. Results showed that the time cost of mode switching in the modern processor is about 57.9 nanoseconds within a round trip, which is consistent with previous study [22]. Moreover, the average execution time of `getpid` is 69.4 nanoseconds on the guest. Therefore, the cost of user-kernel mode switching and the host/guest difference are not the major causes of slowdown.

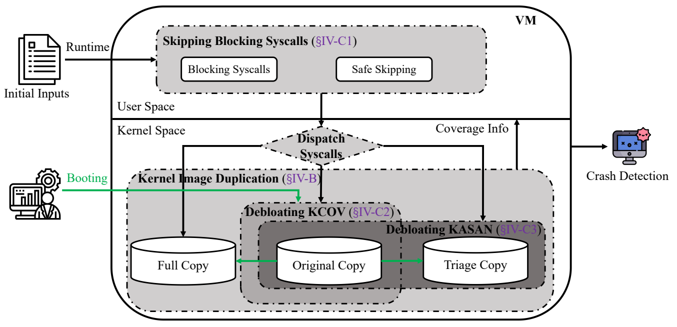


Fig. 4. The workflow of Thunderkaller.

We manually analyze the test programs that take a long time to execute. The primary reason is the complex system calls that require a significant amount of time to execute (e.g., mounting file system, destructing net namespace, etc.), and the execution phases of these system calls are different. Another reason is the `blocking` system calls such as FUSE [16], [38] — a framework that supports user-implemented file systems, etc.. Therefore, we should design specific optimization techniques regarding these complex system calls, such as fuzzing file systems and communication protocols. Besides, we can reduce the cost caused by the blocking system calls to improve Syzkaller's performance.

*4) Root cause analysis:* We find the slow throughput of Syzkaller is due to three aspects, the complex system calls, the blocking system calls, and the cost of instrumentation. The complex system calls, for example, mounting file systems take up about 16.7% of the total executing time, and the average execution of a system call is about 14,451 microseconds. The blocking system calls, such as FUSE, etc., can undermine the performance of Syzkaller. KCOV and KASAN are the two most expensive instrumentation in fuzzing, both decreasing throughput by about 75% in our experiments.

## IV. Design

### A. Overview

After manual analysis, we find that the test program timing out is partly caused by blocking system calls (§III-B3). To reduce the overhead caused by them, we propose the first optimization, Skipping Blocking Syscalls (SBS) (§IV-C1).

As shown in Figure 1, for system calls whose execution time is under 5,000 and 10,000 microseconds, enabling KCOV will lower the throughput by about 50%. A key observation is that Syzkaller only needs to collect coverage caused by test programs, not other Syzkaller's activities such as RPC or process management. To eliminate the cost caused by calling the coverage hooks while executing syscalls that are not under test, we design a technique named KID (§IV-B) to debloat

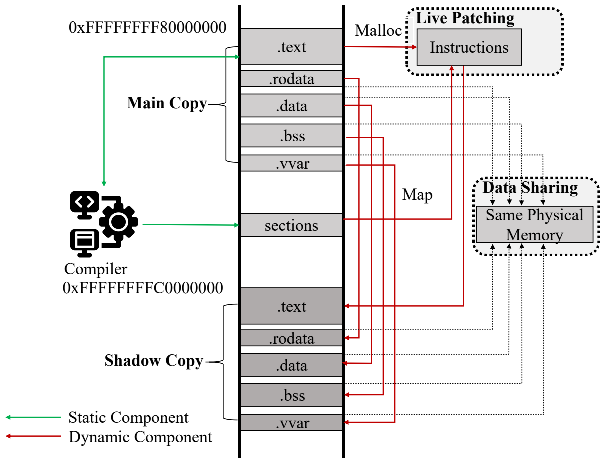


Fig. 5. The details of KID.

kernel coverage tracking (DKCOV) (§IV-C2). We also found that the eager allocation of KCOV memory contributes to extra overhead brought by spawning child processes. Hence, we change it to on-demand allocation.

Figure 1 also shows that KASAN reduces the throughput by about 50%, which is consistent with observations from previous works [11], [48], [47]. Figure 2 shows that the time cost of minimization and triage working modes take up over 40% in a fuzzing campaign. Furthermore, our experiments demonstrate that removing the minimization and triage phases will not harm the crash-detection ability of KASAN. To reduce the overhead caused by KASAN, we propose to leverage KID again to remove KASAN's checks when the system calls are not under test or in these two phases (DKASAN) (§IV-C3).

Figure 4 depicts the workflow of Thunderkaller, which adopts three optimizations. Thunderkaller duplicates the kernel code to get three copies with different instrumentation configurations. The original copy has both KASAN's checks and KCOV instrumentation removed, the full copy enables all instrumentation, and the triage copy only removes KASAN's checks. The test programs other than those at the minimization and triage phases will run on the full copy; Thunderkaller will dispatch other test programs to the triage copy; the rest system calls from other activities will go to the original copy.

## B. Kernel Image Duplication

Previous works have proposed several methods to reduce unnecessary instrumentation when fuzzing userspace programs. However, these techniques do not work in kernel fuzzing and how to remove unnecessary kernel instrumentation is still an open problem. The static or dynamic binary rewriting technique used in Untracer [28] or Zeror [50] is not scalable and time-consuming on kernels. Moreover, the recompilation method proposed by ODIN [42] will introduce enormous overhead for kernels. Hence, we propose a new solution, namely KID, to eliminate the cost caused by unnecessary kernel instrumentation. KID will (1) duplicate the kernel code to produce two identical copies except for the instrumented instructions of interest; (2) dispatch various syscalls to accordingly copies; (3) ensure all syscalls can be executed correctly in each copy and not introduce extra overhead at runtime.

Figure 5 depicts the details of KID. Specifically, KID includes two components: a **static component** in the compiler and **dynamic component** during the kernel's booting process. The static component modifies the code and emits information regarding where to patch the code to achieve different behaviors in the new copies. The dynamic component runs toward the end of the booting process. It duplicates and maps the shadow copy to the correct region and patches the shadow code using the information emitted by the static component.

At the dynamic component, we need to create two similar copies besides certain instrumented instructions and ensure running all syscalls correctly in new copies. KID faces two main challenges: handling *live patching* and *data sharing*. To achieve the live patching, we add a kernel `initcall` at the end of the booting process to initialize KID. During booting, the kernel reads from the specified sections and patches instructions accordingly before changing the code in the shadow copy to be non-writable.

The execution initiated in one copy should stay in the same copy. Otherwise, a system call under test will have control flow diverging to the main copy and miss coverage, even incurring crashes. Thus, we need to share the same data between each copy to solve this problem. In detail, on X86-64, the Linux kernel accesses global data and per-CPU data with `%rip` relative addressing. When the kernel accesses global data in the shadow copy, it will access invalid memory because the address of instructions in the shadow copy is changed. To address this issue, we double-map global and per-CPU data to their correct positions relative to the main and shadow code in both the main and shadow copies, correspondingly. Thus, the execution of either the main or shadow code will reference their respective data addresses. But the memory states will be shared because the data addresses are mapped to the same physical memory.

