Razzer: Finding Kernel Race Bugs through Fuzzing

Dae R. Jeong[†] Kyungtae Kim[‡] Basavesh Shivakumar[‡] Byoungyoung Lee[‡*] Insik Shin[†]

[†] *Computer Science, KAIST,*
[‡] *Computer Science, Purdue University,*
[*] *Electrical and Computer Engineering, Seoul National University*

***Abstract*—A data race in a kernel is an important class of bugs, critically impacting the reliability and security of the associated system. As a result of a race, the kernel may become unresponsive. Even worse, an attacker may launch a privilege escalation attack to acquire root privileges.**

In this paper, we propose RAZZER, a tool to find race bugs in kernels. The core of RAZZER is in guiding fuzz testing towards potential data race spots in the kernel. RAZZER employs two techniques to find races efficiently: a static analysis and a deterministic thread interleaving technique. Using a static analysis, RAZZER identifies over-approximated potential data race spots, guiding the fuzzer to search for data races in the kernel more efficiently. Using the deterministic thread interleaving technique implemented at the hypervisor, RAZZER tames the non-deterministic behavior of the kernel such that it can deterministically trigger a race. We implemented a prototype of RAZZER and ran the latest Linux kernel (from v4.16-rc3 to v4.18-rc3) using RAZZER. As a result, RAZZER discovered 30 new races in the kernel, with 16 subsequently confirmed and accordingly patched by kernel developers after they were reported.

I. Introduction

Data races are detrimental to the reliability and security of the underlying system. Particularly for the kernel, data races are the root cause of various harmful behaviors. If a data race introduces circular lock behavior, the kernel can become unresponsive due to the resulting deadlock. If safety assertions residing in the kernel arise, the kernel would reboot itself, resulting in a denial-of-service. Especially from the perspective of security, data races may turn into critical security attacks if they lead to traditional memory corruptions in the kernel (e.g., traditional buffer overflows, use-after-free, etc.), which may allow privilege escalation attacks, as observed in kernel exploits abusing previously known data races, such as CVE-2016-8655 [26], CVE-2017-2636 [28], and CVE-2017-17712 [27].

In response to these issues associated with data races, there have been extensive research efforts with regard to avoiding, preventing, or detecting them. However, to the best of out knowledge, each technique has certain limitations, mainly due to the fact that a data race inherently stems from the non-deterministic behavior of the kernel. More precisely, understanding the data race requires not only precise control-flow and data-flow information but also precise concurrency execution information, which is heavily impacted by many other external factors of the underlying system (such as scheduling, synchronization primitives, etc.).

In this paper, we propose RAZZER, a fuzzing based data race detector. The key insight behind RAZZER is that it drives the fuzz testing towards potential data race spots in the kernel. To achieve this, RAZZER takes a hybrid approach, leveraging both static and dynamic analyses, to amplify the advantages of two techniques while complementing their disadvantages. First, RAZZER carries out a static analysis to obtain over-approximated, potential data race points. Based on information on such potential data race points, RAZZER performs a two-staged dynamic fuzz testing. The first stage involves a single-thread fuzz testing, which focuses on finding a single-thread input program that executes potential race points (without considering whether the program indeed triggers the race). The second stage is multi-thread fuzz testing. It constructs a multi-thread program, which further leverages a tailored hypervisor intentionally to stall its execution at potential data race points. As such, RAZZER avoids any external factors to render the race behavior deterministic, making it an efficient fuzzer for data races.

We implemented RAZZER's static analysis with an LLVM pass to conduct a points-to analysis, and the hypervisor was developed by modifying QEMU and KVM for x86-64. The corresponding two-staged fuzzing framework is developed to fuzz system call interfaces of the kernel, while leveraging static analysis results as well as the tailored hypervisor. Once RAZZER identifies a data race, it outputs not only the input program to reproduce the race but also provides a detailed report that facilitates an easy understanding of the root cause of the race.

Our evaluation of RAZZER demonstrates that RAZZER is truly a ready-to-be-deployed race detection tool. We applied RAZZER to the latest versions of the Linux kernel (from `v4.16-rc3` to `v4.18-rc3`) at the time of writing this paper, and RAZZER found 30 new data races in the kernel. We have reported these; 16 races have been confirmed and patches of 14 have been submitted thus far by the kernel developers. Moreover, 13 races have been merged into various affected kernel versions, including the mainline kernel as well.

To highlight the effectiveness of RAZZER in finding data races, we performed a restricted comparison study (§III-C) with other state-of-the-art tools, specifically `Syzkaller` [42] (i.e., a kernel fuzzer developed by Google) and `SKI` [16] (i.e., an academic research prototype which randomizes the thread

Correspondence to: Byoungyoung Lee (byoungyoung@snu.ac.kr) interleaving to find races in the kernel). Summarizing this comparison study, RAZZER significantly outperformed both tools in identifying three race issues. Compared to Syzkaller, RAZZER takes much less time to find a race, ranging from 23 to 85 times (at minimum). Compared to SKI, RAZZER was far more effective in exploring thread interleaving cases to find a race, ranging from 30 to 398 times.

Furthermore, our reporting experience with the kernel developers suggests that RAZZER's detailed analysis report assists developers to fix a reported race easily. More specifically, because RAZZER points out a specific race location (i.e., two memory access instructions in the kernel incurring the race) as well as the call stack when the race occurs, developers were easily able to determine the root cause of the race and develop a patch for it. As an extreme example, once we reported newly discovered races through LKML [4], two of our reported races were patched (within 20 minutes and in 2 hours by the respective kernel developers). In light of the common knowledge about data races, particularly the difficulty in determining the root causes, we believe that our reporting experience suggests the strong potential to facilitate easy, low-cost patching for data races.

This paper makes the following contributions:

- **Race-Oriented Fuzzer**. We present a new fuzz testing mechanism which is specifically designed to detect races in the kernel. It leverages both static and dynamic analysis techniques to focus its fuzzing on potential race points.
- **Robust Implementation**: We implemented RAZZER based on various industry-strength frameworks, ranging from KVM/QEMU to LLVM. It requires no manual modification of the target kernel to be analyzed. We believe its implementation is robust enough given that it can easily support the latest Linux kernel without any manual intervention.
- **Practical Impacts**: We ran RAZZER on the Linux kernel, and it found 30 races, where 16 races were already confirmed and accordingly fixed by the respective kernel developers. We will open source of RAZZER[1] such that kernel developers and researchers can benefit from using RAZZER.

This paper is organized as follows. §II defines the problem scope and identifies the design requirements of RAZZER. §III presents the design detail of RAZZER, and §IV describes its implementation. §V presents various evaluation results of RAZZER. §VI discusses related work, and §VIII concludes the paper.

II. Problem Scope and Design Requirements

In this section, we first define the data race, which is the main focus of this paper. We then summarize existing approaches to find race bugs in the target program. Lastly, we describe our motivating example and briefly explain our approach.

A. Problem Scope and Terminology

This paper aims to identify data races in system software. A data race is behavior in which the output is dependent on the sequence or the timing of other non-deterministic events. More specifically, a data race occurs when two memory access instructions in a target program meet the following three conditions: (i) they access the same memory location. (ii) at least one is a write instruction. and (iii) they are executed concurrently.

If all above three conditions above are met, the memory accesses performed by the two memory instructions can be non-deterministic, rendering computational result to vary depending on the execution order. Throughout this paper, we use the term $\texttt{RacePair}_{\text{cand}}$ to denote two memory access instructions that may satisfy the three conditions described above (i.e., a candidate race pair), and we use $\texttt{RacePair}_{\text{true}}$ for those that are confirmed to meet the three conditions (i.e., a true race pair, which is a subset of $\texttt{RacePair}_{\text{cand}}$).

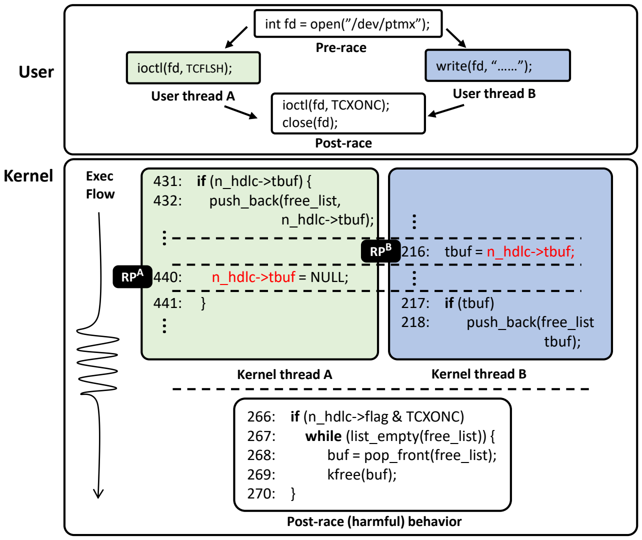
Data races can be further classified into two groups: benign and harmful. A benign race is an expected (or intentional) data race by developers, tolerating a potential deviation in the computational results. For example, it is common to allow data races in maintaining performance counters, as doing so can avoid sluggish data contentions on a performance counter variable (while tolerating a small error of a counter value). We use the term $\texttt{RacePair}_{\text{benign}}$ to refer to two memory instructions raising such a benign race.

A harmful race is a data race that negatively impacts a runtime behavior of a program, and we use the term $\texttt{RacePair}_{\text{harm}}$ for this case. Due to the non-deterministic computational result, there can be various aftermaths of a harmful race, including deadlocks, raising safety assertions in the kernel, and memory safety violations (including stack/heap buffer overflows, use-after-free, and double-free), etc. We note that these aftermaths are critical to kernel's reliability and security: deadlocks may make the kernel unresponsive, violating safety assertions may cause a reboot the kernel, and memory safety violations allow a privilege escalation attack. We use the term $\texttt{RacePair}_{\text{harm}}$ to refer to two memory instructions in this case.

Summarizing the terminology, $\texttt{RacePair}_{\text{cand}}$ refers to two memory access instructions that may cause a race, and $\texttt{RacePair}_{\text{true}}$ refers to two instructions that are confirmed to cause a race. $\texttt{RacePair}_{\text{true}}$ can be further classified into two groups: $\texttt{RacePair}_{\text{benign}}$ and $\texttt{RacePair}_{\text{harm}}$.

