CARDSHARK: Understanding and Stablizing Linux Kernel Concurrency Bugs Against the Odds

Tianshuo Han, Xiaorui Gong, and Jian Liu, *{CAS-KLONAT, BKLONSPT}, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences*

https://www.usenix.org/conference/usenixsecurity24/presentation/han-tianshuo

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

CARDSHARK: Understanding and Stablizing Linux Kernel Concurrency Bugs Against the Odds

Tianshuo Han, Xiaorui Gong, Jian Liu*
{CAS-KLONAT †, BKLONSPT ‡}, Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
{hantianshuo, gongxiaorui, liujian6}@iie.ac.cn

Abstract

Concurrency bugs in the Linux kernel are notoriously difficult to reproduce and debug due to their non-deterministic nature. While they bring constant headaches to Linux kernel developers, the reasons behind the non-determinism and how to improve the efficiency in triggering concurrency bugs to ease the debugging process still need to be studied.

This work aims to fill the gap. We comprehensively study the concurrency bug stability problem in the Linux kernel, dissect the factors behind the non-determinism, and systematize the insights into a model to explain the non-deterministic nature of concurrency bugs.

Based on insights derived from the model, we identify an under-studied factor, named *misalignment*, which plays a vital role in triggering concurrency bugs. By controlling this factor, we significantly reduce the randomness in the concurrency bug-triggering process.

Inspired by this insight, we design a novel technique, named CARDSHARK, that can significantly improve the efficiency in triggering concurrency bugs when kernel instrumentation is possible. A variant of CARDSHARK, named BLINDSHARK, enables developers to improve efficiency in triggering concurrency bugs without knowing their root causes, making the use of CARDSHARK practical.

In our evaluation of 12 real-world concurrency bugs, CARDSHARK and BLINDSHARK significantly reduce the needed time and the number of attempts to trigger concurrency bugs in the Linux kernel. Notably, CARDSHARK can deterministically trigger 10 out of the 12 concurrency bugs with a single attempt. Our evaluation shows that CARDSHARK significantly outperforms existing works in stabilizing concurrency bugs, making it a potential great help to developers in analyzing and fixing concurrency bugs.

*Corresponding author: liujian6@iie.ac.cn
†Key Laboratory of Network Assesment Technology, CAS
‡Beijing Key Baboratory of Network Security and Protection

1 Introduction

Concurrency significantly enhances the Linux kernel by enabling efficient resource utilization, rapid response to user actions, and greater scalability, thus solidifying its role as the backbone of modern computing infrastructure. However, it also introduces unprecedented complexity, leading to a class of bugs known as concurrency bugs. These bugs arise from unexpected thread interleaving orders that developers often do not anticipate, potentially causing severe damage to the kernel's stability and security.

In addition to undermining the kernel's stability and leading to kernel panics, concurrency bugs are also known to open doors to memory corruption vulnerabilities [4–6], such as out-of-bound access, use-after-free, and double-free, which can be exploited to compromise the whole system. Its severity has been demonstrated by many successful concurrency bug-based exploits [1–3] that bypass the latest Linux kernel protections and achieve local privilege escalation on real-world devices in the past.

The non-deterministic nature of concurrency bugs poses significant challenges in their analysis and fixing. Coupled with the rapid evolution of fuzzing techniques, there are more bugs than developers can handle. In Linux kernel drivers, which is known to be a significant source of vulnerabilities [41,50], 19% of their bugs are concurrency bugs, which leads to a substantial number of concurrency bugs [50]. This overwhelming amount of hard-to-analyze concurrency bugs leads to the delay in fixing severe concurrency bugs [18], threatening the security and privacy of billions of Linux users.

In response to these challenges, it is imperative to develop new methods to efficiently trigger and debug concurrency bugs. Despite extensive research on detecting, reproducing, and exploiting these bugs [11–14,22–24,49,51–53], little has been done to understand the underlying factors contributing to their non-deterministic nature and systematize how concurrency bugs manifest. The latest related study, EXPRACE [26], suggests that increasing the time window size could improve the success rate in triggering concurrency bugs. However, our experiments indicate that, despite a larger time window, the triggering of concurrency bugs still suffers from significant non-determinism (§ 6.2), suggesting a more complex underlying nature.

In light of the need, we embarked on a comprehensive analysis of the manifestation process of concurrency bugs in the Linux kernel. We manually analyzed 12 real-world concurrency bugs and systematized our insights into a model, which we term as the Concurrency Bug Reproduction Model. This model highlights *misalignment*—the deviation in execution timing from the ideal timing required to trigger a bug—as a vital yet unstudied factor. By manipulating the elements that cause *misalignment*, we can improve the triggerability of concurrency bugs.

Armed with these findings, we developed an intuitive yet powerful technique, named CARDSHARK, that can substantially increase the success rate of triggering concurrency bugs by manipulating system call timing to minimize the *misalignment*. As shown in Table 2, CARDSHARK significantly reduces the time and number of attempts required to trigger concurrency bugs. Notably, CARDSHARK enables deterministic triggering of 10 out of 12 concurrency bugs in our dataset with a single attempt.

We further explored the potential of CARDSHARK and implemented a downstream application, named BLINDSHARK, that can be seamlessly incorporated into proof-of-concept programs of concurrency bugs to make them more efficient without knowing the root causes. The method employs a two-step approach: it first identifies an approximate *align time* and then compensates for the *misalignment* with it, thereby increasing the likelihood of achieving the desired execution interleaving. As shown in Table 3, BLINDSHARK significantly reduces the time and number of attempts needed to trigger concurrency bugs without knowing their root causes.