Because there is only one physical copy of data, all function pointers inside any data structure will be pointing to the main copy. This would make any indirect control-flow transfer always go to the main copy. To avoid this, during compilation, we unfold every indirect control-flow transfer instruction that uses memory operands to three separate instructions: a `mov` to load the operand into a register, an `add` to potentially fix up the control-flow transfer target, and a respective control-flow transfer instruction that uses the register as its operand (shown as lines 8–10 in Figure 6(b)). All the addresses of the fix-up immediates are recorded in a dedicated ELF section. During the booting process, KID patches the immediate used by the `add` instructions in the shadow copy such that indirect control-flow transfers in the shadow copy can stay in the same copy (shown as lines 8–10 in Figure 6(c)).

## C. Three Optimization Techniques

*1) Skipping Blocking Syscalls:* Listing 1 shows an example of blocking syscalls related to FUSE. The test program first sets up the file system (lines 1–5) and then opens a directory

```
1 mkdirat(...,&(0x7f0000002040)='./file0\x00', 0x0)
2 r0 = openat$fuse(...,&(0x7f0000002080), 0x2, 0x0)
3 mount$fuse(0x0, ...)
4 read$FUSE(r0, &(0x7f00000021c0)={..., <r1=>0x0}, 0x2020)
5 write$FUSE_INIT(r0, &(0x7f0000004200)={..., r1}, 0x50)
6 r2 = openat$dir(..., &(0x7f0000004280)='./file0\x00', 0x0
      , 0x0) // Timeout
7 syz_fuse_handle_req(r0, ...)
8 getdents64(r2, &(0x7f0000006380)=""/1024, ...)// Timeout
9 syz_fuse_handle_req(r0, ......)
```

Listing 1. An example of blocking syscall.

(line 6). Such a request will block until the user program that implements the file system finishes processing it. To prevent blocking syscalls from stopping the execution of the entire program, Syzkaller puts the execution of each syscall into a separate thread, and starts executing the next syscall (e.g., line 7) after a timeout. Similarly, the test program reading data from the dictionary into the buffer will block too (line 8). A long enough timeout is needed because there are slow syscalls that require more time to execute, and executing the next one prematurely can lead to the following system calls not having the intended context. However, when a syscall is blocked, it will harm the performance of Syzkaller.

To address this, we design an algorithm that automatically skips blocking syscalls by monitoring the state of the current working thread and whether the syscall has been blocked before. Another observation is that in the same kernel context, if a syscall has been blocked before, it will also be blocked later. This algorithm is based on two empirical assumptions:

► **Blocking Syscalls:** *If a thread is in the idle or sleeping state for a long time, then the syscall it is executing is blocked.*

► **Safe Skipping:** *If a syscall has been blocked before, then it is safe to skip it and start the next syscall immediately after the calling thread enters the idle or sleeping state.*

The evaluation also demonstrates that this optimization will not undermine the code coverage (§VI-E). Based on these two assumptions, Syzkaller can skip blocking syscalls correctly.

*2) Debloating Kernel Coverage:* We use KID to reduce unnecessary coverage hooks. Two things should be different between the main copy and the shadow copy: KCOV instrumentation and the indirect branching fix-ups.

A detailed example is shown in Figure 6 to elaborate on how KID eliminates unnecessary coverage hooks with the static and dynamic components. In the static component, we choose to nop out all the coverage hooks in the main copy (line 3 in Figure 6(b)). In the dynamic component, adding them back in the shadow copy (line 3 in Figure 6(c)), and patching the fix-up immediates into an offset (line 9 in Figure 6(c)). By dispatching between the two copies, Thunderkaller can only let the syscalls under test run on the instrumented shadow copy.

► **On-demand Allocation.** To use KCOV, Syzkaller allocates a buffer that is shared with the kernel to retrieve the coverage information. This buffer is also shared between parent and child processes, such that the parent can have access to coverage created by child processes, even if child processes exit abnormally during testing. By default, Syzkaller uses 32MiB of buffers, accommodating potentially complex syscalls. The problem is that all the page tables are eagerly duplicated when the parent spawns a child, yet most test programs will only use a small fraction of the buffer. In other words, KCOV's default coverage buffer implementation does not scale down. We change the coverage buffer in the child processes to be mapped on-demand allocation, reducing the cost of spawning child processes, which happens for every new test program.

*3) Debloating Kernel Address Sanitizer:* As aforementioned, Syzkaller will re-execute and shorten test programs during the triage and minimization phase. Any new coverage discovered during these two phases is scheduled to be recursively triaged and minimized again in the future. The intuition is that the test programs executed in these two phases are either identical or very similar to the original test programs that triggered the new coverage. In the later smash and hint phases, Syzkaller will perform random mutations including those executed in the triage and minimization phases. Even when KASAN could have caught new memory errors during the minimization and triage phases, the random mutation in the later phases could likely trigger the same memory errors. The evaluation shows that removing KASAN's checks in these two phases will not introduce false negatives (§VI-D). Thus, we use KID again to diminish the overhead incurred by the redundant KASAN's checks in Syzkaller. We skip KASAN's checks in the main copy and recover them in the shadow copy.

Figure 7 shows the default KASAN's checks and the difference after KID modifies the kernel code. During compilation, we choose to skip over KASAN's checks of load/store instructions, and stack variables in the main copy (lines 1, 6, 11, 15 in Figure 7(b)). During booting, we nop them out in the shadow copy (lines 1, 6, 11, 15 in Figure 7(c)). By dispatching syscalls, KID can make the test programs go to the main copy under the minimization and triage phases. Others run on the shadow copy where KASAN's checks are recovered.

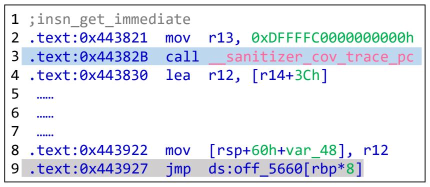
TABLE II
IMPLEMENTATION DETAILS OF THUNDERKALLER.

| Component | Base Tool | LoC |
|---|---|---|
| Measurement Pipeline | Syzkaller, Linux Kernel | 500 (Go) |
| Kernel Image Duplication | LLVM, Linux Kernel | 500 (C++), 700 (C) |
| Fuzzing Loop | Syzkaller, Linux Kernel | 300 (Go), 1,500 (C) |
| Instrumentation | LLVM | 1,000 (C++) |
| Glue scripts | - | 500 (Python) |
| **Total** | - | **5,000** |

## V. Implementation

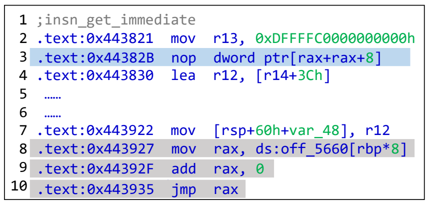
We have implemented a Thunderkaller prototype with the aforementioned optimizations. The optimization can be enabled through a customized configuration. We build Thunderkaller and measurement pipeline on top of Syzkaller (commit `bc5f1d8`) and Clang from LLVM-10.0. Table II summarizes components of them and corresponding lines of code.

