DDRace: Finding Concurrency UAF Vulnerabilities in Linux Drivers with Directed Fuzzing

Ming Yuan[1], *Bodong Zhao*[1], *Penghui Li*[3], *Jiashuo Liang*[4], *Xinhui Han*[4], *Xiapu Luo*[5], *Chao Zhang*[1,2*]

[1]*Tsinghua University* [2]*Zhongguancun Lab*
[3]*The Chinese University of Hong Kong* [4]*Peking University* [5]*The Hong Kong Polytechnic University*

Abstract

Concurrency use-after-free (UAF) vulnerabilities account for a large portion of UAF vulnerabilities in Linux drivers. Many solutions have been proposed to find either concurrency bugs or UAF vulnerabilities, but few of them can be directly applied to efficiently find concurrency UAF vulnerabilities. In this paper, we propose the *first* concurrency directed greybox fuzzing solution DDRace to discover concurrency UAF vulnerabilities efficiently in Linux drivers. First, we identify *candidate use-after-free locations* as target sites and extract the relevant concurrency elements to reduce the exploration space of directed fuzzing. Second, we design a novel *vulnerability-related distance metric* and an *interleaving priority scheme* to guide the fuzzer to better explore UAF vulnerabilities and thread interleavings. Lastly, to make test cases reproducible, we design an *adaptive kernel state migration* scheme to assist continuous fuzzing. We have implemented a prototype of DDRace, and evaluated it on upstream Linux drivers. Results show that DDRace is effective at discovering concurrency use-after-free vulnerabilities. It finds 4 unknown vulnerabilities and 8 known ones, which is more effective than other state-of-the-art solutions.

1 Introduction

The widely deployed multi-core processors and multi-thread programming have brought many concurrency bugs, such as data race, atomic violation, deadlocks, etc. Some concurrency bugs can further lead to vulnerabilities, such as memory corruptions, information leaks, or privilege escalation (e.g., the DirtyCow vulnerability with ID CVE-2016-5195). In particular, concurrency bugs will change the temporal order of events, and are prone to causing temporal memory safety violations, such as use-after-free (UAF) vulnerabilities. On the other hand, a large portion of UAF vulnerabilities in Linux drivers involve concurrency [4]. Thus, it is crucial to discover concurrency UAF vulnerabilities in Linux drivers.

*Corresponding author: chaoz@tsinghua.edu.cn

Static analysis is the most common way of discovering concurrency bugs. Specifically, they either conduct happen-before [9,12] or lockset analysis [18,45] to identify candidate race pairs, i.e., memory access pairs without happen-before relationships or lock synchronizations. Such race pairs could have different orders at runtime, leading to potential concurrency bugs. However, such solutions often consider only one race pair, but concurrency UAF vulnerabilities generally involve two or more pairs, not to mention that most bugs found by them are not vulnerabilities. Further, a small number of solutions (e.g., DCUAF [4], UFO [27], and ConVul [12]) have extended lockset analysis and happen-before analysis to find concurrency UAF vulnerabilities. However, these static analysis-based solutions all require lots of manual efforts to verify results and remove false positives.

Another type of concurrency bug discovery solution is fuzzing [13,30,31,55]. Such solutions explore not only the input space but also thread-interleaving space during testing, by introducing proactive thread scheduling mechanisms or passive thread-interleaving feedback to the fuzzer, to find data race bugs. However, they only focus on data race bugs rather than concurrency UAF vulnerabilities, and they can hardly distinguish harmful races from benign ones. In addition, the thread-interleaving space is too huge to explore for fuzzers.

To address these limitations, in this paper, we present the *first* concurrency directed grey-box fuzzing solution DDRace for discovering concurrency UAF vulnerabilities in Linux device drivers. Note that, several directed grey-box fuzzing (DGF) solutions have been proposed in the past, which can greatly narrow down the search space of fuzzing and quickly reach target sites. However, they generally only consider the control flow distance [7, 14, 60] or data constraints distance [22] to the target sites during testing, insensitive to the thread interleavings. Test cases with the same data but different thread interleavings will be ignored by such fuzzers. DDRace extends DGF to take thread interleavings into consideration, and solves the three challenges as follows, to efficiently discover concurrency UAF vulnerabilities. DDRace entails overcoming several challenges with new observations

and techniques:

C1: Infinite Exploration Space. Although DGF solutions have smaller search space than general fuzzing solutions, they have not explored the thread-interleaving space, which is necessary for discovering concurrency UAF vulnerabilities. However, it will enlarge the search space of DGF again and reduce its efficiency in finding vulnerabilities. Then, how to further reduce the search space of concurrency DGF? **Our solution:** we narrow down the target sites of directed fuzzing and the number of candidate thread interleavings. Specifically, we analyze execution traces to locate memory free and use sites that access the same object at runtime but may have vulnerable order, and mark them as candidate target sites for directed fuzzing. Further, we extract all driver interfaces (i.e., syscalls) that can reach the target free and use sites, and extract data race pairs (i.e., memory operations accessing a same object) from each pair of these related driver interfaces. Only interleavings of these data race pairs will be explored by DDRace.

C2: Convergence Speed of Concurrency DGF. Existing DGF solutions often use the control flow distance to guide the fuzzer to explore target sites, but are blind to thread interleavings and objects accessed by target sites. Some general data race fuzzing solutions, e.g., KRACE [55], take the order of one race pair as a new metric, but ignore the order of multiple race pairs. To trigger concurrency UAF vulnerabilities, we have to not only trigger target free and use sites, but also make them access the same object with the correct interleaving. Moreover, these race fuzzing solutions perform random thread-interleaving exploration and fall into the trap of ineffective repeated thread interleavings. **Our solution:** we introduce a new vulnerability-related distance metric and a new interleaving priority mechanism. Specifically, in addition to refining traditional control flow distance, we track the order changes of multiple race pairs and values used in target sites to calculate a new distance, which reflects how well the test case satisfies the UAF vulnerability's constraints. Furthermore, we prioritize seed test cases whose thread interleavings are less commonly explored when the fuzzer picks seeds for further mutation.

C3: Reproducibility of Test Cases. Most kernel fuzzers utilize persistent fuzzing, i.e., running test cases one by one without restarting the target kernel under test, to get a high testing throughput. For instance, Syzkaller [53] will only restart virtual machine (VM) instances after a crash is reported or time out. However, persistent fuzzing will continuously affect the internal states of the kernel. So, if a test case that sets the kernel to a desirable state is selected again, it may not set the kernel to the same state. It makes a proof-of-concept (PoC) unable to reproduce. Furthermore, it may have different distances to target sites in different kernel states, making the distance-guided fuzzing strategy unstable. **Our solution:** we adopt an adaptive kernel state migration scheme. Specifically, we utilize the QEMU snapshot feature to take snapshots of the kernel state at the proper time, and replay the seed test case on the recovered snapshot, to ensure test cases are reproducible.

After addressing these challenges, we implement a prototype of DDRace and evaluate it on six widely-used Linux device drivers. The evaluation results show that DDRace is highly effective in discovering concurrency UAF vulnerabilities. DDRace found 12 concurrency UAF vulnerabilities, including 4 previously unknown ones, and 3 have been assigned with CVE IDs for their high security impacts. DDRace also significantly outperformed the state-of-the-art approaches by finding *more* vulnerabilities at a much *faster* speed. Our characterization further confirmed the benefits of the key components in DDRace for finding concurrency UAF vulnerabilities. We will release the source code of DDRace after publication[1].

In summary, we make the following contributions:

- We present the first concurrency directed grey-box fuzzing solution to discover concurrency UAF vulnerabilities in Linux drivers, which augments the conventional DGF with sensitivity to thread interleavings and UAF vulnerabilities.
- We design a new vulnerability-related distance metric and a novel interleaving priority mechanism to assist directed fuzzers in finding concurrency UAF vulnerabilities.
- We present an adaptive kernel state migration mechanism to guarantee the reproducibility of test cases in continuous kernel fuzzing.
- We implemented a prototype of DDRace, and found 12 concurrency UAF vulnerabilities in Linux device drivers, outperforming baseline approaches.

2 Background and Motivation

2.1 Linux Drivers

The Linux kernel interacts with hardware devices through drivers. A Linux device driver needs to implement some specific driver interfaces such as kernel-driver interfaces [4]. The functions implementing the kernel-driver interfaces are referred to as interface functions, and they form the entry points of the drivers. All the functionalities in the driver are initialized from the driver interface functions.

As discussed in [4], driver concurrency is often determined by the concurrent execution of driver interfaces. Therefore, it is crucial to identify which driver interfaces can be concurrently executed when trying to discover concurrency issues.