We believe that our work is a solid step toward understanding the intricacies of concurrency bug manifestation and demystifying its non-deterministic nature. In summary, our paper makes the following contributions:

- We comprehensively studied concurrency bugs and proposed the Concurrency Bug Reproduction Model to explain the manifestation process of concurrency bugs.
- Based on the model, we pointed out the importance of *misalignment* in the manifestation of concurrency bugs and designed the CARDSHARK to improve the efficiency of triggering concurrency bugs by manipulating it.
- We developed BLINDSHARK, a downstream application of CARDSHARK that can substantially increase the efficiency in triggering concurrency bugs without knowing vulnerability root causes or instrumenting the kernel.

We release the implementations and the experiment dataset at `https://github.com/keymaker-arch/CARDSHARK`. We believe these artifacts will benefit the community by helping with the evaluation of future concurrency bug stabilization techniques and potentially providing new insights.

2 Background

2.1 Concurrency Bug in the Linux Kernel

Concurrency bugs in the Linux kernel, a significant security concern, arise from the parallel execution of kernel functions [8]. Unlike non-concurrency bugs, which are triggered by reaching specific code paths, concurrency bugs require particular interleavings of multiple execution paths, needing events to occur in a precise sequence to manifest.

Typically, these bugs involve two paths: the *racee*, which consists of two events, and the *racer*, with a single event, as illustrated in Figure 1. A concurrency bug is triggered when the racer's event happens within the time window defined by the racee's events. While some concurrency bugs present more intricate scenarios involving additional execution paths or events, it is noteworthy that over half of these bugs involve just two racing execution paths, with a total of three events [26, 40]. This pattern underscores a typical structure in the manifestation of concurrency bugs.

The triggering of a concurrency bug often results in adverse outcomes like memory corruption, posing significant security risks. Attackers can exploit such vulnerabilities to conduct malicious attacks on the kernel, such as local privilege escalation (LPE) [1–3], underlining the critical need for a comprehensive understanding and effective management of these bugs.

2.2 Concurrency Bug Stability Issue

The triggering of concurrency bugs is inherently non-deterministic [9,12,23,26,49,51,53]. Unlike non-concurrency bugs, their activation relies on a precise chronological sequence of events that is unpredictable and cannot be directly controlled, making their occurrence random and difficult to analyze or fix [15, 16].

Although inherently non-deterministic, the triggerability of concurrency bugs varies. Some bugs can be easily triggered, requiring fewer race attempts for successful manifestation. The time window size is a recognized factor affecting this triggerability - larger time windows typically mean higher chances of bug occurrence. However, the exact reasons behind this non-deterministic nature and the factors influencing triggerability beyond the size of the time window remain largely unknown.

3 Overview

Problem Statement. Our research centers on enhancing the stability of triggering concurrency bugs in the Linux kernel. We aim to delve into this non-deterministic nature of concurrency bugs, dissect its causes, and subsequently improve concurrency bug-triggering stability.

Problem Scope. This study dissects the underlying causes of non-determinism in concurrency bugs. It develops user-level techniques to trigger these bugs more effectively without requiring modification of the kernel source code. We assume that an initial, though unstable, reproducer for the concurrency bugs is available. Moreover, our scope does not cover concurrency bugs involving more complex scenarios, such as multiple (more than two) execution paths or numerous (more than three) racing events, as these represent a minority according to existing studies [26, 40] and present distinct challenges.

4 Concurrency Bug Reproduction

The triggering of concurrency bugs is known to be inherently non-deterministic. While the time window size is a critical factor in this process, as highlighted in EXPRACE [26], it is not the sole determinant. This is exemplified by cases such as CVE-2022-1729 [47], where even a substantial time window does not guarantee deterministic triggering, as evidenced in Section 6.2. Often, hundreds of attempts are required to trigger such vulnerabilities once successfully, indicating that additional factors contribute to their non-deterministic nature.

In this section, we delve into the mechanics of concurrency bug reproduction and unravel the factors contributing to their non-determinism.

4.1 Concurrency Bug In The Vacuum

Triggering of a concurrency bug hinges on a specific interleaving during the parallel execution of kernel functions. This section delves into an idealized scenario, wherein this interleaving is reliably reproducible, to discern the critical elements for deterministic concurrency bug triggering.

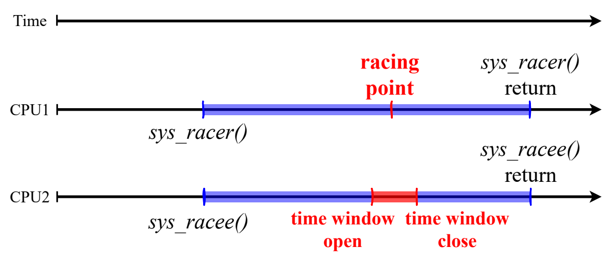

Figure 1: In an ideal environment where the racing point always falls into the middle of the time window, concurrency bugs can reproduce consistently.

Figure 1 depicts a typical concurrency bug pattern, characterized by two racing system calls, termed *sys_racer* and *sys_racee*. The bug is triggered when these system calls execute simultaneously, with the *racing point* of *sys_racer* aligning chronologically within *sys_racee*'s time window.