► **Kernel Image Duplication.** We implement the static part of KID as an LLVM X86-64 backend pass [24]. KID will emit instruction addresses to different ELF sections depending on



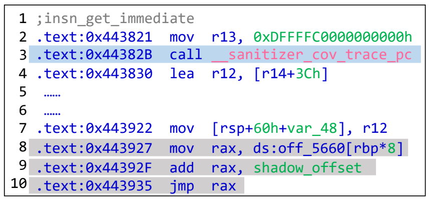
```
;insn_get_immediate
.text:0x443821  mov  r13, 0xDFFFFC0000000000h
.text:0x44382B  call  __sanitizer_cov_trace_pc
.text:0x443830  lea  r12, [r14+3Ch]
 ……
 ……
 ……
.text:0x443922  mov  [rsp+60h+var_48], r12
.text:0x443927  jmp  ds:off_5660[rbp*8]
```

(a) Original KCOV instrumentation



```
;insn_get_immediate
.text:0x443821  mov  r13, 0xDFFFFC0000000000h
.text:0x44382B  nop  dword ptr[rax+rax+8]
.text:0x443830  lea  r12, [r14+3Ch]
 ……
 ……
.text:0x443922  mov  [rsp+60h+var_48], r12
.text:0x443927  mov  rax, ds:off_5660[rbp*8]
.text:0x44392F  add  rax, 0
.text:0x443935  jmp  rax
```

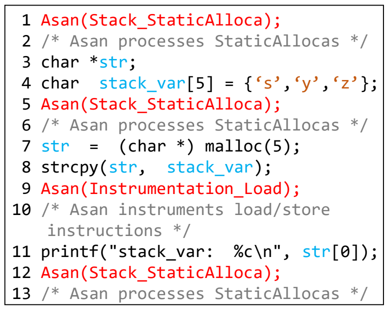
(b) KID modified instrumentation (Main Copy)



```
;insn_get_immediate
.text:0x443821  mov  r13, 0xDFFFFC0000000000h
.text:0x44382B  call  __sanitizer_cov_trace_pc
.text:0x443830  lea  r12, [r14+3Ch]
 ……
 ……
.text:0x443922  mov  [rsp+60h+var_48], r12
.text:0x443927  mov  rax, ds:off_5660[rbp*8]
.text:0x44392F  add  rax, shadow_offset
.text:0x443935  jmp  rax
```

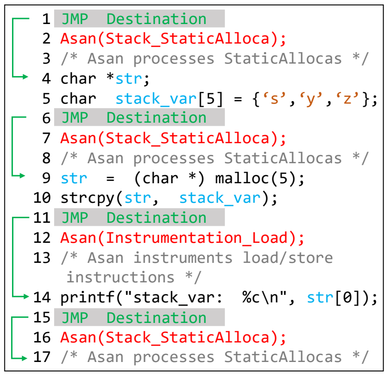
(c) KID modified instrumentation (Shadow Copy)

Fig. 6. Debloating KCOV example.



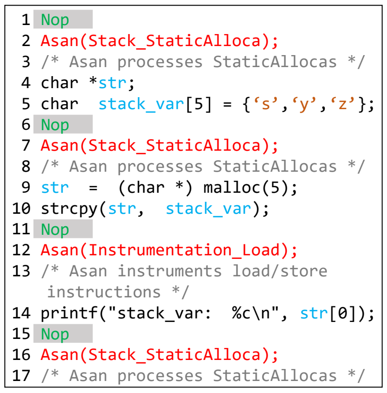
```
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
char *str;
char  stack_var[5] = {'s','y','z'};
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
str  =  (char *) malloc(5);
strcpy(str,  stack_var);
Asan(Instrumentation_Load);
/* Asan instruments load/store
 instructions */
printf("stack_var:  %c\n", str[0]);
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
```

(a) Original KASAN instrumentation



```
JMP  Destination
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
char *str;
char  stack_var[5] = {'s','y','z'};
JMP  Destination
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
str  =  (char *) malloc(5);
strcpy(str,  stack_var);
JMP  Destination
Asan(Instrumentation_Load);
/* Asan instruments load/store
 instructions */
printf("stack_var:  %c\n", str[0]);
JMP  Destination
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
```

(b) KID modified instrumentation (Main Copy)



```
Nop
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
char *str;
char  stack_var[5] = {'s','y','z'};
Nop
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
str  =  (char *) malloc(5);
strcpy(str,  stack_var);
Nop
Asan(Instrumentation_Load);
/* Asan instruments load/store
 instructions */
printf("stack_var:  %c\n", str[0]);
Nop
Asan(Stack_StaticAlloca);
/* Asan processes StaticAllocas */
```

(c) KID modified instrumentation (Shadow Copy)

Fig. 7. Debloating KASAN example.

how to fix them. Moreover, we change the linker script of the Linux kernel to map these sections to accordingly region, exposing their starting and ending addresses as symbols to be used during initialization.

The dynamic part of KID is built inside an `initcall` of the Linux kernel. For handling the live patch, KID duplicates the `.text` segment by allocating a virtual memory block and copying kernel codes to the allocated memory. It patches instructions of interest in the memory block. Then, KID maps the code in the memory block to the shadow copy.

Furthermore, we compile the Linux kernel with the compiler option `-mcmodel=kernel`, which means that all code lives inside the upper 2GiB region. We also disable KASLR in Thunderkaller just like Syzkaller, so the address of the main copy starts at `0xFFFFFFFF80000000`. Then we reduce and move the region for kernel modules and place the shadow copy at `0xFFFFFFFFC0000000`. KID does not duplicate modules because Thunderkaller chooses to link all modules into the kernel image (which is also consistent with Syzkaller). Another modification needed is in exception handling [21], specifically the `__ex_table` section. When a kernel instruction faults, the faulting instruction address is used to look up where to continue from. Hence, we extend the `__ex_table` section so that KID can correctly handle faults in the shadow copy.

► **Skipping Blocking Syscalls.** We build a bitmap recording whether a syscall has been blocked in *syz-fuzzer*. Thunderkaller will not reset it unless the VM reboots from kernel crashes. When transferring test programs to *syz-executor*, we use an additional flag to indicate whether the syscall has been blocked before, so *syz-executor* can decide whether to skip the syscall when detecting an idle or sleeping state. Thunderkaller detects the state by monitoring child threads via `procfs` [23]. After executing programs, *syz-fuzzer* will update the bitmap through the value returned from *syz-executor*.