2.2 Concurrency UAF Vulnerabilities

Use-after-free (UAF) vulnerabilities refer to vulnerabilities that illegally access dangling pointers. Concurrency UAF vulnerabilities are a special kind of UAF, where unintended memory release and access are introduced by multi-threading [13]. Such vulnerabilities are particularly prevalent and severe in

[1] https://github.com/vul337/DDRace

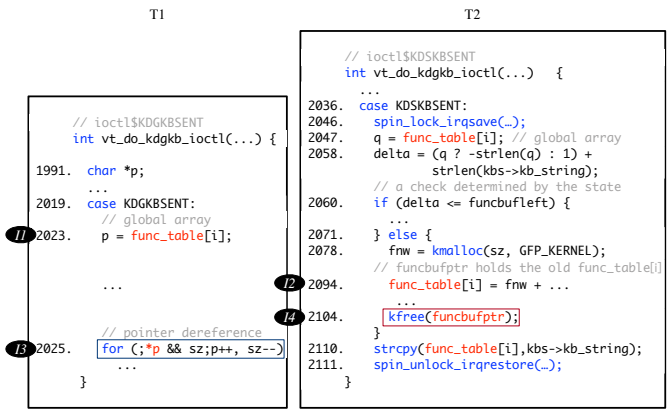

Figure 1: CVE-2020-25656 in Linux 4.19.100.

the Linux kernel. Past studies have shown successful end-to-end exploitation of UAF vulnerabilities [13, 56].

2.3 Grey-box Fuzzing

Directed Grey-box Fuzzing The essence of directed fuzzing is to intensively fuzz towards a set of designated program locations (namely target sites). Existing directed fuzzing techniques design distance metrics that measure how close an input is to target sites [7, 14, 36, 60] and prioritize inputs that are more likely to reach the target sites. Although CAFL [36] points out that the target sites should be reached in order and proposes a constraint-based distance metric, it can be applied to only sequential UAF vulnerabilities rather than concurrency ones. To the best of our knowledge, none of the existing DGF work considers the dimension of thread interleavings in multi-threaded programs, nor does it measure the impact of thread interleavings on reachability.

Thread Scheduling. Thread scheduling is essential in concurrency UAF fuzzing. Based on traditional sequential code coverage feedback, some studies propose to arrange the threads to execute in a specific order [30, 38]. Recent works introduce new concurrency-coverage metrics as feedback and favor inputs that trigger new concurrency coverage, aiming to extensively explore the thread-interleaving exploration space in multi-threaded programs. Notable concurrency-coverage metrics include alias instruction pairs [55], thread-context (e.g., *lock*, *unlock*, and *join* callsites) [13], etc.

Unfortunately, they have several fundamental limitations. They fail to handle scheduling among multiple interesting pairs, making it difficult to get accurate thread-interleaving feedback. They also ignore the memory value changes made by read/write instructions during runtime.

2.4 Motivation Example

A Real-World UAF Vulnerability. Figure 1 shows a concurrency UAF vulnerability (CVE-2020-25656) in *tty* driver of Linux 4.19.100, which is a stream driver managing the connection to the hardware terminal [28]. The example shows two threads (Thread1 and Thread2) that execute system call (i.e., driver interface) `ioctl$KDGKBSENT` and `ioctl$KDSKBSENT`, respectively, and they can be executed concurrently. Specifically, Thread1 fetches a pointer from the global array `func_table` at line 2023 and dereferences it at line 2025. The other thread Thread2 first allocates a new heap memory at line 2078 and stores the corresponding pointer `fnw` to `func_table[i]` at line 2094. Line 2104 then tries to deallocate the obsolete heap memory through the pointer `funcbufptr`. A concurrency UAF vulnerability can happen because Thread1 can potentially use the obsolete memory via `func_table[i]` after Thread2 has freed it.

Challenges to Find the UAF. To trigger this UAF, we first need to enter the `else` branch of the conditional check that guards the vulnerable code in Thread2 at line 2060. Then, only appropriate execution order (thread interleaving) of I1/I2 and I3/I4 (i.e., executing I1 before I2 (I1 $\rightarrow$ I2) and I4 before I3 (I4 $\rightarrow$ I3)) can trigger the vulnerability, which means that we need to execute I2 and I4 within a tiny time window between I1 and I3.

However, discovering such a concurrency UAF vulnerability with fuzzing is extremely difficult. We summarize three inherent challenges. First, since fuzzing is supposed to be automatic and exclude human efforts during fuzzing, a fuzzer needs to re-execute a code snippet thousands of times to explore thread-interleavings and find a proper one that can trigger the vulnerability, as mentioned earlier. However, given the huge code base and high complexity of the Linux kernel, the exploration space is infinite if the fuzzer tries to explore thread interleavings of all code.

Second, even if the fuzzer focuses on only one potential vulnerability and its relevant code snippet, it is challenging for the fuzzer to satisfy all constraints for triggering the vulnerability, including control flow constraints (e.g., conditional statements before the vulnerable code), data flow constraints (e.g., the "FREE" and "USE" operations should operate on the same memory location), and the thread interleavings. The fuzzer needs to acquire guidance from the execution information, instead of fuzzing blindly. For example, to find the UAF in Figure 1, a naive fuzzer without guidance may get stuck with repeated useless thread interleavings and fails to find the useful one. Existing works use basic-block level code coverage and input distance as the guidance, aiming to either increase the code coverage or shorten the input distance during fuzzing. However, they are mostly unaware of the dimension of thread interleavings, and thus not able to comprehensively schedule the treads.

Third, due to the huge overhead of rebooting the kernel frequently, most kernel fuzzers continuously execute test cases in the target kernel. As a result, a general challenge in kernel fuzzing lies in state aging, i.e., the internal state of the kernel accumulates and changes over time. The state aging problem

causes poor reproducibility of seeds, thus hurting the performance of the fuzzer. For example, in Figure 1, reaching the target of deallocation (line 2104) requires the fuzzer to enter the `else` branch with the condition of `delta > funcbufleft`. However, the global variables such as `func_table` change over fuzzing trials. For example, the code at line 2110 can modify the string `func_table[i]`, and as a result, change the value of `delta` in next fuzzing trial. Consequently, a previously successful input that has reached line 2104 would likely fail later due to the state changing in persistent kernel fuzzing.

3 DDRace

3.1 Overview

We propose a new directed fuzzing approach, DDRace, to effectively find concurrency UAF vulnerabilities in Linux drivers. Figure 2 illustrates the overview workflow, consisting of two stages. It first utilizes a lightweight trace analysis to recognize the target sites (i.e., memory access instructions (USE) and deallocation instructions (FREE) related to potential concurrency UAF vulnerabilities). Next, DDRace identifies driver interfaces relevant to the target sites, and further identifies race pairs. In the second stage, DDRace performs novel concurrency directed grey-box fuzzing that explores thread interleavings using new guidance from the instrumented race pairs, to discover concurrency UAF vulnerabilities around the target sites. During the fuzzing process, DDRace ensures that the preserved seeds are consistent with the expected kernel states by a state migration scheme. DDRace entails overcoming the technical challenges with several new techniques. Below we introduce the main components of DDRace.

Fuzzing Scope Identification (§3.2). Given the huge code base and high complexity of the Linux kernel, it is infeasible to thoroughly fuzz the whole kernel, especially exploring the complex thread-interleavings of all code. Moreover, only a small portion of driver interfaces can potentially trigger vulnerabilities. Therefore, DDRace narrows down the fuzzing scope and focuses on the code with a high probability of triggering a UAF. DDRace first obtains the fuzzing target sites (i.e., the UAF pairs) and identifies the relevant driver interfaces that can reach the target sites from the call graph. To further narrow down the exploration space, DDRace performs points-to analysis on identified driver interfaces and extracts target-related race pairs (i.e., pairs of memory operations that access the same object). DDRace explores only the interleavings about the data race pairs in its fuzzing stage.

Directed Scheduling Fuzzing (§3.3) A directed fuzzer for concurrency UAF vulnerabilities needs not only to reach the target sites, but also to meet specific thread scheduling requirements, e.g., the execution order of threads. A naive fuzzer with only basic block feedback is unaware of the thread interleavings. Thus it cannot sufficiently explore the dimension of thread interleaving, making it difficult to find concurrency UAF vulnerabilities. To make a fuzzer aware of thread interleavings, we design new concurrency feedbacks in the process of *seed preservation*, *seed selection*, and *seed mutation*, to guide the DDRace to explore thread interleavings on both control flow and data flow. Furthermore, to make DDRace directed to fuzzing target sites, DDRace dynamically monitors the value of key variables and chooses proper values to meet specific data constraints and trigger vulnerabilities.