A *race attempt* is defined as a concurrent execution of *sys_racer* and *sys_racee*. In each *race attempt*, two threads (or processes) are spawned, each invoking a respective system call. Under ideal conditions, these two system calls commence synchronously, ensuring the *racing point* falls within the time window, as depicted in Figure 1. In such an ideal scenario, every *race attempt* leads to the specific execution interleaving necessary to trigger the concurrency bug, thereby enabling consistent bug triggering.

The inference drawn from this scenario is clear: deterministic triggering of a concurrency bug hinges on the *racing point* occurring within the time window for each *race attempt* consistently. However, this synchronicity of events remains a theoretical construct in this idealized setup and only sometimes translates to real-world situations. The following section will explore the challenges of reproducing concurrency bugs in real-world scenarios and why this idealized assumption often fails to materialize.

4.2 Concurrency Bug in Practice

In real-world scenarios, the *racing point* seldom consistently aligns within the time window in every race attempt. This non-determinism arises from the fact that user space programs have no precise control over the commence timing of racing point and the time window in real-world conditions. As a result, the *racing point* may randomly fall within or outside the time window in one race attempt. This issue is rooted in several functional aspects of the Linux kernel.

When a system call is invoked, it transitions the execution flow from user space to kernel space. The kernel then requires a certain duration to process the call. This indicates a fact that there is a time span from the invocation of *sys_racer* to the execution of *racing point*, and similarly, from the invocation of *sys_racee* to the time window midpoint. We term this duration as the *path time* of a system call.

The *path time* of a system call is an inherent property of that call. Thus, for any combination of *sys_racer* and *sys_racee*, their *path times* are inherently unequal. This difference in *path time* is referred to as *path disparity*. As the specific *sys_racer* and *sys_racee* differ in each concurrency bug context, *path disparity* becomes a unique characteristic of each bug. In real-world scenarios, the existence of *path disparity* often leads to the *racing point* not aligning with the time window, as illustrated in Figure 2.

Moreover, *invocation disparity* also plays a significant role in this context. This disparity emerges because parallel invocation of racing system calls typically requires the creation of two distinct threads (or processes). However, due to the

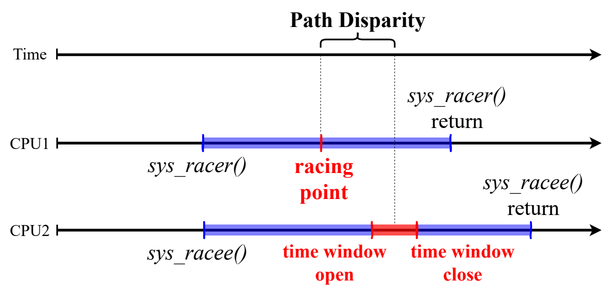

Figure 2: In reality, path disparity is one of the reasons why racing point does not always fall into the time window.

nature of the Linux kernel's scheduling policy, new threads do not start executing immediately after spawned [27]. An unpredictable delay between thread spawning and the commencement of execution is expected. Hence, when spawning two threads for racing system calls, a random disparity in their actual invocation timings is inevitable. This can lead to the *racing point* being misaligned with the time window, as shown in Figure 3.

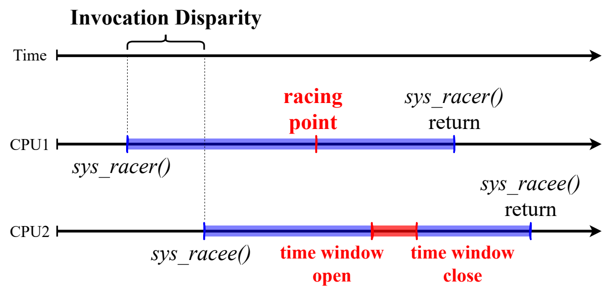

Figure 3: Random invocation disparity also introduces randomness to the concurrency bug reproduction process.

In summary, the existence of *path disparity* along with *invocation disparity* can lead to a timing difference between the racing point and the time window in a race attempt. Specifically, we define the timing difference between the *racing point* and the midpoint of the time window as *race timing misalignment*, or simply *misalignment*, as demonstrated in Figure 4.

4.3 Concurrency Bug Reproduction Model

Refining our understanding from previous sections, it is evident that *racing point* does not consistently align within the time window in real-world scenarios due to *misalignment*, a consequence of both *path disparity* and *invocation disparity*. While *path disparity* and the size of the time window (T_{win}) are intrinsic characteristics of a concurrency bug, *invocation disparity* introduces a variable element, manifesting as a random value in each race attempt.

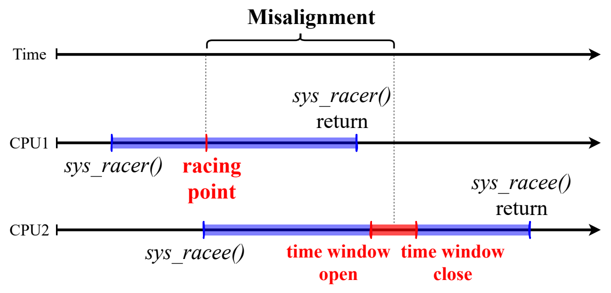

Figure 4: We define *misalignment* as the time difference between the racing point and the midpoint of the time window.