► **Debloating Kernel Coverage and Address Sanitizer.** For debloating KCOV, we change the static component of KID to nop out all kernel coverage hooks and record addresses of these nopped-out instructions in ELF sections. KID reads from these sections to recover kernel coverage hooks during the booting process. Syzkaller activates the `KCOV_MODE` to allow kernel processes to collect coverage. When invoking a system call under this mode, Thunderkaller will route the system call to execute on the shadow copy.

The static part of debloating KASAN is implemented in the address sanitizer pass of LLVM and adds `jmp` instructions to go over KASAN's checks. KID will also record addresses of these `jmp` instructions in a dedicated ELF section for the dynamic component to replace with `nop` instructions. We add a new field in `struct task_struct` marking whether the kernel process needs to do KASAN's checks. If not under this mode, Thunderkaller will dispatch syscalls to the main copy to skip KASAN's checks.

► **Switching between Different Copies.** We implement three optimizations in a three-copy way. The original copy, the full copy, and the triage copy start at `0xFFFFFFFF80000000`, `0xFFFFFFFFC0000000`, and `0xFFFFFFFFE0000000`, respectively. First, Thunderkaller acquires the mode of kernel processes to determine which copies the system call should go. Then, Thunderkaller dispatches them to the corresponding copies by adding offsets to the system call function address. In detail, we augment `do_syscall_64()`, which serves as the entry of X86-64 system calls. When systems invoke a syscall, it accesses the `sys_call_table` to obtain the address of the corresponding syscall function. Then it stores the return value into the `regs->ax` field. Adding an offset to this field, Thunderkaller can route various syscalls to accordingly copies.

## VI. Evaluation

To demonstrate the effectiveness of Thunderkaller, we comprehensively evaluate it from various aspects. We first introduce the experimental setup. Then, we present the performance of Thunderkaller in terms of throughput, basic block coverage, and crash detection, respectively. For throughput and code coverage, Thunderkaller is orthogonal to other refinements of the fuzzing algorithm to Syzkaller [30], [40], [37], [13]. Healer is written by Rust and applying Thunderkaller to it requires a tremendous engineering effort. Hence, we apply Thunderkaller to Moonshine. We next conduct the ablation experiment. Moreover, we also use Thunderkaller to test other Linux kernel versions. To evaluate the detection ability of Thunderkaller, we compare time-to-exposure (TTE) of bugs with the default Syzkaller during 24 hours. Besides, we perform a 7-day experiment to detect previously unknown bugs and analyze whether Thunderkaller would introduce false negatives. Last, we evaluate whether skipping blocking system calls would undermine the code coverage.

In summary, we aim to address the following questions:

- RQ1: How does Thunderkaller perform in terms of throughput? (§VI-B)
- RQ2: How does Thunderkaller act regarding code coverage? (§VI-C)
- RQ3: How does Thunderkaller perform in the matter of crash detection? (§VI-D)
- RQ4: Does optimization skipping blocking syscalls undermine the code coverage? (§VI-E)

### A. Experimental setup.

We evaluate Thunderkaller on a machine with an Intel 14-core Core i9-10940X 3.30GHz CPU and 128GB of memory. To evaluate throughput and code coverage, we conduct each experiment with one fuzzer VM configuring two cores, one process, and 2GB memory, fuzzing the Linux kernels 5.10 and 5.15 for 24 hours with all system calls enabled. We apply our three optimizations to Moonshine and compare the performance of Thunderkaller and Syzkaller with different initial corpus. First, we utilize the system calls traces provided by the Moonshine [30], which analyzes the dependency of system calls from Open Posix Tests [2], kselftest [1], and Linux Testing Project [25], to create the Moonshine corpus consisting of 383 test programs. For comparison, we also created a 24-hour corpus by running the vanilla Syzkaller for 24 hours, containing 17348 test programs.

To compare the TTE of bugs we test on the Linux kernel 5.10 with four VMs configuring three cores, six processes, and 2GB memory in each VM for 24 hours. To detect previously unknown bugs, we perform experiments for 7 days. Besides, we build a benchmark, consisting of 14 bugs found by Syzbot [7] (shown in Table IV), to analyze false negatives by picking crashes spread across multiple Linux kernel versions over the past years. We only choose bugs with `poc` since we can reproduce them manually. Because Clang is not fully supported for some Linux kernel versions, we recreate buggy kernels by undoing the proposed patch when necessary. We utilize `syz-execprog`, a utility tool provided by Syzkaller, to execute and reproduce test programs manually. To reduce bias, we repeat each experiment 10 times.

### B. Throughput Evaluation (RQ1)

We apply Thunderkaller to Moonshine, comparing the throughput of Moonshine with and without our optimizations. Besides, different seeds can drastically change the workload of fuzzers, and we also evaluate Thunderkaller on three different initial corpora. We run Thunderkaller and Syzkaller on the empty initial corpus, Moonshine corpus, and the 24-hour corpus, respectively. Results are shown in Figure 8(a).

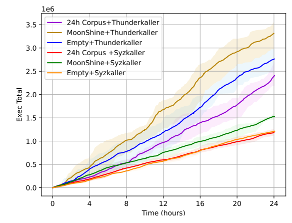
In contrast, regardless of the type of the initial corpus, Thunderkaller outperforms Syzkaller and achieves 2.3×, 2.6×, 2.2× speedup with Moonshine, empty corpus, and 24-hour corpus, respectively. Besides, the different corpus has different impacts on Thunderkaller and Syzkaller. Moonshine speeds up about 21.7% on Thunderkaller, while 27.4% on the default Syzkaller. We also find the 24-hour corpus slows down both Thunderkaller and Syzkaller because fuzzing with the initial corpus, Syzkaller would poll seeds from the corpus rather than generate a new one. Moreover, the seed selection strategy of Syzkaller would prioritize seeds with higher initial coverage and other imported seeds won't get many mutation chances. These seeds with higher initial coverage contain complex syscalls, such as building file systems, etc.. Consequently, it results in most test programs being unavoidably slow and reducing the efficiency of our optimizations.

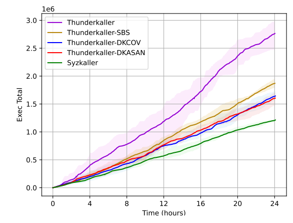
Figure 8(b) shows the throughput of different optimizations with kernel 5.10. It shows that debloating KASAN, debloating KCOV, and skipping blocking syscalls can give about 30%, 40%, and 50% speedup, respectively. In total, Thunderkaller achieves 2.6× speedup increase.