Adaptive State Migration (§3.4). Most kernel fuzzers utilize a persistent mode that continuously executes test cases in the kernel to avoid the overhead of frequent kernel restarts. The internal state of the kernel accumulates and changes during fuzzing, i.e., state aging. As a result, even the same test case may behave differently (e.g., achieve different coverage, distance, etc.) as the state changes. However, the state aging problem makes concurrency directed fuzzing ineffective. For example, a previously successful test case might fail to reach the target sites when executed in a different state or time. DDRace overcomes the problem with a state snapshot solution. DDRace saves snapshots for kernel states during fuzzing. When a test case needs to be executed in a specific state, DDRace restores the corresponding snapshot and executes the test case. Ensuring the consistency of the kernel state and a test case is helpful for improving the efficiency of directed kernel fuzzing.

3.2 Fuzzing Scope Identification

3.2.1 Target Site Recognition

DDRace focuses fuzzing on a set of UAF vulnerability targets. The fuzzing target sites can be obtained through static analysis [4] or dynamic analysis [11]. Specifically, in our prototype, we identify the targets with a lightweight dynamic trace analysis as follows. For each driver, we first instrument the memory access instruction and instructions that invoke deallocation functions (e.g., `*free` function such as `kfree`), to monitor potential USE operations and FREE operations respectively. Next, we run a standard kernel fuzzer (e.g., Syzkaller) as a profiler for a period of time. For each test case, we dump the operand values of instrumented instructions and retrieve an execution trace. Finally, we analyze the trace and recognize pairs of USE and FREE operations that have the same memory object as operands, and mark them as potential UAF targets. Besides, we do the following filtering to further narrow down the target scope: (1) exclude the operation of the stack variable, and (2) exclude the explicit dependency of the driver interface on which it resides, such as the argument of one driver interface depending on the return value of another (which means that they usually do not execute concurrently).

Note that this target site recognition step is not the main contribution of this work. We can leverage other orthogonal techniques like DCUAF [4] to help recognize target sites.

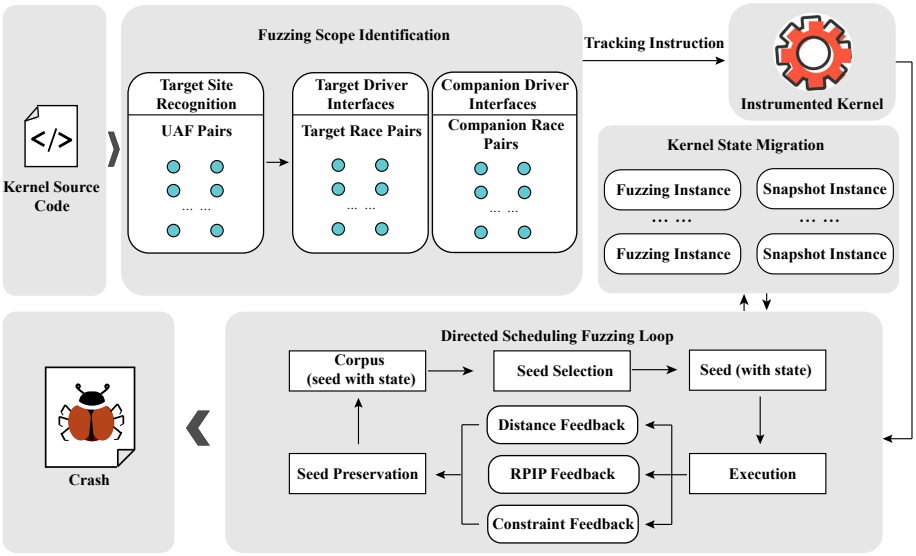

Figure 2: Workflow of DDRace.

3.2.2 Target Race Pair Extraction

After target site recognition, DDRace performs static analysis to identify target driver interfaces and extract target race pairs. DDRace analyzes the source code of Linux kernel to construct control-flow graph and call graph. In order to narrow the scope of scheduling in directed fuzzing, DDRace focuses on the interleavings of shared memory read and write instructions, which are most widely used in scheduling [21, 23, 30, 55]. DDRace first identifies the interfaces of the driver that can reach the target sites from the call graph. We refer to these identified driver interfaces as *target driver interfaces*. The *target race pairs* are the race pairs located in the control flow paths from target driver interfaces to target sites and DDRace also recognizes them. For example, in Figure 1, we recognize `ioctl$KDGKBSENT` and `ioctl$KDSKBSENT` as target driver interfaces, and I1/I2 as target race pairs.

3.2.3 Companion Race Pair Extraction

In addition to the memory access contained in the target driver interfaces, other driver interfaces may influence the thread interleavings of the target driver interfaces. However, due to the huge size of the Linux driver subsystem, it is not feasible to explore all of the code adequately. We narrow down the scope based on two key observations: (1) concurrent thread execution usually occurs in the driver interfaces of the same driver, (2) other driver interfaces usually have read and write operations on the same shared memory of the target ones. In particular, we further reduce the analysis scope to memory access that can affect the control/data flow of the target driver interfaces through concurrency execution. Specifically, we find shared variables that are involved in read operations in the target driver interfaces, and then find write operations on the same shared variables in other driver interfaces by points-to analysis. We refer to these driver interfaces that have concurrency associations with the target driver interfaces as *concurrency companion driver interfaces*, and refer to the race pairs located in the control flow paths starting from the concurrency companion driver interfaces as *companion race pairs*.

3.3 Directed Scheduling Fuzzing

After identifying the fuzzing scope (driver interfaces and race pairs), DDRace performs concurrency directed grey-box fuzzing to find concurrency UAF vulnerabilities. We design multiple concurrency vulnerability tailored mechanisms that enable DDRace to explore the concurrency thread interleavings in a directed manner towards the targets (§3.3.1). Based on the feedback, we develop new seed preservation and selection policies (§3.3.2) and seed mutation techniques (§3.3.3).

3.3.1 Distance Metrics and Feedback Mechanisms

We propose several new vulnerability-related distance metrics to guide DDRace to approach the target sites. In particular, DDRace models the conditions of control-flow, data constraints, and thread interleaving using dominator depth distance, vulnerability model constraint distance, and race pair interleaving path feedback, respectively.

Dominator Depth Distance. Prior work [7, 60] define a distance metric that measures how close a test case is to the target sites at the basic block level. However, they greedily prioritize the test cases with the shortest distance from the call graph but do not consider the control or data flow conditions.

Test cases that exhibit shorter distances might be useless, e.g., they can not satisfy the control-flow precondition to reach the target site.

To better measure the current execution distance to the target site, DDRace uses the depth of the dominator tree instead of the basic block distance. DDRace only cares about whether the current execution can reach the target sites instead of which path it should take. The formula of distance calculation is based on AFLGo's harmonic mean. Similarly, a smaller value of the sum means a closer distance to target sites.

Vulnerability Model Constraint Distance. We introduce a vulnerability model feedback mechanism to DDRace for better discovering concurrency UAF vulnerabilities. In addition to the control flow distance to the target sites, DDRace also computes the data flow distance that measures the probability of satisfying the vulnerability constraints.

The UAF model is summarized as the following three constraints: (1) there is a deallocation operation and a use operation, (2) the targets of these two operations are about the same pointer or memory, and (3) the use operation is executed after the free operation. When one constraint is satisfied, the distance will be decreased accordingly, denoting the decrease of difficulty in triggering the vulnerability. If all constraints are satisfied, the constraint distance is reduced to 0. The constraint feedback can be utilized as guidance for thread scheduling (i.e., the vulnerability constraints can only be satisfied with the correct scheduling).

Race Pair Interleaving Path Feedback. Traditional fuzzers use coverage feedback and favor test cases that trigger new coverage (e.g., hitting new edges or increasing edge-hit count). With only such coverage feedback, the fuzzers are unaware of the interleavings among threads. Therefore, they cannot sufficiently prioritize test cases to explore the dimension of thread interleavings.

To efficiently expose concurrency UAF vulnerabilities, a desired fuzzer is expected to thoroughly explore the thread interleavings. Since concurrency UAF vulnerabilities occur when multiple threads illegally access certain memory, we propose a new feedback metric, Race Pair Interleaving Path coverage (`RPIP` in short), to help with thread-interleaving exploration during fuzzing. Our solution is based on the intuition that the memory access patterns of shared variables can indicate thread interleaving situations among threads. A different memory access pattern of shared variables suggests a different thread interleaving. §3.2 identifies a set of race pairs, and we use the access order of the race pair among threads as the *thread-interleaving edge*. We refer to the set of thread-interleaving edges in a fuzzing trial as the *thread-interleaving path*. DDRace then favors test cases triggering new thread-interleaving paths during the exploration (see §3.3.2 for more details). Different from "alias coverage" in KRACE [55], `RPIP` tracks the value and interaction order of shared variables to infer thread-interleaving edge.