We formalize these concepts in this section. We represent the *misalignment* in a single race attempt as M, the *path disparity* as P, and the *invocation disparity* as E. The relationship between these factors is mathematically represented as:

$$M = E + P$$

As we define it, *misalignment* measures the time distance between the racing point and the *midpoint* of the time window. Specifically, *misalignment* takes a positive value when the racing point occurs before the midpoint of the time window and a negative value otherwise. As detailed in Section 4.1, a race bug is triggered when the *racing point* is positioned within the time window, meaning the distance between the racing point and the time window's midpoint does not exceed half of the time window's span. This condition can be formally expressed through the following inequality:

$$|M| < \frac{T_{win}}{2}$$

This inequality serves as the success criterion for a race attempt. It implies that a race attempt will successfully trigger the bug if the *misalignment* is not substantial enough to displace the *racing point* outside the time window. Essentially, this criterion serves as a quantifiable measure, guiding us in evaluating the likelihood of a race attempt to successfully trigger a concurrency bug, taking into account the interplay between *misalignment* and the time window.

This criterion succinctly delineates the difference between the idealized and practical scenarios, as detailed in Sections 4.1 and 4.2, respectively, and sits at the core of our Concurrency Bug Reproduction Model.

In the idealized setup of Section 4.1, we postulate synchronous execution of racing system calls ($E = 0$) and a consistent alignment of the *racing point* at the midpoint of the time window ($P = 0$). This scenario leads to a consistently null *misalignment* ($M = 0$), ensuring the criterion is perpetually satisfied. Consequently, each race attempt reliably triggers the concurrency bug.

In contrast, real-world conditions yield non-zero *path disparity* and *invocation disparity*. The stochastic nature of *invocation disparity* bestows an element of randomness upon the *misalignment*. Given the fixed time window length intrinsic to the bug, this criterion is only sometimes achievable. Hence, a race attempt triggers the bug in real-world conditions only when a fortuitous *invocation disparity* incidentally satisfies the success criterion.

The criterion also clarifies why bugs with larger time windows (T_{win}) are more prone to triggering yet lack deterministic certainty. A more extensive T_{win} increases the likelihood of a *misalignment* aligning with the criterion. This principle underpins techniques like EXPRACE [26], which extends T_{win} by injecting interrupts, enhancing the chance of criterion fulfillment. However, due to the inherent randomness of *misalignment*, even an enlarged T_{win} does not guarantee consistent bug triggering.

As will be demonstrated in Section 6.1, the ability to derive observable and verifiable properties evidence the correctness of the Concurrency Bug Reproduction Model.

5 CARDSHARK

In light of the Concurrency Bug Reproduction Model (§ 4.3), there are two ways to increase the likelihood of triggering a concurrency bug: 1) enlarging the size of the time window, and 2) decreasing the *misalignment*. While the existing concurrency bug stabilizing technique EXPRACE [26] focuses on enlarging the time window, we explore the possibility of stabilizing concurrency bugs by controlling *misalignment*.

5.1 Technique Overview

As previously discussed in Section 4.2, *misalignment* is contributed by two factors: *invocation disparity* and *path disparity*. Controlling *misalignment* therefore involves addressing both its contributing factors.

The *invocation disparity* stems from the timing variance in the invocations of the racing system calls. This is because a newly created thread (or processes) does not commence execution immediately after it is spawned, and a random timing disparity appears when we create two threads to invoke the racing system calls in parallel. Thus, *invocation disparity* can be nullified by precisely synchronizing the invocation timing of racing system calls in parallel threads.

We synchronize two system calls by inserting an execution barrier before them, ensuring the precisely synchronized invocation timing. The detailed implementation can be found in the Appendix 11.1.

Addressing the *path disparity* presents a more intricate challenge. As an intrinsic property of system calls, *path time* can not be directly altered by user space programs. Nonetheless, we can strategically manipulate the invocation timing of the racing system call precisely in user space to compensate for its influence. Specifically, as shown in Figure 5, we insert a precise delay before the invocation *sys_racer* to ensure that the racing point will fall into the time window. Conversely, we insert a delay before *sys_racee* in case racing point falls behind the time window.

Theoretically, if this delay precisely equals the *path disaprtiy*, it can offset the racing point within the time window and achieve consistent concurrency bug triggering. We refer to this delay as *align time*. However, in reality, *path disparity* cannot be precisely measured without knowing the root cause of the concurrency bugs. As a result, we can only significantly increase the chance of triggering concurrency bugs but not reproduce them consistently in practice.

To summarize, our control over *misalignment* consists of two steps:

1. Precisely synchronizing the invocation timing of racing system calls to eliminate random *invocation disparity*.

2. Injecting a delay equal to the *path disparity* into the system call invocation process to compensate for the *path time dispartiy*.

We have termed this method of manipulating *misalignment* as CARDSHARK. Implementing CARDSHARK theoretically nullifies *misalignment*, leading to consistent alignment of the racing point within the time window, as illustrated in Figure 5.

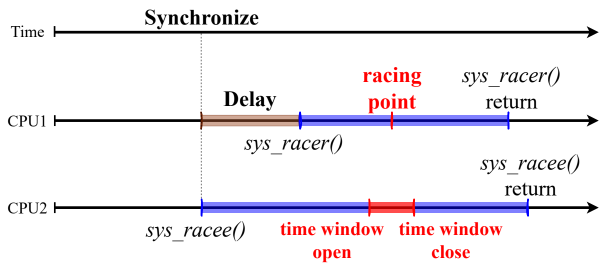

Figure 5: CARDSHARK injects a delay to racing system calls to increase the chance of racing point falling into the time window, thus triggering concurrency bugs.

5.2 Algorithmic Implementation of CARDSHARK With Presumed Align Time

For clarity in this section, we assume that the *align time* has already been accurately determined, allowing us to focus on the principle of CARDSHARK. In the following section, we will demonstrate the efficacy of CARDSHARK without this assumption.