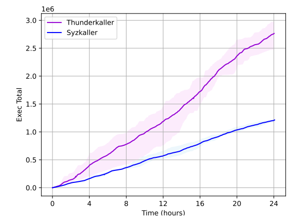
Results of testing with different kernels show in Figure 8(c), 8(d). Compared to the vanilla Syzkaller, fuzzing with kernels 5.10 and 5.15, Thunderkaller speeds up about 2.6×, 3.0× on average, respectively.

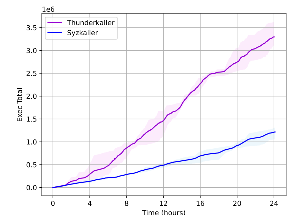
### C. Basic Block Coverage Evaluation (RQ2)

Figure 9(a) shows the coverage with different initial corpora of Thunderkaller and the vanilla Syzkaller. It shows that Thunderkaller could better use empty corpus and the two kinds of the initial corpus. Compared to the vanilla Syzkaller with the same corpus, Thunderkaller yields 3.0% more coverage with the 24-hour corpus, 10.0% with Moonshine, and 15.7% with an empty corpus. Compared with the empty corpus, 24-hour corpus and Moonshine yield 13.0% and 4.4% more coverage in Thunderkaller, and about 26.6% and 10.6% more coverage in Syzkaller, respectively. We also observe that Thunderkaller



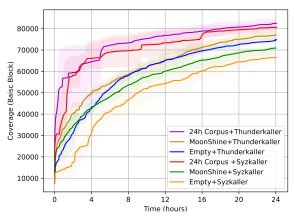


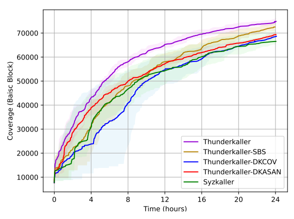


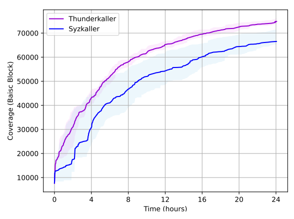


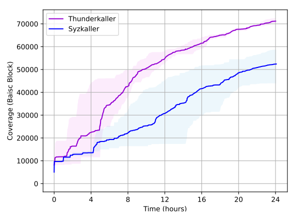
(a) Throughput of initial corpus (b) Throughput of optimizations (c) Throughput on Linux kernel 5.10 (d) Throughput on Linux kernel 5.15

Fig. 8. Statistics of throughput.









(a) Coverage of initial corpus (b) Coverage of optimizations (c) Coverage on Linux kernel 5.10 (d) Coverage on Linux kernel 5.15

Fig. 9. Statistics of coverage.

doesn't significantly improve coverage than Syzkaller with the 24-hour corpus. It's also caused by the default fuzzing strategy of Syzkaller, discussed in VI-B. Besides, coverage spikes first but flattens out later regardless of Thunderkaller or Syzkaller with 24-hour corpus. Since Syzkaller picks test programs from the initial corpus with more raw coverage.

Figure 9(b) shows the coverage achieved by three optimizations compared to the default Syzkaller. From our experiments, we can still find that every optimization outperforms the default Syzkaller. During 24 hours, Thunderkaller with three optimizations can improve code coverage by about 12.3%, 6.1%, and 7.7% on average, respectively.

Fuzzing with various kernels, results in Figure 9(c), 9(d) show the coverage achieved by Syzkaller and Thunderkaller. On kernel 5.10, Thunderkaller and Syzkaller cover 75774, 65615 unique basic blocks on average, respectively, improving code coverage by 15.5%. While on kernel 5.15, Thunderkaller increases code coverage by about 35.8% during 24 hours.

### *D. Crash Detection Evaluation (RQ3)*

► **Crash Detection.** To evaluate the crash detection ability, we run Thunderkaller and the default Syzkaller 10 times and 24 hours each time, comparing the time-to-exposure (TTE) of unique bugs and their total number. The result in Table V shows the unique bugs and the TTE of bugs that we find. Thunderkaller discovers 496 crashes (46 unique), while the default Syzkaller discovers 196 crashes (25 unique) each time on average. 36 unique bugs are both triggered by Thunderkaller and Syzkaller. 32 out of 36 unique bugs are triggered faster than the default Syzkaller. Among these bugs, Thunderkaller outperforms Syzkaller in detecting the same target bugs about 6.3× faster on average.

To detect previously unknown bugs shown in Table III, we use Thunderkaller to fuzz more Linux kernel versions for 7 days. After manual deduplication, Thunderkaller found 6 unique kernel bugs. We have reported them to maintainers and one of them has been assigned a CVE.

► **False Negative Analysis.** Thunderkaller may miss some bugs for two reasons. First, KID changes the instrumentation which may affect bugs detection capability. The second reason is that Thunderkaller removes KASAN's checks during the

TABLE III
THUNDERKALLER FOUND 6 UNIQUE LINUX KERNEL BUGS.

| Kernel | Crash Type | Function | Status |
|---|---|---|---|
| 5.10 | Null-ptr-deref | gfs2_evict_inode | CVE-2023-3212 |
| 5.10 | Task hung | blkdev_open | Confirmed |
| 5.10 | Page fault | lookup_one_len | Reported |
| 5.10 | Page fault | tun_chr_write_iter | Reported |
| 5.10 | Page fault | jfs_evict_inode | Reported |
| 5.10 | Out-of-memory | io_alloc_req | Fixed |

TABLE IV
FALSE NEGATIVE ANALYSIS BASED ON PREVIOUS CRASHES.

| Crash Type | Function | Commit | CVE # | Reproducible | |
|---|---|---|---|---|---|
| | | | | Syz* | Th* |
| Protection fault | nft_set_elem_expr_alloc | ad9f151e560b | CVE-2021-46283 | ✓ | ✓ |
| Protection fault | sockfs_setattr$^U$ | 6d8c50dcb029 | CVE-2018-12232 | ✓ | ✓ |
| Page faul | csum_partial | 9cf448c200ba | CVE-2021-39633 | ✓ | ✓ |
| Null-ptr-deref | filp_close | 3b0462726e7e | CVE-2021-4154 | ✓ | ✓ |
| Null-ptr-deref | l2cap_chan_put | 1bff51ea59a9 | CVE-2021-3752 | ✓ | ✓ |
| Kernel bug | fuse_get_acl$^U$ | 5d069dbe8aaf | CVE-2020-36322 | ✓ | ✓ |
| Deadlock | scheduler_tick$^U$ | a83829f56c7c | (N/A) | ✓ | ✓ |
| Invalid-free | security_tun_dev_free_security | 158b515f703e | (N/A) | ✓ | ✓ |
| Out-of-bounds | strcpy$^U$ | 49d31c2f389a | CVE-2017-7533 | × | × |
| Use-after-free | ip_mc_drop_socket$^U$ | 657831ffc38e | CVE-2017-8890 | × | × |
| Use-after-free | __isofs_iget | e96a1866b405 | (N/A) | ✓ | ✓ |
| Use-after-free | __lock_sock | 5ec7d18d1813 | (N/A) | ✓ | ✓ |
| Use-after-free | vlan_dev_real_dev | 563bcbae3ba2 | (N/A) | ✓ | ✓ |
| Use-after-free | alloc_ucounts | 345daff2e994 | (N/A) | ✓ | ✓ |

Syz*: Syzkaller; Th*: Thunderkaller; $^U$: Undoing patch.