Algorithm 1 shows the details of `RPIP` in DDRace. In a fuzzing trial, for each shared variable in the race pairs obtained in §3.2, DDRace maintains a read tuple *RT* and a write tuple *WT* (line 1-2), which store the information about the last read or write access. A tuple is in the form of $< ID_{instruction}, ID_{thread}, Value_{shared\ variable} >$. During the fuzzing, DDRace accordingly updates the *RT* and *WT* based on the following criteria for every instruction (line 4):

i. Whenever executing a memory read instruction, DDRace first updates *RT* (line 6). If the *WT* has a record (line 7) and its thread is different from the one of read instruction (line 9) , there is a write-to-read thread-interleaving edge and DDRace inserts the edge $< i_w, i >$ to $InterEdge$ (line 10).

ii. If it is a memory write instruction that updates the value of the corresponding shared variable, DDRace will update *WT* (line 14-16). If the thread and value of the read operation in *RT* are all different from the ones of the last write instruction (line 17-19), there is a read-to-write thread-interleaving edge and DDRace inserts the edge $< i_r, i >$ to $InterEdge$ (line 20).

After obtaining thread-interleaving edges ($InterEdge$), we sequentially integrate them as a thread-interleaving path and calculate a hash for the path, which we refer to as `RPIP`. DDRace uses `RPIP` for later seed preservation and selection. Like prior sequential path coverage guided fuzzers [22, 58], the thread-interleaving path coverage also faces the puzzle of path explosion. DDRace thus separately calculates a RPIP for each global variable of the race pairs identified in §3.2.3 and §3.2.2, instead of combining all global variables together. This allows DDRace to get rid of excessive overhead caused by exploring all the scheduling space.

3.3.2 Seed Preservation and Selection

We define the final distance of a test case as the sum of domination depth distance and vulnerability constraint distance. We regard a test case as interesting and preserve it as a seed in two dimensions. In the first dimension, a test case is preserved if it introduces a new distance value that has not been seen before, even though it is not the shortest distance. Considering that some target sites are dependent on specific thread scheduling to reach them, a test case may not get the shortest distance in a certain execution, but it can reach the targets with a change in scheduling, so we will keep it.

In the second dimension, DDRace leverages `RPIP`. As mentioned earlier, `RPIP` suffers from the path explosion issue. We optimize this type of feedback in the seed preservation stage. In particular, if a test case hits an undiscovered (i.e., not being hit by the fuzzer before) thread-interleaving edge, DDRace saves it into corpus directly. Besides, DDRace adopts a path prioritization formula similar to CollAFL [22], which calculates a weight for a path based on the number of undiscovered neighbor branches. Slightly differently, we calculate the

Algorithm 1 Thread-interleaving edge calculation.

Input: T - test case, t - thread, v - value
Output: $InterEdge$ - thread-interleaving edge

```
1: RT ←<>
2: WT ←<>
3: InterEdge ←<>
4: for i ∈ instructions(T) do
5:    if isReadAccess(i) then
6:       RT ←< i,t,v >
7:       if isNotEmpty(WT) then
8:          i_w,t_w,v_w ← WT
9:          if t ≠ t_w then
10:            InterEdge.insert(< i_w,i >)
11:         end if
12:      end if
13:   else
14:      oldValue := GetOperandValue(i)
15:      if oldValue ≠ v then
16:         WT ←< i,t,v >
17:         if isNotEmpty(RT) then
18:            i_r,t_r,v_r ← RT
19:            if t ≠ t_r & v ≠ v_r then
20:               InterEdge.insert(< i_r,i >)
21:            end if
22:         end if
23:      end if
24:   end if
25: end for
```

weight of undiscovered neighbor thread-interleaving edges for a path p as shown in Formula 1. e denotes the thread-interleaving edge in path p, and NE denotes the set of all neighbor edges. Neighbor thread-interleaving edges are obtained during each fuzzing trial. DDRace only preserves a test case if its weight exceeds the average weight value.

$$Weight(p) = \sum_{\substack{e \in p \\ <e,e_i> \in NE}} IsUndiscovered(< e,e_i >) \quad (1)$$

The seed selection in DDRace is two-fold. First, DDRace prioritizes seeds with shorter distances because a shorter distance often indicates that the seed is more beneficial for triggering vulnerabilities. Second, DDRace prioritizes seeds that exhibit *rare thread-interleaving paths*. Past practices have shown that exercising rare paths or branches allows testing extreme situations and can better expose vulnerabilities [8, 37], however, they do not leverage paths about thread interleavings. To the best of our knowledge, DDRace is the first work that prioritizes seeds in rare thread-interleaving paths over common paths to achieve better thread-interleaving exploration.

3.3.3 Seed Mutation

DDRace employs a two-stage mutation mechanism for the seeds selected from the fuzzing corpus.

We design a scheduling pseudo-syscall `setdelay` to arrange the thread scheduling. In general, `setdelay` accepts an array pointer argument that can be used to determine which race pairs to schedule and delay the given time. In the first stage of mutation (i.e., the initial stage before reaching the target sites), DDRace adopts the same strategy as Syzkaller, such as adding or deleting syscalls, mutating syscall parameters, etc. In particular, DDRace ensures the existence of `setdelay` in the syscall sequences and mutates its parameter. At the second stage (i.e., after reaching target sites), DDRace focuses on thread scheduling mutation, that is, increasing the mutation probability accompanying driver interface and `setdelay`.

3.4 Adaptive State Migration

Recent Linux kernel fuzzers execute test cases continuously without restarting the target kernel or resetting its state. Therefore, the state of the kernel is constantly changing during fuzzing, resulting in poor reproducibility of test cases. Because a preserved seed may behave differently (e.g., achieving different code coverage or distance) in different states, such poor reproducibility makes distance-guided fuzzing strategy unstable, thus seriously hurting the performance.

To avoid this dilemma, we utilize snapshots to store kernel states. We then restore appropriate states and apply the test cases to them when needed. However, it is not feasible to store all states due to huge overhead. We propose a novel adaptive state migration approach that makes a trade-off between accuracy and overhead. The core idea is that we create and restore snapshots on demand. Specifically, we re-execute the executed test cases sequentially in a clean target kernel to traverse all past states triggered by the fuzzer, and create snapshots of the states required for reproducing. Then we can restore the snapshots to enter the required states.

The detailed steps are as follows:

i. When a *fuzzing instance* is launched, an identical *snapshot instance* is also launched, running with the same configuration. The fuzzing instance starts the fuzzing loop after startup, creates a snapshot S_0 in the initial state, and waits for inputs.

ii. When there is a highly valuable but not reproducible test case p_i in the fuzzing instance, DDRace will add it to the queue `snapWorkQueue`. Specifically, a test case is valuable if it has reached the target sites or obtained shortest distance in the fuzzing instance, and is not reproducible if it cannot get the same effect in a repeated test.

iii. The snapshot instance keeps polling the `snapWorkQueue`. Once it receives an item, there is a new test case that needs to be executed in the proper state. Snapshot instance does not immediately synchronize the state with the fuzzing instance due to the high overhead, but first restores to the initial state S_0, then executes the test case p_i. If the feedback of the

execution is as expected (e.g., the target sites can be reached), the test case will be marked with snapshot id S_0 and saved into the corpus. If not, go to step iv.

iv. If the initial state of the kernel is not sufficient to meet the conditions required by p_i, DDRace will take extra effort to keep the snapshot instance synchronized with the fuzzer instance. Specifically, the snapshot instance will re-execute all the test cases before p_i that fuzzer instance has executed since the fuzzer instance boots and then creates a snapshot S_i. It then runs the test case p_i in this state. If the execution result is as expected, we will save p_i marked with snapshot id S_i into the corpus. If it does not, we think that p_i may have been interfered by other factors, such as interrupts, external input, etc., and we will discard the test case.

4 Implementation

We implemented a prototype of DDRace with around 5.6K LoC. Table 1 summarizes the components of DDRace. We plan to release our prototype implementation to facilitate future research upon publication. We describe the implementation of the main components in detail below.

Race Pair Extraction. To implement the race pair extraction component, we first construct the control-flow graph and call graph based on CRIX [40] that builds a precise call graph with static analysis. We then extract the target driver interfaces from the generated graphs by LLVM Passes. Next, we utilize SVF [48] to perform a points-to analysis to collect the instructions accessing shared variables and identify them as race pairs. The points-to analysis only focuses on the instructions located on the paths from the corresponding driver interfaces to the target sites. Points-to analysis in general cannot scale to complex programs with large code base. As Linux kernel is highly modular, we partition kernel objects to sub-module and perform points-to analysis for each sub-module. This is a common practice used by prior works [30, 40].

Directed Scheduling Fuzzing Loop. We implement the fuzzing component of DDRace based on the kernel fuzzing tool Syzkaller [53]. We modify Syzkaller to receive the distance and `RPIP` feedback for prioritizing inputs. Since Syzkaller randomly selects threads for each syscall in a test case, which may cause two system calls that should be concurrent to execute sequentially. To resolve this problem and better preserve the "concurrency characteristics" of the test cases, we also save the thread ID information of each syscall as an appendix to the seed when a seed is preserved in the corpus. Therefore, our fuzzer can reproduce the thread scheduling for syscalls when the seeds are selected for future mutation and re-execution.