As established before, CARDSHARK controls *misalignment* by addressing its two contributors. This is achieved by meticulously manipulating the invocation timings of the racing system calls. The specific steps of this approach are detailed in Algorithm 1.

CARDSHARK operates by precisely controlling the invocation timing of system calls to manipulate *misalignment*

Algorithm 1 CARDSHARK Algorithm

```
 1: Global Variable: delay_racee, delay_racer
 2: function THR_RACEE
 3:     SYNCHRONIZE
 4:     DELAY(delay_racee)
 5:     SYS_RACEE
 6: end function
 7: function THR_RACER
 8:     SYNCHRONIZE
 9:     DELAY(delay_racer)
10:     SYS_RACER
11: end function
12: function RACE_ATTEMPT(delay1, delay2)
13:     PIN_CPU(0)
14:     THREAD_CREATE(thr_racee)
15:     PIN_CPU(1)
16:     THREAD_CREATE(thr_racer)
17:     PIN_CPU(2)
18:     THREAD_JOIN
19: end function
20: function CARDSHARK(align_time)
21:     delay_racee ← 0
22:     delay_racer ← 0
23:     if align_time > 0 then
24:         delay_racer ← align_time
25:     else
26:         delay_racee ← -align_time
27:     end if
28:     RACE_ATTEMPT
29: end function
```

using a predetermined *align time*, as outlined in Algorithm 1. If *misalignment* is positive, the *align time* is applied when invoking *sys_racer*; if negative, it is applied to *sys_racee* (lines 21 to 27).

At its core, CARDSHARK relies on the *RACE_ATTEMPT* function (lines 12 to 19) to launch two threads for the racing system calls. The main thread sets its CPU affinity with the *PIN_CPU* primitive before creating the two threads to ensure that they commence execution in parallel (lines 12 to 19). The threads, executing *thr_racer* and *thr_racee* functions, synchronize using the *SYNCHRONIZE* primitive for coordinated execution. Following synchronization, one thread waits for the *align time* before calling its system call, while the other executes its call immediately.

This algorithm's prototype, including the implicated primitives, is presented in pseudo-C code in the appendix11.1.

The efficacy of CARDSHARK under this assumption is assessed in Section 6.2. To overcome this assumption and enhance CARDSHARK's practical applicability, the following section outlines the methodology for identifying a feasible *align time*.

5.3 BLINDSHARK: CARDSHARK with Aprpoximated Align Time

The practical application of CARDSHARK hinges on a viable *align time*. This section explores methodologies for determining this parameter for a concurrency bug.

A straightforward approach to identify *align time* involves calculating the *path time* difference between the racing system calls. This method necessitates measuring the *path time* of each racing system call and the time window, achievable through kernel instrumentation. For *sys_racer*, the *path time* is determined by the interval between its invocation and the racing point, both pinpointed via instrumentation. Likewise, the *path time* and the time window for *sys_racee* are ascertained through similar kernel instrumentations.

While this direct measurement approach to determine *align time* is conceptually straightforward, its real-world application faces complexities. It involves complex steps, including understanding the root cause of the concurrency bug to identify the racing point and the time window, which defeats the purpose of stabilizing the triggering of concurrency bugs for analysis and fixing.

We propose a brute-force BLINDSHARK method as an alternative, which obviates the need for deep insights into the concurrency bug's root cause or kernel code instrumentation. This approach is premised on the fact that the *range* of *align time* can be determined quickly, and it is feasible to test for all possible *align time* to determine a viable one.

The principle behind BLINDSHARK is that *path time* does not surpass the total execution time of the racing system calls since racing point and the time window are integral to the execution path. Consequently, the difference in *path times*, or *misalignment*, will not exceed the bounds of the execution time of the system calls.

BLINDSHARK methodically determines *align time* by iterating through all possible values. Initially, it measures the execution times of *sys_racer* and *sys_racee*, then introduces incremental delays to each, monitoring for kernel crashes. By testing every potential *align time*, a kernel crash at a specific *align time* indicates a viable candidate. The algorithm of BLINDSHARK, as shown in Algorithm 2, efficiently navigates through all possible *align times*, leveraging kernel crashes as indicators for a feasible *align time*.

BLINDSHARK begins by measuring the execution times of racing system calls with the *MEASURE* function (defined in lines 1 to 6). The *READTIME* primitive reads high-resolution timers like time stamp counters [20, 26, 35] or HPET devices [37]. Next, from lines 10 to 21, BLINDSHARK systematically explores potential *align times* by incrementally applying them to *sys_racee* and *sys_racer* during race attempts, utilizing the *CARDSHARK* function (Algorithm 1). Appendix 11.2 provides a prototype implementation in pseudo-C code.

It is worth mentioning that the feasible *align time* is not a single, fixed value; rather, it encompasses a range. An *align*

Algorithm 2 BLINDSHARK Algorithm

```
function MEASURE(syscall)
    start ← READTIME
    SYSCALL
    end ← READTIME
    return end - start
end function
function BLINDSHARK
    T_racer ← MEASURE(sys_racer)
    T_racee ← MEASURE(sys_racee)
    for t in [0, T_racer] do
        CARDSHARK(-t)
        if CRASH then
            return -t
        end if
    end for
    for t in [0, T_racee] do
        CARDSHARK(t)
        if CRASH then
            return t
        end if
    end for
end function
```

time is considered suitable if it adjusts the racing point to fall within the defined time window. Due to the inherent non-deterministic elements of CARDSHARK, as will be elaborated in Section 5.4, the ideal *align time* would place the racing point at the midpoint of the time window. However, BLINDSHARK does not explicitly aim to locate this exact *align time*. It terminates its search upon the occurrence of the first kernel crash, offering the associated *align time* as the output. This resulting *align time* can shift the racing point to a position within the time window but does not necessarily align it with the window's center unless the *path disparity* for the concurrency bug is zero.