TABLE V

24-hour bug detection for 10 runs of Thunderkaller and Syzkaller. The third and fourth columns show the minimum time-to-exposure (TTE) of bugs among 10 runs. The fifth and sixth columns show the total number of triggered runs. The last column shows whether Syzbot has already detected this bug. (Bugs uniquely found by Thunderkaller are omitted.)

| Crash Type | Function | Min. TTE | | Crash Discovered | | |
|---|---|---|---|---|---|---|
| | | Th* | Syz* | Th* | Syz* | Syzbot |
| Use-after-free | fw_load_sysfs_fallback | **00h48m** | 06h58m | 10 | 3 | ✓ |
| Use-after-free | kill_pending_fw_fallback_reqs | **00h50m** | 07h14m | 10 | 4 | ✓ |
| Use-after-free | fb_mode_is_equal | **02h00m** | 03h03m | 10 | 10 | ✓ |
| Use-after-free | ntfs_test_inode | **14h16m** | 21h04m | 3 | 2 | ✓ |
| Use-after-free | lock_sock_nested | **01h30m** | 22h25m | 3 | 2 | ✓ |
| Use-after-free | diFree | **04h50m** | 06h10m | 1 | 1 | ✓ |
| Out-of-bounds | soft_cursor | **01h14m** | (N/A) | 2 | 0 | ✓ |
| Out-of-bounds | leaf_paste_entries | **10h04m** | 20h12m | 10 | 7 | ✓ |
| Inconsistent lock state | waiting for DEV to become free | **04h00m** | 04h31m | 8 | 5 | ✓ |
| Protection fault | cgroup_file_write | **02h42m** | 13h13m | 5 | 4 | ✓ |
| Page fault | imageblit | 08h46m | **06h34m** | 5 | 8 | ✓ |
| Warning | account_page_dirtied | **00h14m** | 06h47m | 10 | 10 | ✓ |
| Warning | f2fs_is_valid_blkaddr | 04h12m | **03h38m** | 8 | 10 | ✓ |
| Warning | kthread_is_per_cpu | **07h48m** | 15h38m | 7 | 4 | ✓ |
| Warning | sta_info_insert_rcu | **04h04m** | 07h02m | 10 | 9 | ✓ |
| Warning | floppy_queue_rq | **08h36m** | 14h26m | 6 | 4 | ✓ |
| Warning | pwq_unbound_release_workfn | **06h02m** | 18h21m | 9 | 8 | ✓ |
| Warning | submit_bio_checks | **00h43m** | 05h21m | 6 | 3 | ✓ |
| Warning | xfs_destroy_mount_workqueues | **03h21m** | 05h55m | 1 | 2 | ✓ |
| Warning | j1939_sk_queue_activate_next | 18h21m | **12h13m** | 1 | 1 | ✓ |
| Warning | send_hsr_supervision_frame | **00h22m** | 01h06m | 10 | 2 | ✓ |
| Warning | hci_conn_timeout | **01h20m** | 06h22m | 10 | 3 | ✓ |
| Warning | ieee80211_bss_info_change-_notify | **01h55m** | 06h19m | 10 | 1 | ✓ |
| Warning | close_fs_devices | 04h22m | **03h16m** | 10 | 10 | ✓ |
| Warning | netdev_run_todo | **08h01m** | 08h59m | 5 | 2 | ✓ |
| Warning | nbd_dev_add | (N/A) | 01h46m | 0 | 1 | ✓ |
| Task hung | hub_port_init | **04h09m** | 19h44m | 8 | 1 | ✓ |
| Task hung | __unmap_and_move | **00h37m** | 07h55m | 7 | 3 | ✓ |
| Task hung | blkdev_put | **03h36m** | 12h55m | 7 | 2 | ✓ |
| Task hung | p9_fd_close | **00h28m** | 11h40m | 8 | 1 | ✓ |
| Task hung | sync_inodes_sb | **01h18m** | 02h50m | 8 | 9 | ✓ |
| Task hung | gfs2_make_fs_ro | **13h16m** | 17h51m | 1 | 2 | ✓ |
| Task hung | do_proc_bulk | **00h44m** | 07h55m | 8 | 2 | ✓ |
| Task hung | io_uring_cancel_task_requests | **02h32m** | 09h13m | 6 | 1 | ✓ |
| Deadlock | console_lock_spinning_enable | **04h05m** | 07h16m | 10 | 10 | ✓ |
| Deadlock | console_trylock_spinning | **04h06m** | 19h18m | 8 | 3 | ✓ |
| Kernel bug | do_journal_end | **00h34m** | 04h05m | 2 | 1 | ✓ |
| Kernel bug | f2fs_new_node_page | (N/A) | 22h13m | 0 | 1 | ✗ |
| Kernel bug | gfs2_glock_nq | **03h51m** | 06h07m | 6 | 1 | ✓ |

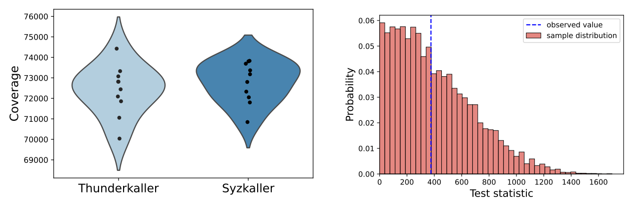
Syz*: Syzkaller; Th*: Thunderkaller.

minimization and triage phases of fuzzing.

To estimate how the kernel changes have affected the error detection, we manually run the 14 bugs of the benchmark. Table IV shows the results. 12 bugs can be reproduced both on Thunderkaller and the default Syzkaller. But two bugs can not be reproduced on either Thunderkaller or Syzkaller because we failed to undo the patches. In conclusion, we do not observe KID itself introducing false negatives.

To evaluate the false negatives introduced by removing KASAN's checks under the minimization and triage phases, we perform an experiment running Thunderkaller and Syzkaller for 24 hours, repeating 10 times, then compare how many runs a given unique bugs found by either system, the details of bugs and counts are listed in Table V.