Race Example: CVE-2017-2636. To clarify how a data race occurs, we take the real-world example from the Linux kernel, CVE-2017-2636 [28], as illustrated in Figure 1. This race can be induced by an adversarial multi-threaded user program which invokes a specific list of syscalls in a specific order. The data race occurs while the kernel is processing such syscalls, which in turn causes a double-free issue that allows a privilege escalation attack. In particular, the key to trigger the race consists of two system calls, `ioctl(fd, TCFLSH)` and `write(fd, ...)`, each of which is executed concurrently by user thread A or B, respectively. In response, the kernel starts executing the corresponding syscall handlers on an individual kernel thread (i.e., kernel thread A or B). In order to trigger the race, kernel thread A first checks if `n_hdlc->tbuf` (`n_hdlc` is a structure allocated in the heap) is a null pointer (line 431

[1] https://github.com/compsec-snu/razzer



```
int fd = open("/dev/ptmx");
```

```
ioctl(fd, TCFLSH);
```

```
write(fd, "……");
```

```
ioctl(fd, TCXONC);
close(fd);
```

```
431:  if (n_hdlc->tbuf) {
432:     push_back(free_list,
                 n_hdlc->tbuf);
  ⋮
440:        n_hdlc->tbuf = NULL;
441:  }
  ⋮
```

```
  ⋮
216:  tbuf = n_hdlc->tbuf;
  ⋮
217:  if (tbuf)
218:       push_back(free_list
                   tbuf);
```

```
266:  if (n_hdlc->flag & TCXONC)
267:     while (list_empty(free_list)) {
268:         buf = pop_front(free_list);
269:         kfree(buf);
270:  }
```

Fig. 1: A simplified race example on CVE-2017-2636. As a user program executes two syscalls concurrently, a data race on `n_hdlc->tbuf` may occur depending on the execution order, which leads to a double-free issue allowing an attacker to launch privilege escalation attack.

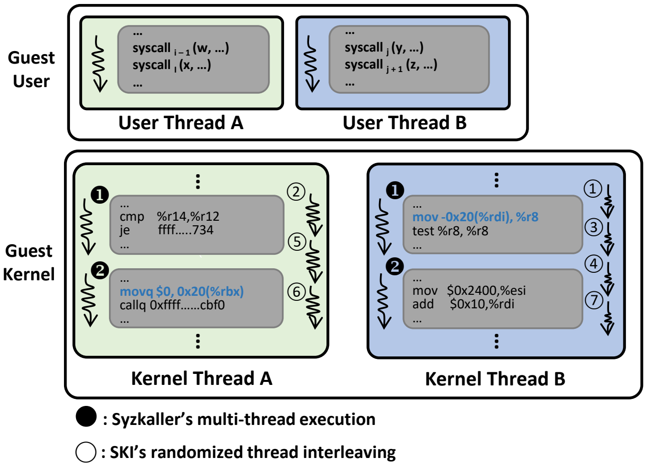
in Figure 1). If not, it pushes the pointer to the free list (line 432) such that actual free operations will occur later. Then, two memory instructions, `n_hdlc->tbuf = null` in kernel thread A (i.e., $\mathtt{RP}^{\mathtt{A}}$ in line 440) and `tbuf = n_hdlc->tbuf` (i.e., $\mathtt{RP}^{\mathtt{B}}$ in line 216) in kernel thread B, enter into data races. More specifically, if the address of `n_hdlc->tbuf` is identical in kernel threads A and B, the computational result will differ depending on the execution orders of these instructions. That is, if $\mathtt{RP}^{\mathtt{B}}$ is executed first and $\mathtt{RP}^{\mathtt{A}}$ is executed later, as ordered in Figure 1, all of the subsequent instructions of kernel thread B will see a non-null pointer stored in a local variable `tbuf` despite the fact that it has already been pushed to the `free_list` by the kernel thread A. Thus, `tbuf` will remain a valid pointer (line 217); hence, `n_hdlc->tbuf` will be pushed to the `free_list` (line 218) again by kernel thread B. Note that if $\mathtt{RP}^{\mathtt{A}}$ is executed first, `tbuf` of B will hold a null pointer and thus a redundant free list push would not arise.

To launch a security attack based on this race, proper post-race harmful behaviors are necessary. In this example, `ioctl(fd, TCXONC)` from the user program should be invoked to perform the actual free operation stored in the free list, which eventually leads to the double-free.

B. Design Requirements

In this subsection, we first identify desirable design requirements to discover data races in the kernel. Then we revisit existing tools from the perspective of meeting such requirements.

Design Requirements. One important design goal of RAZZER is to avoid any false positives in during the race detection process. Towards achieving this goal, we identify the following two desirable design requirements in order to discover data races in the kernel.



```
…
syscall_{i-1}(w, …)
syscall_i(x, …)
…
```

```
…
syscall_j(y, …)
syscall_{j+1}(z, …)
…
```

```
…
cmp   %r14,%r12
je    ffff.....734
…
```

```
…
movq $0, 0x20(%rbx)
callq 0xffff......cbf0
…
```

```
…
mov -0x20(%rdi), %r8
test %r8, %r8
…
```

```
…
mov   $0x2400,%esi
add   $0x10,%rdi
…
```

Fig. 2: Race detection mechanisms of `Syzkaller` and `SKI`

R1: *Find an input program which executes* $\mathtt{RacePair}_{cand}$*.* More precisely, an analysis should discover a multi-threaded user-land program, where each thread in the program executes each instruction in $\mathtt{RacePair}_{\mathrm{cand}}$.

R2: *Find a thread interleaving for the input program which executes* $\mathtt{RacePair}_{cand}$ *concurrently.* In other words, as R1 alone does not ensure that $\mathtt{RacePair}_{\mathrm{cand}}$ can be executed concurrently to trigger data races, the analysis should identify a specific thread interleaving case that executes $\mathtt{RacePair}_{\mathrm{cand}}$ concurrently.

We find that existing tools mostly focus on meeting only one of the two requirements above, limiting the effectiveness of discovering unknown races. Below, we present our analysis of the requirements of two general techniques to identify race bugs, namely fuzz testing and random thread interleaving, to discuss how well they meet the two aforementioned requirements.

Requirement Study: Traditional Fuzz Testing. Traditional fuzz testing (such as `AFL` to fuzz user-land programs and `Syzkaller` [42] to fuzz the kernel) focuses on R1, attempting to find inputs extending the kernel code coverage. Because R2 is not considered at all, it is not effective to discover data races. To better illustrate this limitation, the filled circles in Figure 2 depict the execution flow of `Syzkaller` as it finds race bugs. With regard to `Syzkaller`, the core of its fuzzing to find races is linked to its execution of randomly generated (or mutated) syscalls, where each syscall is again randomly assigned either to user thread A or B. Although this indeed interleaves the execution of syscalls into one of two kernel threads (i.e., $\mathrm{syscall}_{\mathrm{i-1}}$), its chance to trigger a race would be very low. According to our evaluation in §V-D1, `Syzkaller` failed to find races of three previously known race bugs given 10 hours. In contrast, RAZZER found all three races within from 7 mins to 26 mins, meaning that RAZZER was faster than `Syzkaller` 23 times to 85 times at least.

Requirement Study: Thread Interleaving Tools. Regarding random thread interleaving tools (such as SKI [16] or the PCT Algorithm [10]), their focus is in meeting R2, attempting to explore all possible thread interleaving cases for a specific (and static) input program. As they do not consider R1, they can only run existing programs (such as benchmarks) and thus cannot efficiently explore massive code spaces, leaving a majority part of the kernel untested. Moreover, because thread interleaving tools are based on random scheduling, their efficiency on R2 alone (i.e., simply searching all thread interleaving cases) is also severely limited. More precisely, in the beginning SKI randomly selects one kernel thread, and then executes it until encountering any memory access instruction. After stopping at the memory instruction, it randomly selects the kernel thread again, and repeats this process. For example, the unfilled circles in Figure 2 illustrate the execution flow of SKI. First, it selects kernel thread B (①) randomly, and executes until the next memory access instruction. It then randomly selects the kernel thread A (②) again and repeats this process. This explores all possible thread interleaving cases to identify race bugs for a given program, but large search spaces of interleaving cases would make it inefficient. According to our evaluation in §V-D2, $\texttt{SKI}_{\text{Emu}}$ (i.e., an emulated version of SKI) requires an exploration of interleaving cases from 6,435 to 27,132 to identify three previously known races. However, RAZZER only requires the exploring of 23 to 56 interleaving cases (i.e., RAZZER is 30 to 398 times more efficient than $\texttt{SKI}_{\text{Emu}}$).

III. Design

In this section, we describe the design details of RAZZER. The key idea behind RAZZER is in driving the analysis towards potential data race points in the kernel. In particular, RAZZER takes a hybrid approach, leveraging both static and dynamic analysis. First, RAZZER performs a static analysis to obtain over-approximated potential data race points. Next, RAZZER conducts a two-staged dynamic analysis. The first stage is single-thread fuzz testing, which focuses on identifying a single-thread input program that executes potential race points (attempting to meet R1). The second stage is multi-thread fuzz testing. During this stage, a multi-thread program is constructed with the help from the first stage, utilizing a custom hypervisor to control the thread interleaving deterministically (attempting to meet R2). Once a race is found, RAZZER outputs a concrete user program (i.e., a program triggering the data race) as well as the detailed root cause information (i.e., a report describing where the data race occurred) such that kernel developers can easily understand the root cause of the race and accordingly patch it.