To sum up, we do not claim that the viable *align time* obtained by Algorithm 2 is the optimal *align time*. Instead, it finds a viable or approximated *align time* that can improve stability in triggering concurrency bugs. Nonetheless, as evidenced in Section 6.3, the approximated *align time* determined by BLINDSHARK proves to be significantly effective in increasing the stability of triggering concurrency bugs.

5.4 Non-determinism Factors in CARDSHARK

While CARDSHARK is conceptually capable of eliminating *misalignment* to deterministically trigger concurrency bugs, achieving this in practical scenarios is hampered by inherent non-determinism factors. This section delves into these factors and assesses their impact on deploying CARDSHARK in real-world settings.

As delineated in Section 5, CARDSHARK endeavors to control *misalignment* by meticulously addressing its two contributors. The first contributor, *invocation disparity*, arises due to the variable timing of racing system call invocations in user space. Since this disparity originates within user space, it can be effectively eliminated with precise synchronization primitives implemented in user space.

However, addressing the second contributor, *path disparity*, is not as straightforward. This disparity arises due to variations in *path time* of the racing system calls. Since *path time* is inherent to the system call and can not be directly altered by user space programs, the resulting *path disparity* can not be directly manipulated or seamlessly eliminated.

In CARDSHARK, *path disparity* is not eliminated directly; instead, its influence on the concurrency bug triggering is mitigated by strategically introducing a delay in the user space. This indirect approach to managing *path disparity* unfortunately adds a layer of non-determinism to CARDSHARK, primarily due to the inherent variability of *path times* for racing system calls in real-world scenarios.

By definition, *path time* refers to the duration from invoking a system call in user space to the execution of a specific point in kernel space (for *sys_racer*, this point is the *race point*, and for *sys_racee*, it is the midpoint of the time window). This duration encompasses the transition from user to kernel state and the execution of specific kernel functions. During any given race attempt, the *path time* may extend due to non-deterministic occurrences such as interrupts [26] or scheduling processes [25, 27]. These events necessitate additional handling by the kernel, thereby prolonging the *path time*.

Despite the absence of non-deterministic events, *path time* remains variable. Various factors can influence its duration. Notably, hardware characteristics such as memory cache and CPU frequency scaling significantly impact the execution time of instructions. For instance, the time to execute a given set of instructions increases if a memory cache miss occurs or when the CPU operates at a lower frequency [19, 34, 38]. Furthermore, kernel utilities, like memory allocators and locking mechanisms, exhibit variability in their execution times. For example, the SLUB memory allocator demands more time to allocate a memory chunk when it needs to request a new page from the buddy system [25, 55]. Similarly, the duration for acquiring locks within the kernel can also vary.

As such, the actual *path disparity* in race attempts under real-world conditions is inconsistent. Consequently, pinpointing an exact *align time* that ensures the complete mitigation of its effects proves challenging. Nonetheless, the strategy of diminishing its impact by injecting a delay, as implemented in CARDSHARK, effectively addresses its influence on the triggering of concurrency bugs. As will be demonstrated in Section 6.2, CARDSHARK can adeptly manage *misalignment* and enhances the stability of concurrency bug triggerings, even when the *path disparity* is inconsistent.

Vulnerability	Race Attempts	Racee Path	Racer Path	Path Disparity	Time Window
CVE-2017-2636 [43]	262	6824	1120	5704	11398
CVE-2017-7533 [45]	147	35486	36411	925	7983
CVE-2021-26708 [44]	17	7031	2006	5025	10061
CVE-2021-32606 [46]	>10000	4076	2408	1668	17899
CVE-2022-1729 [47]	379	134674	52400	82274	77731
CVE-2023-31083 [48]	>10000	4397	3372	1025	1207
CVE-2017-15265 [42]	>10000	4027	16391	12364	1878
423ecfea77 [30]	47	141132	52886	88246	262551
3c4f8333b5 [29]	2	306635	5852	300783	608908
20f2e4c228 [28]	>10000	13279	4218	9061	1292
458c15dfbc [31]	967	53005	40207	12798	5180
4ccf11c4e8 [32]	481	41591	8675	32916	65832

Table 1: Precise timing evaluation of 12 real-world concurrency bugs. The time unit in the table is TSC. The *path disparity* in the table are absolute values for simplicity. The result shows that the time window size is not the only factor contributing to the randomness of the concurrency bug reproduction process.

6 Evaluation

In this section, we evaluate the effectiveness of CARDSHARK technique by answering the following research questions:

RQ1: Is it true that the larger the race time window is, the more stable concurrency bugs can be reproduced? Moreover, is it the only factor behind the non-determinism of concurrency bug reproduction?

RQ2: How much improvement does CARDSHARK offer in enhancing the efficiency of triggering concurrency bugs by manipulating *misalignment*?

RQ3: Without a thorough analysis of the root causes of concurrency bugs (hence, without knowing the ideal alignment time), is BLINDSHARK capable of determining an effective *align time* and successfully stabilizing concurrency bugs?