In total, KASAN captures 8 unique bugs in Thunderkaller, while discovering 7 unique bugs in the default Syzkaller. Among these bugs, Thunderkaller detects all bugs triggered by Syzkaller. 2 unique bugs are found only in Syzkaller runs, and both are found in 1 run. Thunderkaller catches neither



(a) Coverage distribution (b) Permutation test

Fig. 10. Code coverage of Thunderkaller and Syzkaller in 10 runs.

of these 2 bugs, and one of these bugs is not found by Syzbot, which can be attributed to variation among runs and the randomness of the fuzzing strategy. Therefore, eliminating KASAN's checks during the minimization and triage phases of Syzkaller will not undermine the detection ability of KASAN.

### *E. Does the optimization skipping blocking syscalls undermine the code coverage? (RQ4)*

If this optimization falsely skips the long-running syscalls as blocking syscalls, it will harm the code coverage. To evaluate this, we replay the 24-hour corpus by Syzkaller, and Thunderkaller with only this optimization enabled, respectively, comparing the code coverage collected by each other. Even though replaying the same corpus on the default Syzkaller several times, the coverage collected each time is different. Since there are some non-deterministic behaviors, such as thread interleaving, interrupting, etc.. Therefore, we replay the same corpus 10 times and utilize the permutation test [3] to verify whether this optimization would harm the coverage.

Figure 10 shows the results of each experiment. We calculate the `p-value`, equal to 0.467 greater than 0.05. Consequently, this optimization would not undermine the code coverage.

## VII. Discussion and Future Work

**Debloating KASAN.** Currently, Thunderkaller only skips over KASAN's checks of load/store instructions, and stack variables. We will improve our implementation to skip over the rest of the checks. Moreover, we do not do stack malloc since we implement the jmp instructions by the inline assembly.

**Complex Syscalls.** We find some syscalls are disproportionately slow. For example, `syz_mount_image` aims to mount custom images and execute random strategies to detect file systems. There are other syscalls focusing on testing Linux `USB` drivers, which are asynchronous by nature, and require long time waiting for devices to respond. Our optimization techniques do not have a significant impact on these syscalls.

**Improving the Concurrent Performance.** Syzkaller can perform multiple fuzzing instances inside a single guest OS. Various performance bottlenecks may exist when parallel fuzzing of Syzkaller, which is our future research direction.

## VIII. Related Work

**Kernel Fuzzing.** Several excellent kernel fuzzers have discovered plenty of vulnerabilities. Some focus on improving the fuzzing algorithm [40], [30], [37], [13]. For example, SyzVegas utilizes the reinforcement-learning algorithm to improve the fuzzing strategy of Syzkaller. Moonshine analyzes the dependency between system calls and generates high-quality initial seeds for Syzkaller. FastSyzkaller [14] utilizes the N-Gram model to optimize the test programs generation process. Furthermore, some kernel fuzzers aim to detect race bugs of OS kernels [12], [44], [6]. While DIFUZE [4] proposes an interface-aware fuzzing tool via static analysis to detect kernel driver bugs. Agamotto [36] implements a lightweight snapshot to accelerate kernel driver fuzzing.

**Use of Instrumentation in Fuzzing.** Instrumentation help fuzzers catch more bugs by providing exploration directions to guide fuzzing. For example, some fuzzers utilize instrumentation by compilers to collect coverage [26], [34], [32], [46], using code coverage to generate the following input and directly drive fuzzing toward non-explored paths. Besides, other fuzzers also use instrumentation to collect other knowledge to guide fuzzing, such as program states [49], the stack frame [31], and sanitizer's checks [29], etc.. Sanitizers also use instrumentation to catch bugs. They instrument targets at compile time and detect bugs during development and testing, including memory errors [15], [33], [48], race bugs, and undefined behavior errors [43].

**Reducing Instrumentation Overhead.** The methods used to reduce cost caused by instrumentation can be divided into three aspects. (1) The first method aims at reducing unnecessary instrumentation at compile time. UnTracer [28] only traces the coverage-increasing input to eliminate the unnecessary cost. Instrim [10] instruments the basic block depending on the knowledge of the control flow graph. RIFF [41] precomputes information used at runtime to reduce the cost of collecting coverage. ASAP [39] profiles the target before fuzzing. Recently, ODIN [42] proposes the on-the-fly recompilation technique to reduce the cost. (2) The second is binary rewriting. Zeror [50] utilizes the instrumentation points to replace and recover the original coverage hooks. UnTracer uses the same method to rewrite targets during fuzzing. (3) The last is reducing the instrumentation cost by recompilation. ASAP recompiles the target eliminating unnecessary checks. ODIN only recompiles the changed and associated fragments of targets. Besides, some works reduce the cost in other ways. For example, CollAFL [5] designs a new hash algorithm and optimizes the way of recording coverage.

**Reducing Overhead of Operating System.** Reducing the OS's overhead is another research for improving fuzzing performance. AFL utilizes the fork server mode to eliminate the overhead caused by `execve` [27], and it implements the persistent mode further to reduce the cost of `fork`. Xu *et al.* [45] implement a new system call to improve fuzzing performance by storing the states of fuzzing. Invoking system calls is intensive when fuzzing kernels. FlexSC [35] implements the exception-less system calls to reduce the cost of invoking syscalls. Reducing the execution time of system calls is another prominent problem.

**Main Differences.** Optimizations of kernel fuzzing strategies are orthogonal to Thunderkaller, and most of them can also benefit from it. In our experiment, applying Thunderkaller to Moonshine, it achieves better performance. Different from Zeror and UnTrace, etc., we propose a new technique to reduce unnecessary instrumentation during kernel fuzzing. Zeror, etc., rewrite instrumentation at runtime, which introduces extra overhead. Besides, dynamic rewriting kernel binary is not easy and harms the performance of systems. Hence, we propose KID, which just patches instructions during the booting process. Using KID to accomplish multiple kernel copies, more improvements can be achieved.

## IX. Conclusion

In this paper, we have measured and dissected the cost of program execution, invoking system calls, and kernel instrumentation. Based on the measurement, we propose Thunderkaller, which assembles three techniques to optimize the performance bottleneck without changing the default fuzzing strategy of Syzkaller. We apply these optimizations to Moonshine, which is orthogonal to Thunderkaller. Moreover, we evaluate Thunderkaller by enabling all system calls and the default kernel configuration on different Linux kernel versions. During 24 hours, it demonstrates that compared to the default Syzkaller, Thunderkaller can obtain $2.8\times$ speedup, achieve 25.8% more coverage, detect 21 more unique bugs, and trigger the same bugs $6.3\times$ faster. Besides, Thunderkaller also discovered 6 unique kernel bugs and got a CVE.

## Acknowledgement

We sincerely thank all the anonymous reviewers for their valuable feedback that greatly helped us to improve this paper. This work is supported in part by the National Key Research and Development Program of China (2021YFB2701000), the National Natural Science Foundation of China (61972224), and the Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006.