In the following paragraphs, §III-A first describes how RAZZER performs its static analysis to obtain potential data race points. Then §III-B describes how RAZZER tailors the hypervisor to trigger the race deterministically for its dynamic analysis. Lastly, §III-C illustrates dynamic fuzz testing by RAZZER

A. Identifying Race Candidates

The goal of our static analysis is to identify all $\texttt{RacePair}_{\text{cand}}$ (i.e., a set of race candidate pairs) in the kernel, where each $\texttt{RacePair}_{\text{cand}}$ consists of two memory access instructions and may entail the potential to race at runtime. In general, a points-to analysis would be a popular choice to collect $\texttt{RacePair}_{\text{cand}}$ since it reasons about where each memory instruction points to. However, it is well known that the points-to analysis is limited in terms of accuracy and performance. With regard to accuracy, the points-to analysis is associated with high false positive rates as precise (and concrete) control/data-flow information can only be known at runtime. Even worse, it is more challenging to handle race issues because it requires not only precise control/data-flow information but also precise concurrency information, which is heavily impacted by external factors (such as scheduling or synchronization primitives). Moreover, given the time complexity of a typical points-to analysis (i.e., an Andersen analysis [6]) is $O(n^3)$, where n denotes the size of the program to be analyzed, it would take a very long time to analyze the entire kernel.

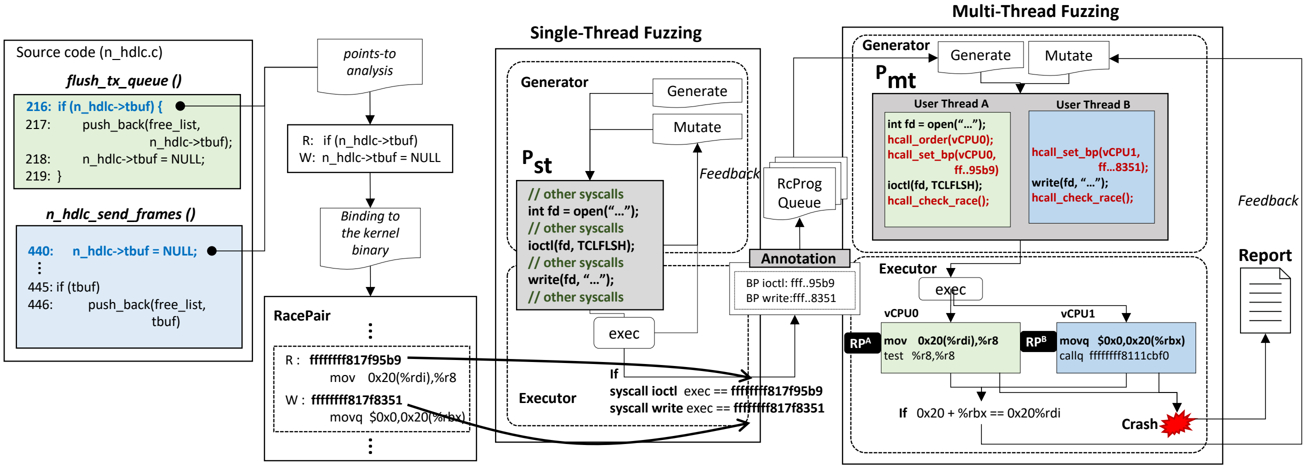
RAZZER addresses these issues with the following two approaches. First, to address accuracy issues, RAZZER allows the points-to analysis to over-approximate a $\texttt{RacePair}_{\text{cand}}$ set (i.e., some $\texttt{RacePair}_{\text{cand}}$ may not be $\texttt{RacePair}_{\text{true}}$), and it resolves false positive issues through its dynamic fuzz testing, as described later (§III-C). RAZZER's points-to analysis is context-insensitive and flow-insensitive but field-sensitive. Thus, it over-approximates $\texttt{RacePairs}_{\text{cand}}$ while excluding pairs that must not race (e.g., RAZZER determines that two instructions must not race if accessing different member variables in the structure).

Second, in order to mitigate performance issues, RAZZER performs a tailored partition analysis of the kernel. It partitions kernel objects according its module component, and performs a pre-analysis for every module. Especially for the Linux kernel, our point-to analysis partitions the kernel based on the directory structure in the source code repository (e.g., `kernel`, `mm`, `fs`, `drivers`), as each subdirectory represents a well-confined module. When performing a pre-analysis per module, RAZZER always provides core kernel modules as well, which remains necessary irrespective of the module being analyzed. For example, RAZZER always includes `fs` and `net/core`, as these two modules are used globally by many other sub-modules.

It is worth noting that our static analysis does not consider synchronization primitives in the kernel (e.g., `read_lock()`, `br_read_lock()`, `spin_lock_irqsave()`, `up()`). Leveraging this information would reduce the false positive rate (as it can help to determine memory pairs that must not race), and we leave this as future work.

B. Per-Core Scheduler in Hypervisor

As discussed earlier in §II, a race condition rarely manifests itself due to the non-deterministic and random nature of kernel's thread interleaving. Therefore, RAZZER runs the target kernel on a tailored virtualized environment such that RAZZER avoids non-deterministic behaviors from external events. In other words, RAZZER runs a multi-thread program in the guest user space, and this program attempts to trigger a race in the guest kernel. Moreover, for complete control over guest



```
216:  if (n_hdlc->tbuf) {
217:        push_back(free_list,
                      n_hdlc->tbuf);
218:        n_hdlc->tbuf = NULL;
219:  }
```

```
440:    n_hdlc->tbuf = NULL;
  ⋮
445: if (tbuf)
446:        push_back(free_list,
                      tbuf)
```

```
R:  if (n_hdlc->tbuf)
W:  n_hdlc->tbuf = NULL
```

```
R :  ffffffff817f95b9
         mov   0x20(%rdi),%r8
W :  ffffffff817f8351
         movq  $0x0,0x20(%rbx)
```

```
// other syscalls
int fd = open("…");
// other syscalls
ioctl(fd, TCLFLSH);
// other syscalls
write(fd, "…");
// other syscalls
```

```
If
syscall ioctl  exec == ffffffff817f95b9
syscall write exec == ffffffff817f8351
```

```
BP ioctl: fff..95b9
BP write:fff..8351
```

```
int fd = open("…");
hcall_order(vCPU0);
hcall_set_bp(vCPU0,
             ff..95b9)
ioctl(fd, TCLFLSH);
hcall_check_race();
```

```
hcall_set_bp(vCPU1,
             ff…8351);
write(fd, "…");
hcall_check_race();
```

```
mov   0x20(%rdi),%r8
test  %r8,%r8
```

```
movq  $0x0,0x20(%rbx)
callq ffffffff8111cbf0
```

```
If  0x20 + %rbx == 0x20%rdi
```

Fig. 3: Overall architecture of RAZZER

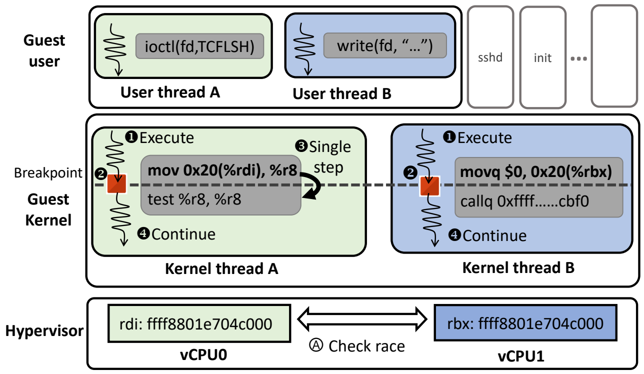


Fig. 4: Workflow of RAZZER's hypervisor

kernel's behavior with regard to thread interleaving, RAZZER modifies the hypervisor (i.e., a virtual machine monitor). RAZZER's modified hypervisor provides the following features for the guest kernel: (i) setting up a breakpoint per CPU core (more precisely, per virtual CPU core as they run on a virtual machine) (§III-B1); (ii) resuming the execution of kernel threads after the guest kernel hits breakpoints (§III-B2); and (iii) checking whether a race truly occurred due to a guest kernel (§III-B3). In the following section, we describe how RAZZER supports each of these three features, and we describe later how these hypervisor-level support features will be leveraged during the fuzzing (§III-C2).

1) Setup Per-Core Breakpoint: RAZZER provides a new hypercall interface, `hcall_set_bp()`, so that the guest kernel can setup a per-core breakpoint as needed. In particular, `hcall_set_bp()` is invoked by the guest kernel while using two parameters: (1) `vCPU_ID` specifies a virtual CPU (vCPU) on which RAZZER should install the breakpoint; and (2) `guest_addr` specifies an address in guest OS's address space where a breakpoint is to be installed. Once receiving this hypercall, the hypervisor installs the hardware breakpoint at `guest_addr`, which is only effective on the specified vCPU.

There are two particular tasks involved here: (i) accurately controlling the per-core execution behavior (more precisely, the per vCPU execution behavior as the guest kernel is virtualized), and (ii) determining whether a breakpoint is triggered by a specific kernel thread while running a specific syscall.

In order to achieve the first task, RAZZER utilizes a hardware breakpoint supported by a virtualized debug register, while ensuring that the breakpoint event is always delivered to the hypervisor first (instead of being delivered back to the guest kernel). In the case of the x86 architecture, RAZZER stores the guest address in a virtualized debug register. As this debug register is virtualized and thus maintained for every vCPU, RAZZER can ensure that the hardware breakpoint is triggered only if a corresponding vCPU is executing it. Moreover, to ensure that the hardware breakpoint event is delivered first to the hypervisor, RAZZER leverages the Virtual Machine Control Structure (VMCS) in Intel VT-x. VMCS contains an interrupt bitmap, where each bit corresponds to a guest's interrupt number. When an interrupt is raised and the corresponding interrupt bit is set in VMCS, the interrupt causes an immediate `VMEXIT` event, which is delivered subsequently to the hypervisor. Thus, RAZZER sets the bit in VMCS corresponding to the hardware breakpoint interrupt such that RAZZER can firstly monitor the hardware breakpoint.

The second task is related to the fact that while our hypervisor understands the vCPU context, it does not understand the kernel thread context. In other words, while the guest kernel is running a given user program, it may also run other user programs (e.g., `Xorg` and `sshd`) or kernel tasks (e.g.,`kworker` and `ksoftirqd`) that are not related to the given user program. In this case, if we simply install per vCPU breakpoint, the breakpoint can be triggered by such unrelated programs or kernel tasks as well. Because these breakpoint trigger events occur irrespective of the given user program, RAZZER carries out virtual machine introspection (VMI) to determine the kernel thread context of the guest kernel. Specifically, RAZZER's hypervisor retrieves the kernel thread id assigned by the guest kernel using VMI. For example, the current version of Razzer running on Linux first retrieves the thread_info structure, as it is always located at the top of the kernel stack. Razzer then obtains the kernel thread id stored in task_struct by following the reference to task_struct stored in thread_info. Therefore, Razzer's hypervisor can determine whether or not the breakpoint is fired by the destined kernel thread.