Table 1: Implementation details of DDRace.

Component	Base Tool	LoC
Race Pair Extraction	SVF, CRIX, LLVM	1,500 (C++)
Fuzzing Loop	Syzkaller	2,500 (Go)
Instrumentation	LLVM SanitizerCoverage	600 (C++)
Glue scripts	-	1,000 (Python)
Total	-	5,600

Snapshot and Instrumentation. Our state migration scheme is implemented with the snapshot feature of QEMU. DDRace utilizes LLVM SanitizerCoverage [39] to instrument the target Linux kernel for perceiving the feedback information. We also modify the system call `setdelay` for the scheduling. Specifically, we instrument before the race pair instructions and invoke `mdelay` for scheduling. The delay time is determined by the parameters of `setdelay`. DDRace mutates its parameters as for other syscalls, i.e., design a specific template for that `setdelay`, and then mutate the parameters according to the template. DDRace uses the feedback mechanisms mentioned in §3.3.1 to decide if the mutation of `setdelay` is favored, and accordingly preserves it. All these tracking instructions are instrumented via LLVM 10.0.

5 Evaluation

In this section, we comprehensively evaluate DDRace from various aspects, to answer the following questions:

- How effective is DDRace at extracting target-related race pairs?
- What is the capability of DDRace in exposing concurrency UAF vulnerabilities?
- How is DDRace comparable to existing approaches?
- How do the techniques in DDRace contribute to its performance?

5.1 Experimental Setup

Dataset. We evaluate DDRace on upstream Linux device drivers to understand its efficacy. The fuzzing component of DDRace is implemented atop Syzkaller, which employs QEMU and has limited support for Linux drivers. It is non-trivial and time-consuming for us to configure Linux drivers for the evaluation. Therefore, we currently limit our dataset to six drivers: `tty`, `drm`, `sequencer`, `midi` , `vivid` and `floppy` that are well supported by QEMU. The evaluated Linux drivers are quite popular and used by a large population of devices. We evaluate the drivers in the Linux kernel 4.19.100.

Environment and Configuration. In our experiments, we utilized LLVM-10.0 to compile the Linux kernel with the configuration of syzbot [2]. Specifically, we enabled KCOV for code coverage instrumentation and KASAN sanitizer for use-after-free bugs detection. All experiments were conducted on a machine running Ubuntu 16.04 LTS with 2 Intel(R) Xeon(R) CPU cores (E5-2695 v4 2.10GHz) and with 384GB RAM.

Table 2: Vulnerabilities found by DDRace in Linux 4.19.100. The column "File Names of Target Pairs" shows the file names of the FREE and USE target sites. File names of unfixed vulnerabilities are temporally removed for security concerns. † denotes 0-day vulnerabilities newly discovered by DDRace. Among them, Vul. #1, #2, and #6 have been assigned with CVE-2022-1280, CVE-2022-1419 and CVE-2022-1652.

Vul. ID	Driver	File Names of Target Pairs	Status
1	drm	drivers/gpu/drm ↔ drivers/gpu/drm	Confirmed†
2	drm	drivers/gpu/drm/drm_gem.c ↔ drivers/gpu/drm/vgem/vgem_drv.c	Confirmed & Fixed†
3	drm	drivers/gpu/drm/drm_gem.c ↔ drivers/gpu/drm/vkms/vkms_gem.c	known
4	drm	drivers/gpu/drm/drm_auth.c ↔ drivers/gpu/drm/drm_ioctl.c	known
5	floppy	drivers/block/floppy.c ↔ drivers/block/floppy.c	Confirmed & Fixed†
6	floppy	drivers/block/floppy.c ↔ drivers/block/floppy.c	Confirmed & Fixed†
7	tty	drivers/tty/vt/selection.c ↔ drivers/tty/n_tty.c	known
8	tty	drivers/tty/vt/keyboard.c ↔ drivers/tty/vt/keyboard.c	known
9	tty	drivers/tty/vt/vt_ioctl.c ↔ drivers/tty/vt/vt.c	known
10	tty	drivers/tty/vt/vt_ioctl.c ↔ drivers/tty/vt/vt.c	known
11	sequencer	sound/core/seq/seq_ports.c ↔ sound/core/seq/seq_ports.c	known
12	midi	sound/core/rawmidi.c ↔ sound/core/rawmidi.c	known

5.2 Race Pair Extraction

We run the original Syzkaller for 24 hours, and obtain 227 UAF pairs after analysis. We further analyze these potential UAF pairs to extract their related concurrency elements (i.e., race pairs). The number of related race pairs (including target race pairs and companion race pairs) are shown in the Figure 3. In our evaluation, most UAF pairs have fewer than 20 pairs of related race pairs. Unlike KRACE or RAZZER, our analysis is from the driver interface entry to the target sites, which significantly reduces the number of race pairs. Besides, we identify race pairs in other driver interfaces that affect target sites, so that DDRace will not miss the key scheduling and interleaving feedback to trigger the vulnerability.

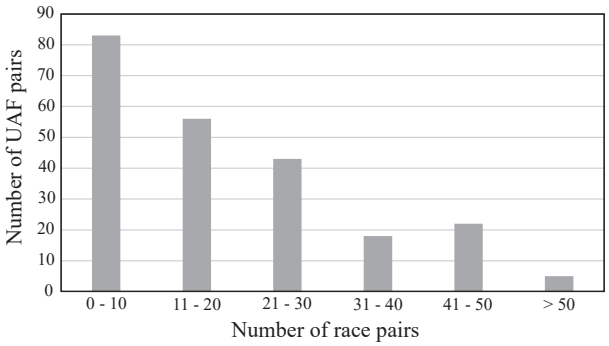

Figure 3: Race pairs statistics.

Figure 4 shows the statistics of corresponding driver interfaces (including target driver interfaces and concurrency companion driver interfaces) to each UAF pair. It shows that most UAF pairs require no more than 9 driver interfaces. It means that instead of exploring all driver interfaces for a particular driver, we can allocate computing resources to a small number of interfaces and improve the efficiency of DDRace.

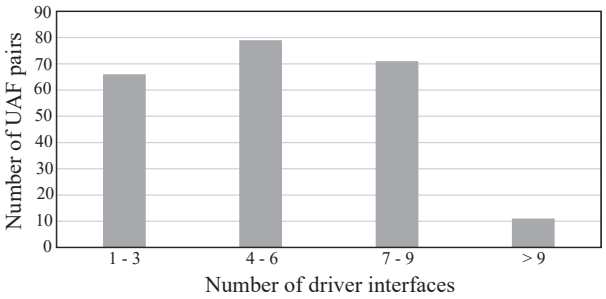

Figure 4: Driver interfaces statistics.

5.3 Vulnerability Discovery

As shown in §5.2, DDRace obtains 227 UAF pairs and their related race pairs. We then apply DDRace to perform concurrency directed fuzzing on the target drivers. For each UAF pair, we set a fuzzing time limit of 12 hours.

Table 2 shows the vulnerability-finding results. In general, DDRace is highly effective in finding concurrency UAF vulnerabilities. DDRace discovers 12 concurrency UAF vulnerabilities across all 6 evaluated drivers, including 4 previously unknown ones. We responsibly report all these new vulnerabilities to Linux maintainers and all of them are confirmed or fixed. Due to the Linux security policy, we temporally hide the file names for vulnerabilities that are not fixed yet.

As can be seen from the results, only 12 vulnerabilities are confirmed in 227 target UAF pairs, which indicates that just reaching the target code is not enough to find vulnerabilities (we will discuss it further in §5.5), and also indicates that the

conditions for triggering concurrency UAF crashes are relatively harsh. In fact, even though DDRace detects the crashes, it takes lots of manual effort to diagnose and reproduce these concurrency vulnerabilities.

The vulnerabilities found by DDRace are of high security consequences. Specifically, 3 of them are newly assigned with CVE IDs. For example, Vul. #2 is a newly discovered concurrency UAF vulnerability in `drm` driver. In this case, since root privilege is not required to access related devices, Linux maintainers believe that this vulnerability can be exploited for privilege escalation or docker escape. They explicitly prioritize this vulnerability in their vulnerability fix pipeline.