## References

[1] Linux kernel selftests. https://www.kernel.org/doc/html/v4.15/dev-tools/kselftest.html

[2] Open posix tests. http://posixtest.sourceforge.net

[3] Permutation Test. https://en.wikipedia.org/wiki/Permutation_test

[4] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., Vigna, G.: Difuze: Interface aware fuzzing for kernel drivers. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2123–2138 (2017)

[5] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z.: Collafl: Path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 679–696 (2018). https://doi.org/10.1109/SP.2018.00040

[6] Gong, S., Altinbüken, D., Fonseca, P., Maniatis, P.: Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 66–83 (2021)

[7] google: Syzbot. https://syzkaller.appspot.com/upstream

[8] google: Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller

[9] google: the default kernel configuration. https://github.com/google/syzkaller/blob/master/dashboard/config/linux/stable-5.4-kasan.config

[10] Hsu, C.C., Wu, C.Y., Hsiao, H.C., Huang, S.K.: Instrim: Lightweight instrumentation for coverage-guided fuzzing. In: Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research (2018)

[11] Jeon, Y., Han, W., Burow, N., Payer, M.: {FuZZan}: Efficient sanitizer metadata design for fuzzing. In: 2020 USENIX Annual Technical Conference (USENIX ATC 20). pp. 249–263 (2020)

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

[13] Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee, B.: HFL: Hybrid Fuzzing on the Linux Kernel. In: NDSS (2020)

[14] Li, D., Chen, H.: Fastsyzkaller: Improving fuzz efficiency for linux kernel fuzzing. In: Journal of Physics: Conference Series. vol. 1176, p. 022013. IOP Publishing (2019)

[15] Li, Y., Tan, W., Lv, Z., Yang, S., Payer, M., Liu, Y., Zhang, C.: Pacmem: Enforcing spatial and temporal memory safety via arm pointer authentication. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 1901–1915. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3560598, https://doi.org/10.1145/3548606.3560598

[16] Linux Developers: FUSE. https://www.kernel.org/doc/html/latest/filesystems/fuse.html

[17] Linux Developers: Getpid(2). https://man7.org/linux/man-pages/man2/getppid.2.html

[18] Linux Developers: kasan. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[19] Linux Developers: kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html

[20] linux Developers: Kernel Configuration. https://wiki.gentoo.org/wiki/Kernel/Configuration

[21] Linux Developers: Kernel level exception handling. https://www.kernel.org/doc/html/latest/x86/exception-tables.html

[22] Linux Developers: The Mode Switch in Modern Processor. https://imgur.com/bfgu0EK

[23] linux Developers: Thread Status. https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt

[24] LLVM Developers: The LLVM Compiler Infrastructure. http://llvm.org

[25] LTP developers: Linux testing projects. https://linux-test-project.github.io

[26] M. Zalewski: American fuzzy lop. http://lcamtuf.coredump.cx/afl/ (2017)

[27] Michal Zalewski: Fuzzing random programs without execve(). http://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html (2014)

[28] Nagy, S., Hicks, M.: Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 787–802 (2019). https://doi.org/10.1109/SP.2019.00069

[29] Österlund, S., Razavi, K., Bos, H., Giuffrida, C.: Parmesan: Sanitizer-guided greybox fuzzing. In: Proceedings of the 29th USENIX Conference on Security Symposium. pp. 2289–2306 (2020)

[30] Pailoor, S., Aday, A., Jana, S.: {MoonShine}: Optimizing {OS} Fuzzer Seed Selection with Trace Distillation. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 729–743 (2018)

[31] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: Application-aware evolutionary fuzzing. In: NDSS. vol. 17, pp. 1–14 (2017)

[32] Robert Swiecki: honggfuzz. http://honggfuzz.com/ (2018)

[33] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: {AddressSanitizer}: A fast address sanity checker. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12). pp. 309–318 (2012)

[34] Serebryany, K.: Continuous fuzzing with libfuzzer and addresssanitizer. In: 2016 IEEE Cybersecurity Development (SecDev). pp. 157–157. IEEE (2016)

[35] Soares, L., Stumm, M.: Flexsc: Flexible system call scheduling with exception-less system calls. In: Osdi. vol. 10, pp. 33–46 (2010)

[36] Song, D., Hetzelt, F., Kim, J., Kang, B.B., Seifert, J.P., Franz, M.: Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2541–2557 (2020)

[37] Sun, H., Shen, Y., Wang, C., Liu, J., Jiang, Y., Chen, T., Cui, A.: Healer: Relation learning guided kernel fuzzing. In: Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. pp. 344–358 (2021)

[38] Vangoor, B.K.R., Tarasov, V., Zadok, E.: To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems. In: 15th USENIX Conference on File and Storage Technologies (FAST 17). pp. 59–72 (2017)

[39] Wagner, J., Kuznetsov, V., Candea, G., Kinder, J.: High system-code security with low overhead. In: 2015 IEEE Symposium on Security and Privacy. pp. 866–879 (2015). https://doi.org/10.1109/SP.2015.58

[40] Wang, D., Zhang, Z., Zhang, H., Qian, Z., Krishnamurthy, S.V., Abu-Ghazaleh, N.: {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2741–2758 (2021)

[41] Wang, M., Liang, J., Zhou, C., Jiang, Y., Wang, R., Sun, C., Sun, J.: Riff: Reduced instruction footprint for coverage-guided fuzzing. In: USENIX Annual Technical Conference. pp. 147–159 (2021)

[42] Wang, M., Liang, J., Zhou, C., Wu, Z., Xu, X., Jiang, Y.: Odin: on-demand instrumentation with on-the-fly recompilation. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 1010–1024 (2022)

[43] Wang, X., Zeldovich, N., Kaashoek, M.F., Solar-Lezama, A.: Towards optimization-safe systems: Analyzing the impact of undefined behavior. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. pp. 260–275 (2013)

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

[45] Xu, W., Kashyap, S., Min, C., Kim, T.: Designing new operating primitives to improve fuzzing performance. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2313–2328 (2017)

[46] You, W., Wang, X., Ma, S., Huang, J., Zhang, X., Wang, X., Liang, B.: Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: 2019 IEEE symposium on security and privacy (SP). pp. 769–786. IEEE (2019)

[47] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu: Debloating address sanitizer. In: Usenix Security Symposium (2022)

[48] Zhang, J., Wang, S., Rigger, M., He, P., Su, Z.: {SANRAZOR}: Reducing redundant sanitizer checks in {C/C++} programs. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). pp. 479–494 (2021)

[49] Zhao, B., Li, Z., Qin, S., Ma, Z., Yuan, M., Zhu, W., Tian, Z., Zhang, C.: {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3273–3289 (2022)

[50] Zhou, C., Wang, M., Liang, J., Liu, Z., Jiang, Y.: Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 858–870 (2020)