2) Resume Per-Core Execution: After two guest kernel threads are stopped at their respective breakpoint address (i.e., RacePair$_{cand}$), Razzer resumes the execution of both vCPUs such that both threads execute RacePair$_{cand}$ concurrently. One important decision for Razzer to make here is: *which kernel thread should be resumed first?* This is important for identifying data races because some race bugs are only exhibited on a specific execution order. For example, as shown in CVE-2017-2636 (§II-A), a race occurs only if kernel thread B proceeds first after stopping at RacePair$_{cand}$. For this reason, our hypervisor provides an interface, hcall_set_order(), to control the execution order—i.e., a specified vCPU ID in hcall_set_order() is executed first, followed by the other.

The workflow of an execution resume is shown in Figure 4. First, it is assumed that two kernel threads are stopped at their respective breakpoints (i.e., ❶ and ❷), and hcall_set_order() (vCPU0) has been invoked. To resume, Razzer picks vCPU as specified by hcall_set_order() (i.e., vCPU0 in Figure 4) and conducts a single-step on that vCPU immediately to stop after executing a single instruction (❸). This single-step ensures that vCPU0 proceeds before vCPU1, as commanded by hcall_set_order(). Lastly, Razzer resumes the execution of both vCPU0 and vCPU1 (❹).

3) Check Race Results: Our hypervisor checks whether a given RacePair$_{cand}$ actually results in a race (which we call a true race) when both breakpoints are hit concurrently. More specifically, when both memory instructions in RacePair$_{cand}$ hit breakpoints, our hypervisor conducts an introspection of the destined addresses to be accessed by these instructions. If these addresses are identical, Razzer then concludes that a given RacePair$_{cand}$ truly races, promoting such a pair to RacePair$_{true}$. More technically, our hypervisor computes the destined address value by disassembling the instruction at each RacePair$_{cand}$ location and obtaining the concrete register values stored in each vCPU. For example, as illustrated in Ⓐ (Figure 4), Razzer determines whether a given RacePair$_{cand}$ truly races considering that both threads access the same memory location, 0xffff8801e704c020 (i.e., %rdi + 0x20 == %rbx + 0x20).

C. Two Phased Fuzzing to Discover Races

Here, we describe how Razzer discovers race bugs through fuzzing. Razzer's fuzzing is performed in two phases (as shown in Figure 3). (i) the single-thread fuzzing phase (§III-C1) finds a single-thread user program that triggers any RacePair$_{cand}$; and (ii) the multi-thread fuzzing phase (§III-C2) finally finds a multi-thread user program that triggers a harm race based on the result of the single-thread phase. Each fuzzing phase consists of two components, the generator and the executor, where the generator creates a user program and the executor then runs the program.

1) Single-Thread Fuzzing: In this phase, the single-thread generator initially generates P$_{st}$, a single-thread program with a sequence of random syscalls. Next, the single-thread executor runs each P$_{st}$, while testing whether each execution of P$_{st}$ covers any RacePair$_{cand}$ (which is generated by running a static analysis as described in §III-A). If covered, the single-thread executor passes P$_{st}$, which is annotated with the information on covered RacePair$_{cand}$, to the next phase, multi-thread fuzzing. Below, we describe the details of the single-thread generator and executor in turn.

Single-Thread Generator. The single-thread generator constructs a single-threaded user-land program (which we refer to as P$_{st}$), performing a sequence of random system calls to test the kernel's behavior. Razzer constructs P$_{st}$ with the following two strategies: generation and mutation. When using the generation strategy, Razzer randomly generates P$_{st}$ following the pre-defined system call grammar. This system call grammar includes all available system calls as well as a range of reasonable parameter values for each syscall. Following this grammar, Razzer attempts to construct a reasonable user program by randomly selecting a sequence of system calls. It then randomly populates each system call's parameters, and its return value is randomly piggy-backed onto the parameters of a following the syscall as well. As we describe in more detail in §IV, Razzer utilizes pre-defined system call grammar in Syzkaller [42].

As opposed to generation, mutation randomly mutates the existing P$_{st}$. It may randomly drop some syscalls in P$_{st}$, insert new syscalls, or change certain parameter value.

Single-Thread Executor. Given P$_{st}$ from the generator, the single-thread executor runs each P$_{st}$ while performing the following two tasks. First, if an execution of P$_{st}$ covers two memory access instructions in any RacePair$_{cand}$, Razzer annotates such matched RacePair$_{cand}$ information to P$_{st}$. Razzer then passes this annotated P$_{st}$ to the multi-thread generator so that it can be further checked as to whether it is racing.

More specifically, while running P$_{st}$, Razzer monitors the execution coverage per syscall, leveraging the underlying kernel's support to collect the execution coverage, as this capability is a general feature in modern kernels (e.g., the current prototype of Razzer for Linux relies on KCov [3]). After running P$_{st}$, Razzer checks if its execution coverage matches any RacePair$_{cand}$—i.e., one syscall in P$_{st}$ causes the kernel to execute one instruction of RacePair$_{cand}$, and the other syscall executes the other instruction of RacePair$_{cand}$. If matched, Razzer annotates the detailed information of the matched RacePair$_{cand}$ to P$_{st}$ such that Razzer can test whether RacePair$_{cand}$ can be deterministically triggered in the subsequent fuzzing phases. This annotated information includes the following: (i) two racy syscalls, each of which executes RacePair$_{cand}$; and (ii) the addresses of RacePair$_{cand}$.

Note that there can be multiple RacePair$_{cand}$ matched from a single P$_{st}$. Based on our experience with Razzer, running a single P_{st} matches up to 130 unique RacePairs$_{cand}$. In this case, RAZZER annotates each RacePair$_{cand}$ to a cloned individual copy of P_{st}.

Second, if a running result of P_{st} yields new coverage not executed before by any other P_{st}, RAZZER saves this P_{st} and feeds it back to the single-thread generator again. In fact, this mechanism is largely similar to fuzz testing techniques (known as maintaining a fuzzing corpus). A fuzzing corpus informally represents a minimal set of inputs, which may cover all previously explored basic blocks if all corpus inputs were executed.

Example: CVE-2017-2636 with Single-Thread Fuzz. As illustrated in Figure 3, suppose a single-thread generator generated P_{st} with the following three syscalls in order, `int fd = open(...)`, `ioctl(fd, TCLFLSH)`, and `write(fd, ...)`, while there are other syscalls as well in the middle of these three. If this P_{st} is executed by the single-thread executor, it identifies that a certain RacePair$_{cand}$ has been executed by the program—i.e., the pair(n_hdlc.c:216, n_hdlc.c:440) where the first is executed by `n_hdlc.c:216` and the second is executed by `ioctl(fd, TCLFLSH)`. In such a case all of this matched information is annotated to P_{st} and passed to the multi-thread fuzzing phase.

2) Multi-Thread Fuzzing: After the single-thread fuzzing phase, RAZZER moves on to the multi-thread fuzzing phase. For each RacePair$_{cand}$, the multi-thread generator transforms P_{st} into P_{mt}, a multi-thread version of P_{st}. P_{mt} is also instrumented with hypervisor calls to trigger a race deterministically at the given RacePair$_{cand}$. Lastly, multi-thread executor runs each P_{mt}. If the P_{mt} is confirmed to trigger a race by the hypervisor, RAZZER promotes the corresponding RacePair$_{cand}$ to RacePair$_{true}$, and continues to mutate P_{mt} by feeding it back to the generator. Furthermore, if P_{mt} crashes the kernel, RAZZER produces a detailed report of an identified harmful race.

Multi-Thread Generator. The multi-thread generator takes an annotated P_{st} (which includes RacePair$_{cand}$) as input. Then it outputs P_{mt}, a multi-thread version of P_{st} while leveraging the annotated RacePair$_{cand}$ information to trigger the race deterministically with hypercalls.

Figure 5 illustrates a simplified pseudo-code of this transformation process. It takes the following arguments as input: P_{st}, the program to be transformed; `i` and `j`, each is an index of racing syscalls within P_{st} ($i \leq j$); and `RP_i` and `RP_j`, each is an address of a corresponding RacePair$_{cand}$ instruction. For simplicity, it is assumed that all of the annotated information (from the single executor) is provided as an input argument. This algorithm initially constructs two different execution threads, `thr0` and `thr1`, where each execution is pinned to an individual virtual CPU (lines 8 and 9). RAZZER leverages the kernel's existing feature to pin threads (i.e., the `sched_setaffinity` syscall in Linux), which enables flexible thread controls from the user-space.

```
 1 def Convert_Pst_to_Pmt(Pst, i, j, RP_i, RP_j):
 2     # @Pst: A singled threaded program (annotated)
 3     # @i, @j: an index of racing syscalls within Pst
 4     # @RP_i, @RP_j: an address of a corresponding racepair
 5     # instruction (to syscalls[i] and syscalls[j], respectively)
 6
 7     # Get pinned threads, thr0 and thr1
 8     thr0 = get_pinned_thread(vCPU0)
 9     thr1 = get_pinned_thread(vCPU1)
10
11     # Assign syscalls to thr0 and thr1
12     syscalls = get_syscalls(Pst)
13     thr0.add_syscalls(syscalls[:i])
14     thr1.add_syscalls(syscalls[i+1:j])
15
16     # Determine the execution order
17     r = random([vCPU0, vCPU1])
18     thr0.add_hypercall(hcall_order(r))
19
20     # Trigger and check races
21     thr0.add_hypercall(hcall_set_bp(vCPU0, RP_i))
22     thr0.add_syscalls(syscalls[i])
23     thr0.add_hypercall(hcall_check-race())
24
25     thr1.add_hypercall(hcall_set_bp(vCPU1, RP_j))
26     thr1.add_syscalls(syscalls[j])
27     thr1.add_hypercall(hcall_check_race())
28
29     # Post-race behaviors
30     thr0.add_syscalls(gen_random_syscalls())
31     thr1.add_syscalls(gen_random_syscalls())
32
33     Pmt = Construct_Pmt(thr0, thr1)
34     return Pmt
```

Fig. 5: RAZZER's multi-thread generator algorithm

It then extracts all syscalls from P_{st} (line 12) and subsequently splits these syscalls into two different threads, `thr0` and `thr1`. At this point, we do not add racing syscalls yet—`thr0` contains syscalls before i-th syscalls (line 13) and `thr1` contains syscalls from (i+1)-th syscalls to (j-1)-th syscalls (line 14). To guide the execution order of RacePair$_{cand}$, RAZZER inserts `hcall_set_order()` while randomly selecting the preceding vCPU (lines 17 and 18). Next, in order to trigger the race deterministically at RacePair$_{cand}$, RAZZER resorts to hypervisor's per-core breakpoint functionality by instrumenting `hcall_set_bp()` (lines 21 and 25) immediately before inserting racy syscalls (lines 22 and 26). RAZZER then instruments `hcall_check_race()` further to check precisely if RacePair$_{cand}$ truly causes a race condition (lines 23 and 27). Lastly, in order to induce harmful post-race behaviors further (as described in §II), RAZZER adds randomly generated syscalls (lines 30 and 31).