RQ4: How does the performance of CARDSHARK compare with the existing concurrency bug stabilizing technique EXPRACE [26]?

To address these questions empirically, we devised a set of experiments focusing on real-world concurrency bugs.

Experimental Setup. Our experiment framework was established on a machine with an Intel i5-8700 CPU and 32GB of memory.

Instead of running the experiments in virtual machines, we ran all the experiments on the *physical machine* to avoid unexpected influence introduced by virtual machines. We use a separate Raspberry Pi 4B+ to automate the experiment process by enabling serial console communication between the physical test machine and the Raspberry Pi controller. The controller sends commands to the physical machine to boot into target kernels, run exploits, and restore the physical machine's state when exploits succeed (causing kernel panic) or fail (after more than 10,000 race attempts).

In each experiment, a concurrency bug that succeeds in triggering or fails to trigger is run 50 times to ensure the statistical significance of the result.

Dataset. We diligently collected concurrency bug reproducers from existing works [26,49,55], syzbot [7], and Linux kernel commits [36]. Note that EXPRACE is another work in stabilizing concurrency bugs, and their work was evaluated on 10 real-world bugs. We contacted the authors for bug reproducers used in their evaluation but did not obtain the complete set of functional reproducers. Eventually, we obtained 12 real-world concurrency bugs for our evaluation.

Among the concurrency bugs, some had publicly available reproducers. These existing reproducers merely attempt to invoke the racing system calls repetitively without any form of stabilization. We use them as the baseline in our evaluation. For bugs lacking a public reproducer, we conducted root cause analysis and developed the corresponding reproducers by ourselves.

To ensure the triggerability of real-world concurrency bugs and minimize the influence of extraneous kernel features, we compiled the last known vulnerable kernel version for each concurrency bug based on default configurations (*defconfig*) for the experiment.

6.1 Empirical Verification of the Model

We first designed an experiment to empirically verify the correctness of the Concurrency Bug Reproduction Model to

Vulnerability	Time To Reproduce		Race Attempts	
	Baseline	CARDSHARK	Baseline	CARDSHARK
CVE-2017-2636 [43]	0.1	0.1	262	1
CVE-2017-7533 [45]	0.1	0.1	147	1
CVE-2021-26708 [44]	0.1	0.1	17	1
CVE-2021-32606 [46]	61.1	0.1	>10000	1
CVE-2022-1729 [47]	1.1	0.1	379	1
CVE-2023-31083 [48]	10.2	0.1	>10000	1
CVE-2017-15265 [42]	5.2	0.2	>10000	2524
3c4f8333b5 [29]	0.1	0.1	2	1
423ecfea77 [30]	0.1	0.1	47	1
20f2e4c228 [28]	26.8	1.1	>10000	5152
458c15dfbc [31]	0.1	0.1	967	1
4ccf11c4e8 [32]	0.2	0.2	481	1

Table 2: The time cost (in second) and race attempts for CARDSHARK and baseline approach to trigger concurrency bugs. CARDSHARK significantly reduces the number of needed race attempts to trigger concurrency bugs by manipulating *misalignment*, indicating the correctness of the Concurrency Bug Reproduction Model.

answer RQ1.

In this experiment, we analyzed the root causes of all the concurrency bugs in our dataset and instrumented the kernels at the racing point and the time window to record the exact execution timing of the baseline bug reproducer (the ones with no stabilization techniques) to verify our model. We use time stamp counter (TSC) [20, 35] as a precise time measurement, which is also used in existing research works needing precise timing control [55].

The results in Table 1 indicate that reproducing concurrency bugs becomes challenging with significantly small racing time windows. This finding aligns with prior research suggesting that larger time windows increase the likelihood of bug reproduction [26]. Nonetheless, this pattern does not hold universally to all cases, suggesting that time window size is not the sole determinant of concurrency bug non-determinism.

6.2 CARDSHARK Efficacy

We structured another experiment to evaluate if controlling *misalignment* enhances the efficiency of triggering concurrency bugs, directly addressing RQ2.

The experiment consists of two distinct phases. In the first phase, we reused the kernel instrumentation infrastructure from the last experiment and used the measured *path disparity* as the precise *align time* to be used in CARDSHARK. In the second phase, we apply the precise *align time* delay to trigger the concurrency bug, launching CARDSHARK technique. Here, we inject the precise *DELAY* by injecting a busy loop before the system call invocation and constantly monitoring the elapse of time by constantly reading TSC, similar to what is used in the Context Conservation technique in existing works [55].

In this experiment, we modified the baseline exploits by incorporating invocation synchronization and *align time* injection to create CARDSHARK reproducers. Each version, whether baseline or CARDSHARK, was tested 50 times on a physical machine to determine success or failure. Specifically, for some bugs like CVE-2021-32606 [46], the baseline group required an exceptionally high number of attempts to trigger the bugs. For simplicity, we capped the maximum number of attempts at 10,000 for these baseline experiments.

The results are detailed in Table 2. As shown in the table, CARDSHARK significantly reduces the average time and number of attempts required to trigger concurrency bugs. This substantial improvement strongly suggests that *misalignment* is a key factor contributing to the non-determinism of concurrency bugs.