5.4 Comparison

To understand how is DDRace comparable to existing approaches, we further design controlled experiments and compare DDRace to state-of-the-art kernel fuzzing tools:

- **Syzkaller.** Syzkaller is a classic kernel fuzzer with basic code-coverage feedback. Though it does not support exploring the thread-interleaving space, it has been widely adopted in kernel fuzzing.
- **RAZZER.** RAZZER is a well-known syzkaller-based kernel race fuzzer, which randomly selects a race pair during fuzzing and uses hardware breakpoints for scheduling.
- **KRACE.** KRACE is another state-of-the-art kernel race fuzzer that uses the alias instruction pairs as the concurrency-coverage metric. KRACE injects random delays to schedule threads and does not prioritize the seed selection for concurrency seeds. Although KRACE has been open-sourced on Github [1], unfortunately, we fail to directly apply it to fuzz Linux device drivers because it primarily targets file systems and there lacks necessary documentation. Therefore, we take considerable engineering efforts to implement a variant of KRACE on our own. In particular, we port the alias instruction pair feedback and random delay scheduling to our variant of KRACE.

We configure Syzkaller, RAZZER and KRACE in their default options and apply them to the same set of Linux device drivers under the same kernel compilation options. We run the experiments for 5 runs, each with 12 hours.

Vulnerability Findings. We study the performance of each fuzzer by comparing the number of vulnerabilities they discover within the time limit. Due to the fundamentally random nature of fuzzing, a fuzzer might report diverse results in different runs even with the same configurations. We thus calculate the average number of vulnerabilities found by each tool in 5 testing runs.

We present the experiment results in the Venn diagram of Figure 5. The results show that DDRace is effective and significantly outperforms the compared tools. DDRace detects all concurrency vulnerabilities identified by other fuzzers. Specifically, DDRace, RAZZER, Syzkaller, and KRACE discover

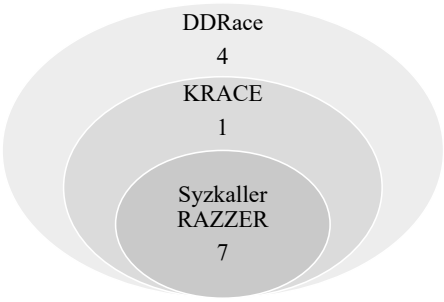

Figure 5: The Venn diagram of vulnerability discovery results on the Linux kernel 4.19.100.

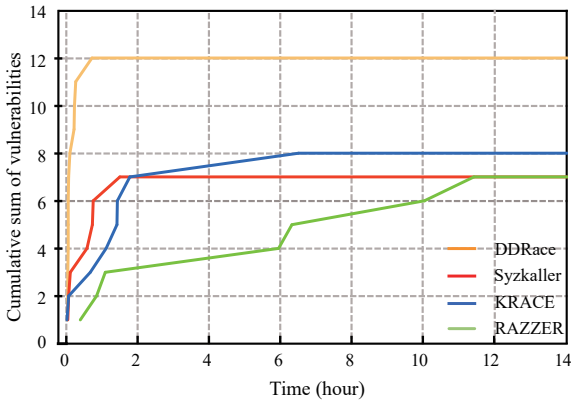

Figure 6: Comparison of vulnerability-finding time for 12 concurrency UAF vulnerabilities.

12, 7, 7, and 8 concurrency UAF vulnerabilities, respectively. DDRace outperforms Syzkaller, RAZZER and KRACE in vulnerability detection by 66.7%, 66.7% and 50%, respectively. The comparison also demonstrates the importance of race pair extraction and direction guidance for narrowing down the scope of concurrency UAF fuzzing.

Performance and Speed. Besides the number of vulnerabilities a tool could find, we evaluate how much time a tool needs to trigger the vulnerabilities. Therefore, we measure the time used for triggering each vulnerability per fuzzer.

We depict the trend of the cumulative number of vulnerabilities detected by a fuzzer over time in the persistent kernel fuzzing of 12 hours. As shown in Figure 6, DDRace detects all these vulnerabilities much earlier and faster than the other tools. It uses around 40 minutes to find all the vulnerabilities, while Syzkaller, KRACE and RAZZER take over 90 minutes, 6 hours, and 11 hours, respectively, yet find much fewer vulnerabilities. This demonstrates the better efficacy of DDRace in exposing concurrency UAF vulnerabilities.

We notice that although KRACE discovers one more vulnerability than Syzkaller ultimately, its progress in discovering vulnerabilities is slower in the early fuzzing stage (first 90 minutes). We investigate the case and find this might be caused by the random delay for scheduling in KRACE, which

wastes excessive computing resources on a large number of repeated useless thread interleavings. This observation can also be deduced from the evaluation results in the KRACE paper [55].

5.5 Ablation Study

To further understand how each technique in DDRace contributes to the final results, we present an ablation study. Since DDRace contains three major components—`RPIP` feedback, concurrency seed selection, and adaptive state migration, we correspondingly design 3 variants of DDRace by disabling a feature per variant, respectively, and compare the full-fledged DDRace with them. For the variant $DDRace_D$, we enable only the distance feedback. The variant $DDRace_R$ uses the feedback dimension of distance and `RPIP` tracking. Atop $DDRace_R$, the variant $DDRace_P$ further enables our concurrency seed selection strategy. The adaptive state migration scheme is disabled in all the 3 variants but is activated in full-fledged DDRace. All of the above variants use the results of our static analysis in §5.2 and perform scheduling during fuzzing. Further, in order to better compare with traditional sequential directed fuzzing and understand the contribution of static analysis and scheduling, we also add another variant $DDRace_{PD}$ that does not utilize scheduling (e.g., `setdelay`) or static analysis results, and only enables pure distance feedback. Similarly, we conduct the ablation experiments for 5 runs, each for 12 hours.

Figure 7 shows the mean results and its 95% confidence interval error bars. First, the concurrency factors (e.g., scheduling, race pairs or concurrency companion driver interfaces) are particular helpful to the efficacy of DDRace. $DDRace_{PD}$ has the worst performance in exposing concurrency UAF vulnerabilities compared to all other variants, i.e., spending much longer time in the majority of the cases. In our tests, it fails to find Vul. #1 and #4 within 12 hours. Second, the active scheduling and feedback can further improve the fuzzing effectiveness. Compared to $DDRace_D$, $DDRace_R$ could trigger concurrency UAF vulnerabilities more efficiently, e.g., it improves the vulnerability exposure time by 68.7% above $DDRace_D$. This further demonstrates that directed fuzzers that only focus on reaching target sites (e.g., SemFuzz [60]) are not enough for finding concurrency UAF vulnerabilities in Linux. Third, compared to randomly selecting concurrency seeds, the seed priority strategy of DDRace can increase the speed of crash triggering by up to 35.3% (7.1% on average), especially for cases where the root cause of the vulnerability is the interleaving of multiple race pairs. The variant using adaptive state migration can reduce the vulnerability exposure time by up to 46.7% in the cases with frequent seed failures because of the changing kernel states.

From Figure 7 we also observe that, for concurrency vulnerabilities that can be easily triggered in a short time, the performance gain of full-fledged *DDRace* is not significant. It is reasonable, because the interleaving condition and kernel state needed to trigger these vulnerabilities are not complicated, and random thread interleaving could also easily satisfy the condition and trigger the vulnerabilities. Nevertheless, for complex vulnerabilities such as Vul. #1, #8 and #12, DDRace achieves a significant performance improvement.

6 Discussion

6.1 Code Exploration Stage

Like other DGFs, DDRace provides an exploration stage in which it uses code coverage feedback to reach different code. In this stage, DDRace will not greedily select the seeds closest to the target sites. This is reasonable because it allows DDRace to satisfy other implicit conditions for reaching target sites such as data constraints. Besides, DDRace adopts the simulated annealing algorithm for power scheduling like AFLGo. In particular, the energy of every seed (i.e., the time spent on the seed during fuzzing) is approximately equal initially; As the fuzzing goes, the seeds closer to the target sites are assigned higher priority and energy.

6.2 Syscall Dependency

Recent works [24, 33, 42] shed light on extracting the dependency between syscalls. They focus on composing valid syscall sequences during fuzzing to achieve higher coverage. However, they do not consider concurrency associations during the process. For example, moonshine [42] infers the order of syscalls based on their parameters and return values. We believe these works are orthogonal to DDRace, and can be applied to DDRace to improve DDRace's performance. For instance, based on these works, DDRace can infer whether two driver interfaces must be executed sequentially and discard race pairs whose memory operations cannot be executed concurrently (e.g., the race pairs whose memory operations are in both `open` and `write` syscalls), making the fuzzing more efficient. We leave the integration as future work.