Multi-Thread Executor. The primary role of the multi-thread executor is to run P_{mt} finally to test if RacePair$_{cand}$ truly triggers a race. While running at runtime, it leverages hypercalls to setup a per-core breakpoint in the RacePair$_{cand}$ instructions before invoking a corresponding racy syscall. `hcall_check_race()` then determines if a race is truly triggered by inspecting the following two conditions: (1) if two breakpoints are indeed captured by the hypervisor, and (2) the concrete memory addresses accessed by the RacePair$_{cand}$ instructions are identical.

Because causing a true race itself does not necessarily imply a harmful race (§II), RAZZER also invokes post-race syscalls in an effort to discern harmful races from true races. Most modern kernels employ runtime race detection mechanisms to check whether a harmful race occurs. For example, the Linux kernel employs various dynamic techniques to detect harmful races. Examples are `lockdep` [29], `KASAN` [2], or assertions manually inserted by kernel developers. We enabled all of these techniques while building the kernel binary such that RAZZER can leverage this enhanced race detection capability. If a violation is detected, RAZZER generates a detailed report about the harmful race.

An important feature of RAZZER is that it provides feedback to the multi-thread generator on P_{mt} causing a true race (even when it is a benign race) such that P_{mt} can be mutated further, but only for the part related to post-race behaviors.

Example: CVE-2017-2636 with Multi-Thread Fuzz. As shown in Figure 3, once receiving P_{st}, the multi-thread generator transforms P_{st} into P_{mt} according to the annotation information of the matched $\texttt{RacePair}_{cand}$. Following the transformation algorithm (Figure 5), the generator places `int fd = open(...)` and `ioctl(fd, TCLFLSH)` in user thread A and `write(fd, ...)` in user thread B. It also instruments hypercalls accordingly—i.e., a `hcall_set_order()` is inserted into thread A (this example randomly selected vCPU0 as a parameter) and `hcall_set_bp()` is inserted before the racy syscalls. After the racy syscalls, `hcall_check_race()` is inserted to check if it causes a true race. If so, RAZZER promotes $\texttt{RacePair}_{cand}$ to $\texttt{RacePair}_{true}$, and P_{mt} is pushed back to the generator such that it can be mutated further. If any future mutation by the generator inserts `ioctl(fd, TCXONC)` and `close(fd)`, the executor will observe the crash and then generate a race report (shown in Figure 16 in the appendix). The race report shows the user program which triggers the race (line 1 to 10 in Figure 16), followed by detailed $\texttt{RacePair}_{true}$ information (line 12 to 20). After that, the report also includes the crash report generated by the kernel (lines 24 to 33). It is worth noting that the race is the root cause of this kernel's crash report.

IV. IMPLEMENTATION

We implemented RAZZER's static analysis based on LLVM 4.0.0 and SVF [39], which provides the points-to analysis framework. Because SVF is implemented for user-land programs, we modified SVF to handle the kernel's memory allocation and free functions in the kernel. Moreover, as SVF ignores all accesses to non-pointer variables even if a non-pointer variable resides in the heap or in global structures, we modified SVF to handle a non-pointer variable if the variable exists in the heap area or in global structures. RAZZER's static analysis generates $\texttt{RacePair}_{cand}$ as a set of two instructions in a source code, each of which is represented with a source filename and a line number within a file. $\texttt{RacePair}_{cand}$ is translated into a machine address during building the kernel using debugging information generated by `GCC`. RAZZER's hypervisor is implemented on QEMU 2.5.0 and utilizes KVM (Kernel-based Virtual Machine) to take the advantage of hardware acceleration. When breakpoints are hit, we use `Capstone` [1] to disassemble the instruction to check $\texttt{RacePair}_{cand}$. RAZZER's fuzzer is implemented based on `Syzkaller` [42], a kernel fuzzer developed by Google.

To summarize the implementation complexity, RAZZER modified the existing framework as follows: SVF [39] to implement its static analysis (§III-A) with 638 LoC in `C++`, QEMU [5] to implement its hypervisor (§III-B) with 652 LoC in `C`, and Syzkaller [42] to implement its fuzzer (§III-C) with 6,403 LoC in `Go` and 286 LoC in `C++`.

V. EVALUATION

In this section, we evaluate various aspects of RAZZER. First, we list the newly discovered harmful races found by RAZZER (§V-A), and then we evaluate the effectiveness of RAZZER's static analysis (§V-B). Next, we measure the performance overhead of our hypervisor (§V-C) and conduct a comparison study of the fuzzing efficiency with state-of-the-art tools (§V-D).

Experimental Setup. All of our evaluations were performed on an Intel(R) Xeon(R) CPU E5-4655 v4 @ 2.50GHz (30MB cache) with 512GB of RAM. We ran Ubuntu 16.04 with Linux 4.15.12 64-bit as the host kernel. To run RAZZER, we created 32 VMs using our modified KVM/QEMU, and allocated 16 VMs for single-thread and 16 VMs for multi-thread fuzzing. To run `Syzkaller` for comparison, we created 32 VMs using the stock KVM/QEMU so as to utilize the same computing power used when RAZZER run.

Target Kernel Preparation. RAZZER requires no manual modification of the target kernel to be analyzed. For each kernel version, it builds the kernel in two phases. First it builds using LLVM compiler-suites to generate Bitcode objects to perform the static analysis. Then, it builds the kernel using GCC, which will be running as a virtual machine on RAZZER's hypervisor.

A. Newly Found Race Bugs

To demonstrate RAZZER's ability to find race bugs, we ran RAZZER on various versions of the Linux kernel, from `v4.16-rc3` (released on Feb 25, 2018) to `v4.18-rc3` (released on July 1, 2018). We ran RAZZER for approximately 7 weeks using the machine described in the experimental setup section.

Figure 6 summarizes the races identified by RAZZER. In total, RAZZER found 30 harmful races in the kernel. After reporting these, 16 were confirmed and patches of 14 were accordingly submitted by the kernel developers. After the patches were proposed, 13 races were merged into the various versions of affected kernel versions (including the mainline as well). We would like to highlight that the Linux kernel has been extensively fuzzed by many different engineers and researchers and thus that it is not easy to find new bugs with moderate computing power. For instance, the `Syzkaller` team at Google runs their fuzzer on their massive cloud infrastructure in a 24/7 manner to detect bugs early in the Linux kernel. Nevertheless, RAZZER found 30 races, demonstrating its strong effectiveness with regard to finding race bugs.

Efficiency in Discovering New Harmful Races. To clarify how long RAZZER takes to discover such races, Figure 7 illustrates the number of unique crash types found by RAZZER over time while running v4.17-rc1. In this figure, we plotted

Kernel crash summary	Crash type	Kernel version	Kernel subsystem	Confirmed	Patch submitted	Fixed
KASAN: slab-out-of-bounds write in tty_insert_flip_string_flag	Use-After-Free	v4.8	drivers/tty/	✓	✓	✓
WARNING in __static_key_slow_dec	Reachable Warning	v4.8	net/	✓		
Kernel BUG at net/packet/af_packet.c:LINE!	Reachable Assertion	v4.16-rc3	net/packet/	✓	✓	✓
WARNING in refcount_dec	Reachable Warning	v4.16-rc3	net/packet/	✓	✓	✓
unable to handle kernel paging request in snd_seq_oss_readq_puts	Page Fault	v4.16	sound/core/seq/oss/	✓	✓	✓
KASAN: use-after-free Read in loopback_active_get	Use-After-Free	v4.16	sound/drivers/	✓	✓	✓
KASAN: null-ptr-deref Read in rds_ib_get_mr	Null ptr deref	v4.17-rc1	net/rdma/	✓(assisted Syzkaller)	✓	✓
KASAN: null-ptr-deref Read in list_lru_del	Null ptr deref	v4.17-rc1	fs/			
BUG: unable to handle kernel NULL ptr dereference in corrupted	Null ptr deref	v4.17-rc1	net/sctp/			
KASAN: use-after-free Read in nd_jump_root	Use-After-Free	v4.17-rc1	fs/	✓	✓	✓
KASAN: use-after-free Read in link_path_walk	Use-After-Free	v4.17-rc1	fs/	✓	✓	✓
BUG: unable to handle kernel paging request in __inet_check_established	Page Fault	v4.17-rc1	net/ipv4/			
KASAN: null-ptr-deref Read in ata_pio_sector	Null ptr deref	v4.17-rc1	net/drivers/ata/			
WARNING in ip_recv_error	Reachable Warning	v4.17-rc1	net/	✓	✓	✓
WARNING in remove_proc_entry	Reachable Warning	v4.17-rc1	net/sunrpc/			
KASAN: null-ptr-deref Read in ip6gre_exit_batch_net	Null ptr deref	v4.17-rc1	net/ipv6/			
KASAN: slab-out-of-bounds Write in __register_sysctl_table	Heap overflow	v4.17-rc1	net/ipv6/			
KASAN: use-after-free Write in skb_release_data	Use-After-Free	v4.17-rc1	net/core/			
KASAN: invalid-free in ptlock_free	Double free	v4.17-rc1	mm/			
Kernel BUG at lib/list_debug.c:LINE!	Reachable Assertion	v4.17-rc1	drivers/infiniband/			
INFO: trying to register non-static key in __handle_mm_fault	Reachable INFO	v4.17-rc1	mm/			
KASAN: use-after-free Read in vhost-chr_write_iter	Use-After-Free	v4.17-rc1	drivers/vhost/	✓	✓	✓
BUG: soft lockup in vmemdup_user	Soft lockup	v4.17-rc1	net/			
KASAN: use-after-free Read in rds_tcp_accept_one	Use-After-Free	v4.17-rc1	net/rds/			
WARNING in sg_rq_end_io	Reachable Warning	v4.17-rc1	drivers/scsi/			
BUG: soft lockup in snd_virmidi_output_trigger	Soft lockup	v4.18-rc3	sound/core/seq/	✓(assisted Syzkaller)	✓	✓
KASAN: null-ptr-deref Read in smc_ioctl	Null ptr deref	v4.18-rc3	net/smc/	✓	✓	✓
KASAN: null-ptr-deref Write in binderf_update_page_range	Null ptr deref	v4.18-rc3	drivers/android/	✓	✓	
WARNING in port_delete	Reachable Warning	v4.18-rc3	sound/core/seq/	✓(assisted Syzkaller)		
KASAN: null-ptr-deref in inode_permission	Null ptr def	v4.18-rc3	fs/	✓	✓	✓