6.3 Limitations and Future Work

6.3.1 Negatives in Static Analysis

DDRace performs points-to analysis to obtain the race pairs related to the target UAF pairs. Naturally, the static analysis can have false positives. Since the results of the static analysis will be used by the dynamic fuzzing, the false positives can be ultimately filtered out at runtime. Besides, DDRace performs partial static analysis, e.g., it mainly focuses on the read and write instructions in the driver sub-module. If there is a race across different kernel modules, it will cause false negatives. The false negatives might lead to miss of some concurrency feedback, and hinder the efficiency of thread interleaving exploration. However, such false negatives are rare

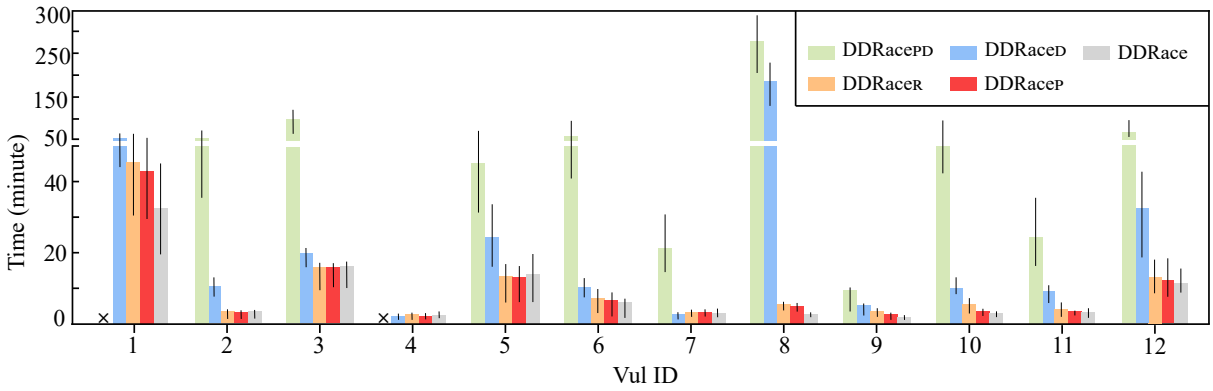

Figure 7: Mean time used for triggering the concurrency UAF vulnerabilities in Linux-4.19.100. *DDRace$_{PD}$* only enables the pure distance feedback. Based on *DDRace$_{PD}$*, *DDRace$_{D}$* utilizes scheduling and static analysis results, and *DDRace$_{R}$* enables the feedback dimension of distance and `RPIP` feedback, while *DDRace$_{P}$* further enables our concurrency seed selection strategy. DDRace is the full-fledged system. × denotes a case is not triggered within the time limit.

because different kernel modules generally do not interact frequently. We can extend the partial static analysis to the whole kernel to resolve this problem.

6.3.2 Thread Scheduling

In addition to scheduling race pairs that can change thread interleaving, scheduling synchronization primitives (such as locks and semaphores) can also affect program behaviors. Their interleavings can be seen as thread interleaving feedback as well. Similar to existing works [30, 55], our current implementation does not support such scheduling synchronization primitives. In the future, we plan to add such scheduling synchronization primitives to scheduling candidates and perform more precise thread scheduling. Besides, as mentioned in §3.3, we currently only focus on the thread-interleaving edge of the same global variable. To get more fine-grained feedback, we can expand it to support more global variables in the future.

6.3.3 Non-Deterministic Cases in State Migration

For performance reasons, our state migration scheme does not store the complete state of the kernel faithfully during fuzzing. As discussed in §3.4, DDRace takes into account the deterministic behaviors of executed test cases, but it does not consider the impact of other non-deterministic behaviors, such as interrupts, non-reproducible thread interleavings, etc. Nevertheless, our evaluation results have demonstrated the effectiveness of our current state migration scheme.

6.3.4 Portability

DDRace currently targets concurrency UAF vulnerabilities in Linux drivers due to their prevalence. However, the techniques proposed in DDRace can be applied to other programs or other type of concurrency vulnerabilities.

On the one hand, with additional engineering efforts, DDRace can be extended to other target programs, including other kernel modules and user-space libraries. By utilizing the modular structure of Linux drivers and clear driver interfaces, DDRace can limit the scope of static analysis to a few interfaces of the driver under test, which improves the accuracy. Except for this, other parts of DDRace is generic for different types of targets. For other kernel modules (e.g., the file system and network modules) and user-space thread-unsafe library code, we could develop similar static analysis solutions (maybe with lower accuracy) to identify candidate locations and apply DDRace to discover concurrency vulnerabilities. We leave it as a future work to develop proper static analysis solutions for more types of target programs and then apply DDRace to find vulnerabilities.

One the other hand, our concurrency distance metrics are also helpful for directed fuzzing to find other types of concurrency vulnerabilities, e.g., concurrency double-free, concurrency null-pointer-dereference, and concurrency out-of-bound bugs. The major modifications we need to make are components related to the vulnerability model, i.e., target site recognition (§3.2.1) and constraint distance (§3.3.1). We also leave it as future work to extend DDRace to support these types of vulnerabilities.

7 Related Works

7.1 UAF Detection

Many detectors report UAF bugs by monitoring memory accesses at runtime [11, 35, 51]. Since they require workloads

to trigger the relevant code and thread interleaving, their effectiveness largely depends on the inputs.

Some works use static analysis [57, 59] or dynamic fuzzing [41, 54] to discover UAF bugs, However, they mainly focus on sequential UAF bugs. DDRace instead investigates concurrency UAF vulnerabilities in Linux device drivers. DCUAF [4] extends static lockset analysis to identify concurrency UAF, and suffers from false positives. UFO [27] analyzes execution traces and utilizes model checking to infer the thread causality. However, such SMT-based solutions are usually applicable to a bounded window of events in practice, as larger windows will cause constraint-solving infeasible. ConVul [12] can identify exchangeable events and is able to detect concurrency memory corruption vulnerabilities, but it relies on execution traces instead of actively generating input.

7.2 Concurrency Bug Discovery

7.2.1 Data Race Detection

In the past decades, there has been lots of work on finding concurrency bugs, especially data race bugs. They can be generally classified into static and dynamic approaches. Static race detectors [5, 6, 19, 34, 52] analyze the source code to find instructions that access the same memory locations and instructions that can be concurrently executed without proper protections, and report potential data races in the program. Dynamic approaches apply these analyses at runtime. RaceTrack [62], FastTrack [20], PACER [9], and CONVUL [12] perform happen-before analysis to determine the intrinsic order of shared memory accesses and whether they can execute concurrently. Eraser [45] and Goldilocks [18] use lockset analysis to identify shared memory accesses that are not guarded by the same lock. Some works combine both of them, such as Helgrind [29] or ThreadSanitizer [3].

In general, these detectors usually suffer from a high false-positive rate, and can hardly distinguish between benign and harmful data races. Rather than detecting data race, DDRace focuses on concurrency vulnerabilities that can cause memory corruption (i.e., UAF).

7.2.2 Thread-Interleaving Exploration

SKI [21] applies the randomized PCT [10] algorithm for thread scheduling, which can increase the probability of exposing concurrency bugs. SnowBoard [23] further optimizes SKI by recognizing and clustering potential memory communication and assigning them different priorities for scheduling. RAProducer [63] reproduces concurrency vulnerabilities by scheduling with hardware breakpoints, and exploring thread interleavings in both data flow and control flow dimensions. Maple [61] schedules by assigning threads priorities, and explores thread interleaving with a heuristic approach.

7.3 Fuzzing

7.3.1 Directed Fuzzing

Many researchers have paid great attention to directed fuzzing for crash reproduction, vulnerability verification, patch testing, etc. Besides the code coverage feedback, directed fuzzers design distance metrics to prioritize inputs that are more likely to reach the targets [7, 14, 17, 36, 60]. For example, AFLGo [7] computes the average distance of the basic blocks in the execution traces of an input to the fuzzing targets. Another line of research advances directed fuzzing by filtering out inputs that cannot reach the fuzzing targets. FuzzGuard [65] proposes a deep learning approach to predict the reachability of inputs and filters out those deemed unreachable ones. Beacon [26] employs a static analysis to infer the necessary conditions for reaching targets and avoids exploring infeasible paths. However, all these works are based on AFL and cannot be applied to complex software like Linux device drivers. They do not particularly investigate the dimension of thread interleaving in multi-threaded programs.

In terms of directed fuzzing for UAF , CAFL [36] points out that the orders and data conditions need to be considered. It thus designs a constraint-distance metric that precisely measures the distance and accurately prioritizes inputs. Unfortunately, CAFL studies sequential UAF vulnerabilities and cannot be used for fuzzing concurrency UAF vulnerabilities in Linux drivers. DDRace, on the other hand, targets concurrency UAF vulnerabilities as the first work.

7.3.2 Kernel Fuzzing

Linux kernels are an important target in fuzzing. Since system calls are the main interface for a program to interact with Linux kernels, the mainstream kernel fuzzers focus on generating system calls to test kernels [32, 53]. Besides employing code coverage as the feedback, many studies additionally utilize static analysis to better generate system calls to improve code coverage and testing efficiency [33, 42, 49]. For example, MoonShine [42], HEALER [49], and HFL [33] statically infer the dependencies between system calls to better generate syntactically valid inputs. DDRace also generates system calls to test Linux drivers. However, DDRace differentiates itself from existing works by targeting concurrency UAF vulnerabilities and the new directed scheduling fuzzing approach.