Fig. 6: List of harmful race bugs newly discovered by RAZZER

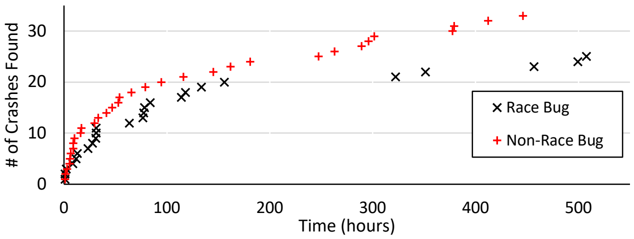
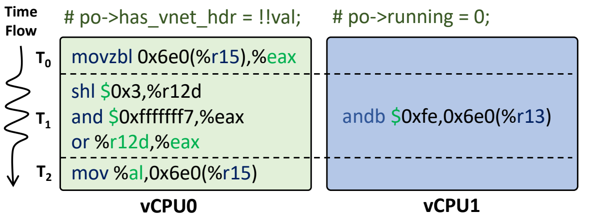


Fig. 7: Number of unique crashes over time (v4.17-rc1).



```
# po->has_vnet_hdr = !!val;
movzbl 0x6e0(%r15),%eax
shl $0x3,%r12d
and $0xfffffff7,%eax
or %r12d,%eax
mov %al,0x6e0(%r15)

# po->running = 0;
andb $0xfe,0x6e0(%r13)
```

Fig. 8: A thread interleaving causing the race of WARNING in refcount_dec (v4.16-rc3)

two classes of bugs: (i) non-race bugs, which are discovered by single-thread fuzzing; and (ii) race bugs, which are discovered by multi-thread fuzzing. While RAZZER's general goal is to find race bugs, its single-thread fuzzer also finds non-race bugs as well (as it is randomly fuzzing the kernel), and all the crashes found by the single-thread fuzzer are highly likely to be non-race bugs [2]. In general, both race bugs and non-race bugs show the similar patterns—most bugs are found in the early stage of fuzzing. Particularly focusing on race bugs, 70% were found within the first 100 hours, whereas the following 400 hours only 30% were found.

Reliability and Security Impacts. All of these harmful races found by RAZZER have severe reliability and security impacts. From the reliability perspective, races are detrimental. They cause unpredictable, non-deterministic crashes of the kernel, which are still challenging to reproduce due to the nature of races, critically damaging the reliability of the entire system. From the security perspective, some of these races can be abused by an attacker who launches a privilege escalation attack (i.e., acquiring root privileges from a non-root execution) as opposed to a denial-of-service attack. In particular, if

[2] It is still possible to observe race bugs from the single-thread user program (i.e., races during interrupt handling in the kernel), but we do not consider these cases in this paper and leave it as future work.

buffer overflows or use-after-free allow a write operation (i.e., KASAN: slab-out-of-bounds write in tty_insert_flip_string_flag, KASAN: slab-out-of-bounds Write in __register_sysctl_table, and KASAN: use-after-free Write in skb_release_data), a user program may overwrite a credential structure in the kernel to escalate its privilege.

Very Old Races in the Kernel. Surprisingly, we noted several harmful races found by RAZZER had existed in the kernel for a very long time, implying that RAZZER is capable of uncovering harmful races that tools such as the kernel fuzzers like `Syzkaller`, the random thread interleaving tool `SKI`, and many other dynamic race detectors cannot find. More specifically, based on our bug report, the kernel developer found that the `WARNING in refcount_dec` issue was present in the kernel since 2007 (Linux v2.6.20), and `KASAN: slab-out-of-bounds write in tty_insert_flip_string_fixed_flag` was present since 2011 (Linux v2.6.38).

To consider about why some race bugs have long remained undetected, we conducted an in-depth study of `WARNING in refcount_dec`. Based on our analysis, we believe it is related to the fact that the chance to observe this race is extremely low in practice. As shown in Figure 8, `RacePair`$_{\text{true}}$ is in this case `mov %al,0x6e0(%r15)` and `andb $0xfe,0x6e0(%r13)`. In order to manifest this race, `vCPU1` should execute `andb $0xfe,0x6e0(%r13)` while `vCPU0` is executing only three non-memory access instructions in T_1, showing a very small time window to trigger the race. Considering the numerous of instructions that each syscall will be executing, this is an extremely rare case, signifying the effectiveness of RAZZER in enforcing deterministic thread interleaving towards potential race spots.

Root Cause Information. One notable feature of RAZZER is that it produces a detailed report on an identified harmful race, which significantly helps to discern its root cause (see Figure 16 and Figure 17 in the appendix). Based on this root cause information, particularly with regard to the two racy syscalls and confirmed `RacePair`$_{\text{harm}}$ locations, developers were able to fix races promptly once we reported these cases. For example, it only took 20 minutes and 2 hours, respectively, for developers to propose the patches for `KASAN: use-after-free in loopback_active_get` and `Unable to handle kernel paging request in snd_seq_oss_readq_puts`[3].

Another interesting example is `KASAN: null-ptr-deref Read in rds_ib_get_mr`. This race was in fact reported by `Syzkaller` 9 days earlier than it was by RAZZER, but the kernel developers did not take any action. We assume that this is due to the difficulty of determining the root cause of this race based on a `Syzkaller`'s report, as it is more suitable for understanding memory corruption issues along with KASAN's report. However, once we sent RAZZER's detailed report, this race was confirmed in one day by the respective kernel developer, implying that our report helped the developer to understand the race.

Kernel module	**Size (LLVM Bitcode)**	**Analysis Time**
cert	148 KB	0.2 sec
init	728 KB	1 sec
security	1.7 MB	3 sec
lib	3.5 MB	36 sec
crypto	3.3 MB	42 sec
arch	11 MB	2 min
block	6 MB	3 min
ipc	5 MB	3 min
mm	15 MB	5 min
fs	17 MB	8 min
sound	20 MB	22 min
kernel	29 MB	24 min
net	47 MB	72 min
drivers	68 MB	134 min

Fig. 9: Performance of RAZZER's static analysis (v4.17-rc1)

CVE ID	**First Instruction**	**Second Instruction**
CVE-2017-2636	drivers/tty/n_hdlc.c:216	drivers/tty/n_hdlc.c:440
CVE-2016-8655	net/packet/af_packet.c:3660	net/packet/af_packet.c:4229
CVE-2017-17712	net/ipv4/raw.c:640	net/ipv4/ip_sockglue.c:748

Fig. 10: `RacePair`$_{\text{true}}$ required to find known race bugs (Linux `v4.8`). RAZZER's static analysis results, `RacePairs`$_{\text{cand}}$, included all of these pairs.

We note that race bugs are generally known to be difficult bugs with regard to understanding their root causes; hence, even when they are easily reproducible, it is still not trivial for developers to understand a race and generate a patch. However, as suggested by our experiences with kernel developers, RAZZER not only finds harmful races but also assists with easy, low-cost patching processes.

B. Effectiveness of Static Analysis

Performance Overhead of Partitioned Analysis. To demonstrate the effectiveness of RAZZER's partitioned analysis, we measured the time taken to obtain all `RacePairs`$_{\text{cand}}$ from the entire kernel. As shown in Figure 9, RAZZER showed moderate performance overhead per module—ranging from 2 minutes to 134 minutes, depending on the size of the target LLVM bitcode. We believe this is fast enough to ensure a proper analysis of the kernel as the kernel is updated. It is worth noting that, the consolidated entire kernel binary (i.e., `vmlinux.bc`) is a 756 MB LLVM Bitcode file and our analysis of this binary did not terminate when we ran it for 7 days.

Correctness of `RacePairs`$_{\text{cand}}$. Because our static analysis prunes out a numerous number of non-racy memory pairs, it may show false negatives (i.e., some `RacePair`$_{\text{true}}$ is not included in the obtained `RacePairs`$_{\text{cand}}$). As such, to demonstrate the correctness of `RacePairs`$_{\text{cand}}$ results, we checked if the obtained `RacePairs`$_{\text{cand}}$ include `RacePair`$_{\text{true}}$ for the three previously known harmful races in Linux v4.8 shown in Figure 10. After checking this, we confirmed that the `RacePairs`$_{\text{cand}}$ from our static analysis indeed includes all

[3] As this taken time simply measures each developer's response time to our report, we assume that the actual time required to develop a patch should be much shorter than this.

Kernel version	# of Paired Memory Access Instr.	# of RacePairs$_{\text{cand}}$
v4.8	578 M	0.3 M (0.05%)
v4.16-rc3	8,509 M	3.4 M (0.04%)
v4.17-rc1	4,025 M	1.5 M (0.05%)

Fig. 11: The total number of race candidate pairs generated by Razzer's static analysis. We also present the number of paired memory access instructions to be referred as the upper-bound number of the analysis.

three RacePair$_{\text{true}}$ cases. Although this represents a limited study to show the correctness of Razzer's static analysis, we emphasize here that Razzer's static analysis mechanism would be effective enough to discover races similar to those three CVE races. Moreover, we believe that the 30 harmful races newly found by Razzer also support the correctness of the RacePairs$_{\text{cand}}$ results, especially from practical aspects.

Effectiveness of RacePair$_{\text{cand}}$. Figure 11 shows the total number of RacePairs$_{\text{cand}}$ obtained by Razzer from the entire kernel through its static analysis. For a clearer understanding of this number, we also show the total number of paired memory access instructions, approximating the upper bound number of RacePairs$_{\text{cand}}$ results, in the kernel which is partitioned in a way that Razzer does.

For all kernel versions, Razzer produced 0.05%, 0.04%, and 0.05% of RacePairs$_{\text{cand}}$ compared to the paired number of memory access instructions. This suggests that Razzer's static analysis effectively guides its fuzzer to focus only on less than 0.1% of potential racy spots, effectively avoiding an enormous number of non-racy spots to be explored.

C. Hypervisor Overhead

Given that Razzer utilizes hypercalls to enable the deterministic behavior of vCPUs, it requires extra overhead to communicate with the hypervisor. To understand how much overhead is incurred due to the hypervisor, we measured the elapsed time of each hypercall 100 M times and computed the average. We used user-space `clock_gettime()` as a timer function for these measurements. For a simple comparison of Razzer's hypercall overhead, we implemented a no-op hypercall (i.e., `hcall_nop()`) as a baseline hypercall. `hcall_nop()` does nothing and immediately returns from the hypervisor once invoked.

Figure 12 shows the performance overhead of Razzer's hypercalls. As shown in the figure, our hypercalls incur overhead ranging from 4.69µs to 5.64µs. This implies that to run each P$_{\text{mt}}$ in the multi-thread executor, Razzer's hypercall based mechanism would requires about 14.92µs (i.e., 5.46 + 4.77 + 4.69), which would not be significant overhead. In particular, while `hcall_set_order()` and `hcall_check_race()` only incurs 8% and 9% of the overhead compared to `hcall_nop()`, `hcall_set_bp()` incurs 25%, as `hcall_set_bp()` must install a hardware breakpoint within the corresponding vCPU, which requires extra switching steps in our underlying hypervisor designs—i.e., our base hypervisor, KVM, must switch to the host kernel to access vCPU registers.

`hcall_nop()`	`hcall_set_bp()`	`hcall_check_race()`	`hcall_set_order()`
4.34µs	5.46µs	4.77µs	4.69µs

Fig. 12: Performance overhead when performing Razzer's hypercalls

	Syzkaller [42]	Razzer		
		Single	**Multi**	**Avg.**
Throughput	144 K	151 K	86 K	118 K

Fig. 13: Fuzzing throughput of Syzkaller and Razzer. We present the number of execution per machine for one hour. Single/Multi denotes the throughput on machines running single-thread/multi-thread fuzzing (v4.17-rc1).

D. Comparison Study of the Fuzzing Efficiency

1) Finding Offending User Programs: This subsection demonstrates how well Razzer finds a user program triggering a race (i.e., how well Razzer meets R1 in §II-B). We measured two different aspects: (1) the fuzzing throughput, showing how many input program instances that Razzer can execute for a certain period of time; and (2) the number of executions required to find a user program triggering a previously known harmful race. In these measurements, we also compare Razzer's performance with that of Syzkaller to clarify how well Razzer meets R1 compared to the state-of-the-art kernel fuzzing tool. In summary, the fuzzing throughput of Razzer is worse than Syzkaller as expected, because Razzer utilizes a hypervisor to enforce deterministic thread interleaving behavior. However, Razzer is much faster than Syzkaller when finding a user program triggering a harmful race (i.e., 23 to 86 times faster at a minimum), as deterministic thread interleaving behavior towards potential racy spots significantly reduces the search space to be fuzzed.

Execution Throughput. We measure the number of user programs executed by Razzer, both in the single-thread (§III-C1) and multi-thread fuzzing phases (§III-C1).

Figure 13 shows the execution throughput result after running both tools for 10 minutes. The number in each cell represents the averaged number per VM for an easy comparison. As expected, Razzer's multi-thread fuzzing shows lower throughput than Razzer's single-thread fuzzing and Syzkaller, mainly due to two extra jobs which arise during the multi-thread fuzzing step. First, it performs extra hypercalls, where the invocation itself imposes extra overhead, and second, if both breakpoints remain unfired, the hypervisor should wait until its own timer expires.

One interesting aspect to note here is that Razzer's single-thread fuzzing shows higher throughput than that of Syzkaller. This occurs because Razzer's single-thread executor is truly single-threaded whereas Syzkaller spawns multiple worker threads to identify data races.

Required Number of Executions to Find a Race. The execution throughput of fuzz testing may be an indirect measure of how efficient the fuzz technique is, but a more direct and important measure should be the time required to find bugs (i.e., a harmful race in this paper). To demonstrate this, we measured the number of executions required to discover previously known harmful races, CVE-2017-2636, CVE-2016-8655, and CVE-2017-17712, while running for 10 hours. To trigger the race within a reasonable time for this experiment, we configured the program generation grammar of both RAZZER and Syzkaller as a limited set of syscalls that is related to the target race bug. For a fair evaluation, we used the same configuration for both tools and provided $\texttt{RacePair}_{\text{cand}}$ generated from the entire kernel to RAZZER (i.e., fewer $\texttt{RacePair}_{\text{cand}}$ leads to a more efficient search for RAZZER).

As shown in Figure 14, RAZZER found all of these previously known races with a reasonable number of execution (i.e., from 246 K to 1,170 K) from as well as within a reasonable amount of time (i.e., from 7 minutes to 26 minutes). However, Syzkaller failed to find all of these cases, although executed from 5 M to 37 M generated/mutated programs over the duration of 10 hours. Particularly based on these CVE cases, these outcomes suggest that RAZZER is faster than Syzkaller, ranging from 23 to 85 times at least.

2) Finding Offending Thread-Interleaving cases: In order to show how well RAZZER finds a thread interleaving for a given program to trigger a race (i.e., how well RAZZER meets R2 in §II-B), we performed the comparison study between RAZZER and SKI.

Experimental Setting. Because we do not have access to the implementation of SKI, we implemented $\texttt{SKI}_{\text{Emu}}$ by extending RAZZER with SKI's random thread interleaving features. The key to implement $\texttt{SKI}_{\text{Emu}}$ lies in implementing its random thread interleaving feature, which is done by modifying RAZZER's multi-thread fuzzing phase. Instead of utilizing RAZZER's hypercalls for the per-core breakpoint at $\texttt{RacePair}_{\text{cand}}$, $\texttt{SKI}_{\text{Emu}}$ performs random thread interleaving, as shown in Figure 2. More specifically, $\texttt{SKI}_{\text{Emu}}$ continues to randomly select one vCPU and then executes it until it meets any memory access instruction. Thus, $\texttt{SKI}_{\text{Emu}}$ interleaves the execution as it faces the memory access instruction, which is identical to what SKI originally proposed.

In this experiment, we assume that RAZZER and $\texttt{SKI}_{\text{Emu}}$ obtained a user program triggering a harmful race so as to focus on evaluating the R2 aspect. This is supported by turning off the input generator during the single-thread fuzzing phase ahd then simply providing a single user program to it. We used three programs, each of which already triggers previously known harmful races. These are CVE-2017-2636 [28], CVE-2016-8655 [26], and CVE-2017-17712 [27].

Number of Executions to Find Races. Figure 15 shows the number of required executions to trigger each race while running RAZZER and $\texttt{SKI}_{\text{Emu}}$. As RAZZER only explores thread interleaving related to $\texttt{RacePairs}_{\text{cand}}$ (spotted by running a given user program) and thus explores far fewer thread interleaving cases, RAZZER requires far fewer executions than $\texttt{SKI}_{\text{Emu}}$, ranging from 30 times to 398 times less. This result also suggests that many thread interleaving cases that are explored by $\texttt{SKI}_{\text{Emu}}$ are not related to races, signifying RAZZER's effectiveness in meeting R2.

VI. RELATED WORK

In this section, we discuss work related to RAZZER, particularly focusing on techniques that can identify (or assist in the identification of) data races.

Dynamic Fuzz Testing. Many recent studies have demonstrated that fuzzing is a promising technique to find bugs in user-land programs [8, 12, 18, 33, 34, 44, 46] and in kernels [13, 19, 23, 24, 42, 43, 45]. The key advantage of fuzzing is not only that this method efficiently finds bugs in target programs but also that it does not suffer from false positives as it generates an input reproducing a bug. However, to the best of our knowledge, all fuzzing techniques are inefficient when used to identify race bugs mainly because their designs are not tailored to races. While most fuzzers focus on leveraging previously explored execution coverage, they do not consider thread interleaving (i.e., traditional fuzzers do not not meet R2 in §II-B). Compared to these, RAZZER considers both the execution coverage and thread interleaving to discover data races more effectively.

Dynamic Thread Scheduler. Several studies [10, 16, 30, 35, 40] have attempted to find instances of race-causing thread interleaving by implementing a customized thread scheduler which randomizes the per-thread execution scheduling. In particular, the PCT Algorithm [10] and SKI [16] discover races in user programs or the kernel through exploring all possible thread interleaving cases. Limitations of these two methods are: (i) they do not consider R1 (§II-B) as they do not generate (or mutate) an input program and thus it cannot find a new program which triggers data race; and (ii) they are not efficient in meeting R2 as they must search the very large spaces of all possible thread interleaving cases. In fact, the design of RAZZER is inspired by the PCT algorithm and SKI—it meets R1 by tailoring the fuzzing process while efficiently meeting R2 by undertaking prioritized searches over $\texttt{RacePairs}_{\text{cand}}$.