There are other channels such as memory mapped I/O (MMIO), direct memory access (DMA), and interruptions for Linux devices to interact with the kernel from the hardware side. Correspondingly, fuzzing techniques have been proposed to investigate erroneous operations in these channels. In particularly, Periscope [47] simulates the corresponding processing functions of peripherals input to the driver through hook-driven DMA/MMIO operations. USBFuzz [44] designs an emulated USB device and mutates device and configuration descriptors. VIA [25] extends the fuzzing scope to the

device driver interrupts and (un-)initialization. These works are orthogonal to DDRace as DDRace focuses on the input source of system calls rather than hardware devices. We will leave it as future work to investigate how to integrate such works with DDRace.

In addition to the Linux kernel, there are other fuzzers for commodity OS kernels including macOS, Windows, etc. Charm [50] designs a virtual machine that enables executing mobile device drivers without any hardware and performs fuzzing atop the virtual machine. IMF [24] retrieves the macOS system call dependencies by analyzing the dynamically collected traces, while SyzGen [15] performs symbolic execution for more explicit dependencies and system call specifications in closed-source macOS drivers. Digtool [43] proposes a virtualization monitor to capture kernel behaviors during fuzzing, as the Windows kernel does not have an available KASAN-like detector. NTFUZZ [16] presents a binary static analysis to infer the types of Windows system calls and performs a type-aware fuzzing on the Windows kernel. These works are orthogonal to DDRace. The techniques in DDRace are generic and can be naturally ported to these approaches to detect concurrency UAF vulnerabilities.

7.3.3 Race Fuzzing

Fuzzing has also been applied to find data race bugs. RaceFuzzer [46], ConAFL [38] and RAZZER [30] use static race detection tools to identify the potential race pairs, control the scheduling of these locations via instrumentation or VM hypervisor, and keep testing the program with various inputs and executive orders. Although they mark race pairs before fuzzing, they are still general fuzzers instead of directed fuzzers. In other words, they try to evenly explore paths or interleavings to reach the target sites. It is quite inefficient because they may put too much effort into some irrelevant branches or thread orders. DDRace resolves this problem by adopting directed fuzzing. DDRace computes both the distance to the target sites via domination depth feedback and the distance to satisfy the vulnerability model constraints, which can largely reduce the search space.

Muzz [13], Conzzer [31] and KRace [55] notice that traditional control flow coverage is not reliable feedback for concurrent programs, so they design new coverage metrics that are aware of thread interleavings (e.g., cross-thread execution path and memory access order) and thread context (e.g., pthread API calling sequence and stack frame). However, none of these metrics take the actual value of each read/write access into consideration. DDRace uses the Race Pair Interleaving Path feedback to grasp the data flow interference of threads, which we think can better balance the effectiveness and runtime overhead.

We characterize existing tools and DDRace in terms of the target programs, code availability, and target bug types. As shown in Table 3, most state-of-the-art race fuzzers cannot be directly applied to find concurrency UAF in Linux drivers. Specifically, most of the work target data race bugs, rather than concurrency vulnerabilities. However, it is ineffective to directly apply the approach of data race detection to find concurrency vulnerabilities. As discussed in [64], detecting a data race bug usually only considers a pair of concurrent accesses to the same memory locations, while detecting a concurrency vulnerability need to consider one or more memory operation pairs on a set of closely related memory locations. Besides, most of them are not designed for kernel drivers.

Table 3: Summary of concurrency fuzzers. `Con. Vul.` denotes concurrency vulnerability.

Tool	Target Program	Code Availability	Target Bug Type
RaceFuzzer [46]	(Java) Userspace	×	Data Race
ConAFL [38]	Userspace	✓	Con. Vul.
RAZZER [30]	Linux Kernel	✓	Con. Vul.
KRACE [55]	Linux File System	✓	Data Race
Muzz [13]	Userspace	×	Data Race
Conzzer [31]	Userspace & File System	×	Data Race
DDRace	Linux Drivers	✓	Con. Vul.

8 Conclusion

In this paper, we have presented a directed fuzzer, DDRace, for concurrency UAF vulnerability detection in Linux device drivers. DDRace entails overcoming several inherent challenges. It equips a new vulnerability-related distance metric and a novel concurrency feedback mechanism to assist directed fuzzing. DDRace resolves the test case reproducibility problem with a new adaptive kernel state migration scheme. Our extensive evaluation on upstream Linux drivers shows that DDRace is highly effective in discovering concurrency UAF vulnerabilities and it successfully detected 12 vulnerabilities, including 4 previously unknown ones with 3 CVEs assigned. DDRace significantly outperformed the related works by identifying more vulnerabilities more efficiently.

Acknowledgements

We would like to sincerely thank all the anonymous reviewers and our shepherd for their valuable feedback that greatly helped us to improve this paper. This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006, and HKPolyU Grant (ZVG0).

References

[1] Krace repository. https://github.com/sslab-gatech/krace.

[2] syzbot. https://syzkaller.appspot.com.

[3] Threadsanitizer. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[4] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency Use-After-Free bugs in linux device drivers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 255–268, Renton, WA, July 2019. USENIX Association.

[5] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. DSAC: Effective static analysis of Sleep-in-Atomic-Context bugs in kernel modules. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 587–600, 2018.

[6] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proc. ACM Program. Lang.*, pages 144:1–144:28, 2018.

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

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, pages 489–506, 2017.

[9] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 255–268, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143, 2012.

[12] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, page 706–717, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342. USENIX Association, August 2020.

[14] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[15] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, page 749–763, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 677–693, 2021.

[17] Zhengjie Du, Yang Liu, Yuekang Li, and Bing Mao. Windranger: A directed greybox fuzzer driven by deviationbasic blocks. In *IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022.

[18] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. *SIGPLAN Not.*, page 245–255, jun 2007.

[19] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, page 237–252, 2003.

[20] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. *SIGPLAN Not.*, page 121–133, jun 2009.

[21] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, page 415–431, USA, 2014. USENIX Association.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[23] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

[24] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 2345–2358, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. Via: Analyzing device interfaces of protected virtual machines. In *Annual Computer Security Applications Conference*, page 273–284, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 104–118, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[27] Jeff Huang. Ufo: Predictive concurrency use-after-free detection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 609–619, 2018.

[28] IBM. Tty drivers. https://www.ibm.com/docs/en/aix/7.2?topic=subsystem-tty-drivers.

[29] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F. Tichy. Helgrind+: An efficient dynamic race detector. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–13, 2009.

[30] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

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

[32] D. Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2011.

[33] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[34] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave Jing Tian. PASAN detecting peripheral access concurrency bugs within {Bare-Metal} embedded applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 249–266, 2021.

[35] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[36] Gwangmu Lee, Woo-Jae Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *USENIX Security Symposium*, 2021.

[37] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[38] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. A heuristic framework to detect concurrency vulnerabilities. *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[39] LLVM. Clang 10 documentation, sanitizercoverage. https://releases.llvm.org/10.0.0/tools/clang/docs/SanitizerCoverage.html.

[40] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check bugs via semantic- and Context-Aware criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1769–1786, Santa Clara, CA, August 2019. USENIX Association.

[41] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *RAID*, 2020.

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

[43] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. Digtool: A Virtualization-Based framework for detecting kernel vulnerabilities. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 149–165, Vancouver, BC, August 2017. USENIX Association.

[44] Hui Peng and Mathias Payer. USBFuzz: A framework for fuzzing USB drivers by device emulation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2559–2575. USENIX Association, August 2020.

[45] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, page 391–411, nov 1997.

[46] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[47] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Krügel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[48] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

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

[50] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 291–307, 2018.

[51] Erik Van Der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419, 2017.

[52] Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, pages 141–155. Citeseer, 2009.

[53] Dmitry Vyukov. Syzkaller. https://github.com/google/syzkaller, 2015.

[54] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, page 999–1010, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660, 2020.

[56] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425, 2015.

[57] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[58] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, page 598–609, New York, NY, USA, 2020. Association for Computing Machinery.

[59] Jiayi Ye, Chao Zhang, and Xinhui Han. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1529–1531, 2014.

[60] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, page 2139–2154, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, page 485–502, New York, NY, USA, 2012. Association for Computing Machinery.

[62] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. *SIGOPS Oper. Syst. Rev.*, page 221–234, oct 2005.

[63] Ming Yuan, Yeseop Lee, Chao Zhang, Yun Li, Yan Cai, and Bodong Zhao. *RAProducer: Efficiently Diagnose and Reproduce Data Race Bugs for Binaries via Trace Analysis*, page 593–606. Association for Computing Machinery, New York, NY, USA, 2021.

[64] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. Owl: Understanding and detecting concurrency attacks. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 219–230. IEEE, 2018.

[65] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. *FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-Box Fuzzing through Deep Learning*. USENIX Association, USA, 2020.