Dynamic Race Detector. Many studies [7, 9, 11, 15, 20, 22, 25, 31, 32, 36, 40, 47, 48] have sought to improve the race detection capability at runtime by collecting rich contextual information on races. These are essentially orthogonal to RAZZER—once deployed together with RAZZER, RAZZER's race detection capability can be augmented as well. This orthogonal relationship is similar how traditional fuzzers (focusing on identifying memory corruption bugs) run with extra sanitizers (i.e., AFL [46] and Syzkaller [42] strongly encourage their users to run them with AddressSanitizer [37] or MemorySanitizer [38]).

In particular, ThreadSanitizer [36] is a commodity race detector developed by Google which was recently utilized with the Linux kernel as well. To augment the performance when detecting races, TxRace [47] leverages the hardware transactional memory and ProRace [48] utilizes a performance monitoring unit. Memory sampling techniques [9, 11, 15, 25] selectively monitor memory accesses to optimize the performances. RaceMob [21] crowdsources runtime race tests from

Race Bugs	Syzkaller [42]			Razzer				
	# exec	Time	Found	# exec			Time	Found
				Single	Multi	Total		
CVE-2017-2636	5 M	10 hrs	×	169 K	77 K	246 K	7 mins	✓
CVE-2016-8655	29 M	10 hrs	×	821 K	349 K	1,170 K	26 mins	✓
CVE-2017-17712	37 M	10 hrs	×	565 K	242 K	807 K	18 mins	✓

Fig. 14: Efficiency of `Syzkaller` and Razzer in finding a user program that triggers a race. We measured the total number of executions and the time required to find previously known races. Razzer found all of the known races within a reasonable amount of time, while `Syzkaller` did not find any within the given time of 10 hours (v4.8).

CVE	CVE-2017-2636		CVE-2016-8655		CVE-2017-17712	
	Found	Total	Found	Total	Found	Total
SKI$_{\text{Emu}}$ [16]	2,038	6,435	636	8,008	8,362	27,132
Razzer	8	23	21	43	21	56

Fig. 15: Efficiency of SKI$_{\text{Emu}}$ and Razzer in uncovering thread interleaving that triggers a race. `Found` column shows the required number of executions to find the interleaving (obtained by repeating the experiment 5 times and computing the average), and the `Total` column shows the theoretical maximum number of required executions for each tool (v4.8).

potential data races that are generated by the static analysis. Snorlax [22] suggests a coarse interleaving hypothesis to leverage coarse-grained timing information to determine the thread interleaving of events.

Static Analysis. A static analysis has been used extensively to discover unknown bugs. In this category, we focus our discussion on static analysis works relevant to either race bug detection or points-to analysis implementations. Relay [41] is a static race detector for large programs such as kernels. Relay generates `RacePairs`$_{\text{cand}}$ by performing a lockset-based bottom-up analysis while summarizing each function's behavior. RacerX [14] is also designed to find race conditions and deadlocks for large, complex multi-threaded systems. Due to the limitations of using static analysis techniques alone, these are essentially incurring a high false positive rate (e.g., Relay showed a false positive rate of 84% on the Linux kernel), critically limiting their usage in practice. However, Razzer also leverages dynamic analysis techniques, addressing the possibility of high false positive rates. In terms of points-to analysis implementations, K-miner [17] was recently presented to uncover memory corruption vulnerabilities in commodity operating systems through an inter-procedural and context-sensitive analysis. Razzer's static analysis is built based on the implementation of K-miner's but modified to identify `RacePairs`$_{\text{cand}}$ through a points-to analysis.

VII. Discussion

False Negatives in Static Analysis. Since Razzer relies on the results of the static analysis, if any true race pair is missing from `RacePair`$_{\text{cand}}$, it would lead to false negatives of Razzer. Such missing cases of the static analysis may occur mainly due to the partition analysis. Specifically, the partition analysis of Razzer is based on the assumption that a race across different kernel modules are rarely happen (e.g., a file system and a terminal device driver). However, if that happens, Razzer's `RacePair`$_{\text{cand}}$ would not include such a race pair. In order to completely address this problem, Razzer needs to avoid the partition analysis. Instead, it should perform more precise static analysis techniques, which aggressively identifies must-not-race pairs. For instance, Razzer can leverage synchronization primitives as done in previous work [14, 41].

Applying Razzer for Other Systems. We believe it would not be challenging to apply Razzer for other modern operating systems, such as Windows, MacOSX, FreeBSD, as long as their source code is available. The only place that Razzer leverages Linux-specific domain knowledge is in its system call invocation model, and all the rest designs are platform-agnostic as its core mechanism is performed either offline (i.e., a static analysis) or transparent (i.e., a tailored hypervisor).

To apply Razzer for userland programs, additional mutation strategies after identifying a race may not be necessary. Unlike the Linux kernel which occasionally allows a race to improve performance, a race itself itself is considered as a bug in most of userland programs.

VIII. Conclusion

We proposed Razzer, a fuzz testing tool tailored to find race bugs. It utilizes a static analysis to spot potential data race points to guide the fuzzer to identify races. Moreover, it modifies the underlying hypervisor to trigger a race deterministically. The evaluation of Razzer demonstrates its strong capability to detect races. It has thus far detected 30 new races in the Linux kernel, and a comparison study with other state-of-the-art tools, specifically `Syzkaller` and `SKI`, demonstrates its outstanding efficiency to detect race bugs in the kernel.

IX. Acknowledgment

We would like to thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. We also would like to thank Linux kernel developers for their helpful feedback and responses. This research is supported in part by ERC through NRF of Korea (NRF-2018R1A5A1059921) and Samsung Research Funding & Incubation Center (SRFC-IT1701-05).

References

[1] Capstone, 2018. https://www.capstone-engine.org.

[2] Kernel address sanitizer, 2018. https://github.com/google/kasan/wiki.

[3] Kcov, 2018. http://simonkagstrom.github.io/kcov/index.html.

[4] Linux kernel mailing list archive, 2018. https://lkml.org.

[5] Qemu, 2018. https://www.qemu.org.

[6] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.

[7] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. Sharc: Checking data sharing strategies for multithreaded c. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, June 2008.

[8] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[9] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[10] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, Mar. 2010.

[11] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016.

[12] S. K. Cha, M. Woo, and D. Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[13] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[14] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[15] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[16] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[17] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[18] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[19] H. Han and S. K. Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[20] G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5), 1997.

[21] B. Kasikci, C. Zamfir, and G. Candea. Racemob: crowdsourced data race detection. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[22] B. Kasikci, W. Cui, X. Ge, and B. Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[23] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim. Cab-fuzz: practical concolic testing techniques for cots operating systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[24] M. Labs. Kernelfuzzer, 2016. https://github.com/mwrlabs/KernelFuzzer.

[25] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[26] MITRE. CVE-2016-8655., 2016. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655.

[27] MITRE. CVE-2017-17712., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17712.

[28] MITRE. CVE-2017-2636., 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2636.

[29] I. Molnar. Runtime locking correctness validator, 2018. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[30] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[31] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[32] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Acm Sigplan Notices*, volume 38. ACM, 2003.

[33] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[34] A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[35] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Tucson, Arizona, June 2008.

[36] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[37] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[38] E. Stepanov and K. Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015.

[39] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016.

[40] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[41] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007.

[42] D. Vyukov. Syzkaller, 2015. https://github.com/google/syzkaller.

[43] V. M. Weaver and D. Jones. perf fuzzer: Targeted fuzzing of the perf event open () system call. Technical report, Technical Report UMAINEVMW-TR-PERF-FUZZER, University of Maine, 2015.

[44] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[45] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[46] M. Zalewsk. American fuzzy lop, 2014. http://lcamtuf.coredump.cx/afl.

[47] T. Zhang, D. Lee, and C. Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[48] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

Appendix

```
[*] User program dump
  # ...
  thr0: int fd = open("...") # syscall index 0
  # ...
  thr0: ioctl(fd, TCLFLSH) # syscall index 8
  # ...
  thr1: write(fd, "...") # syscall index 12
  # ...
  thr1: ioctl(fd, TCXONC)
  thr1: close(fd)

[*] RacePair-0: drivers/tty/n_hdlc.c:440
  Syscall index 8 (sys_ioctl)
  BP-0 at ffffffff817f8351
  Write to ffff8801e704c020

[*] RacePair-1: drivers/tty/n_hdlc.c:216
  Syscall index 12 (sys_write)
  BP-1 at ffffffff817f95b9
  Read from ffff8801e704c020

[*] Confirmed as the true race.

# Begin: A crash report from the kernel.
BUG: KASAN: use-after-free in n_hdlc_buf_get+0x41/0x90 ...
  # ...
  # ...
  Call Trace:
     dump_stack+0xb3/0x110
     # ...
     n_hdlc_buf_get+0x41/0x90
     n_hdlc_tty_close+0x1c8/0x2d0
     # ...
```

Fig. 16: A race report produced by RAZZER on CVE-2017-2636

```
[*] User program dump
  # ...
  thr0: sys_setsockopt(PACKET_AUXDATA) # syscall index 3
  # ...
  thr1: sys_setsockopt(PACKET_RX_RING) # syscall index 7
  # ...

[*] RacePair-0: net/packet/af_packet.c:3773
  Syscall index 3 (sys_setsockopt)
  BP-0 at ffffffff834701bb
  Write to ffff8800b0ca9120

[*] RacePair-1: net/packet/af_packet.c:4303
  Syscall index 7 (sys_setsockopt)
  BP-1 at ffffffff8346d480
  Read from ffff8800b0ca9120

[*] Confirmed as the true race.

# Begin: A crash report from the kernel.
refcount_t: decrement hit 0; leaking memory.
WARNING: CPU: 0 PID: 12248 at lib/refcount.c:228
# ...

Call Trace:
 dump_stack+0x155/0x1f6 lib/dump_stack.c:53
 # ...
 # ...
  __sock_put include/net/sock.h:629 [inline]
 __unregister_prot_hook+0x128/0x190 net/packet/af_packet.c:369
```

Fig. 17: A race report produced by RAZZER on a newly discovered race, `WARNING in refcount_dec` (v4.16-rc3).