Notably, CARDSHARK enables deterministic triggering of 10 out of the 12 bugs in our dataset with just one attempt. However, CVE-2017-15265 [42] and 20f2e4c228 [28] are not deterministically triggered by CARDSHARK. As Table 1 shows, these two bugs have very narrow time windows combined with relatively large *path time*, indicating that the non-deterministic factors, which are not addressed by CARDSHARK as discussed in Section 5.4, still significantly influence the triggering of some concurrency bugs. This is particularly evident in cases like CVE-2017-15265 and 20f2e4c228, where a large *path time* are more likely to be influenced by the non-deterministic factors that can offset

Vulnerability	**Time To Reproduce**		**Race Attempts**	
	Baseline	BLINDSHARK	Baseline	BLINDSHARK
CVE-2017-2636 [43]	0.1	0.1	262	10
CVE-2017-7533 [45]	0.1	0.1	147	5
CVE-2021-26708 [44]	0.1	0.1	17	10
CVE-2021-32606 [46]	61.1	0.1	>10000	22
CVE-2022-1729 [47]	1.1	0.1	379	9
CVE-2023-31083 [48]	10.2	0.1	>10000	37
CVE-2017-15265 [42]	5.2	0.3	>10000	3037
3c4f8333b5 [29]	0.1	0.1	2	1
423ecfea77 [30]	0.1	0.1	47	2
20f2e4c228 [28]	26.8	1.2	>10000	7963
458c15dfbc [31]	0.1	0.1	967	4
4ccf11c4e8 [32]	0.2	0.2	481	2

Table 3: The time cost (in second) and attempt number for BLINDSHARK and baseline approach to trigger concurrency bugs. BLINDSHARK can significantly stabilizing conrrency bugs without knowing their root causes or performing modifications to kernel source code.

the racing point outside of the narrow time window. Future improvements in CARDSHARK should aim to handle these non-determinism factors, as will be discussed in Section 7.4. Nonetheless, the overall efficacy of CARDSHARK supports the validity of our Concurrency Bug Reproduction Model.

6.3 BLINDSHARK Efficacy

In this section, we assess BLINDSHARK's ability to identify an approximated *align time* without prior knowledge of the concurrency bug's root causes or kernel instrumentation, addressing RQ3.

The experiment utilizes the same set of concurrency bugs as in Section 6.2 but with a realistic setting. Specifically, here, BLINDSHARK determines the *align time* without kernel instrumentation or knowing the root cause, and it uses this approximated *align time* for bug triggering. The experimental setup remains consistent as before.

We detail the evaluation result in Table 3. As shown in the table, even without knowing the root causes or kernel instrumentation, BLINDSHARK can still significantly reduce the time and number of attempts required to trigger the concurrency bugs.

BLINDSHARK's performance improvement over the baseline is less pronounced compared to CARDSHARK's. This is attributed to the less accurate approximation of *align time*, stemming from the absence of precise kernel instrumentation. Nonetheless, BLINDSHARK offers a practical solution for developers to reproduce, analyze, and fix concurrency bugs in real-world scenarios.

6.4 Evaluation Against Existing Work

In this section, we evaluate the performance of our technique against the existing concurrency bug stabilization technique EXPRACE [26] on our dataset to answer RQ4. As discussed in Section 4, EXPRACE is primarily designed for non-inclusive multi-variable data races, but its underlying principles apply to the concurrency bugs analyzed in this paper.

EXPRACE offers four distinct variant implementations: *reschedule*, *membarrier*, *TLB-shootdown*, and *HW-interrupt*. According to EXPRACE's evaluation, the *HW-interrupt* variant, which extends the time window by receiving network packets, is the most performant one over all variants in stabilizing non-inclusive data races. However, the performance of *HW-interrupt* variant significantly depends on the network latency, yet the information is not available in the paper. Consequently, we implemented the *membarrier* variant, which operates by sending memory barrier signals from user space to the local CPU, and should thus provide relatively consistent and good performance across different experimental settings. While we do not claim these values precisely represent EXPRACE's performance, they are our best effort to reproduce their results.

Results are presented in Table 4. The result illustrates that while *membarrier* reproducers can improve the reproducibility of concurrency bugs, the improvement does not apply to all bugs. In other cases, *membarrier* achieves comparable performances with the baseline reproducers. This observation conforms with what they reported in the original EXPRACE paper. EXPRACE's limitation arises because the success of

Vulnerability	Time To Reproduce			Race Attempts		
	Baseline	Membarrier	BLINDSHARK	Baseline	Membarrier	BLINDSHARK
CVE-2017-2636 [43]	0.1	0.1	0.1	262	10	10
CVE-2017-7533 [45]	0.1	0.1	0.1	174	5	5
CVE-2021-26708 [44]	0.1	0.1	0.1	17	10	10
CVE-2021-32606 [46]	61.1	39.3	0.1	>10000	>10000	22
CVE-2022-1729 [47]	1.1	0.4	0.1	379	331	9
CVE-2023-31083 [48]	10.2	58.7	0.1	>10000	>10000	37
CVE-2017-15265 [42]	5.2	2.1	0.3	>10000	>10000	3037
3c4f8333b5 [29]	0.1	0.1	0.1	2	2	1
423ecfea77 [30]	0.1	0.1	0.1	47	32	2
20f2e4c228 [28]	26.8	9.2	1.2	>10000	>10000	7963
458c15dfbc [31]	0.1	0.1	0.1	967	1077	4
4ccf11c4e8 [32]	0.2	0.2	0.2	481	494	2

Table 4: The time cost (in second) and the attempt number for each tool to trigger concurrency bugs. BLINDSHARK performs the best in reproducing concurrency bugs compared to baseline and EXPRACE's *membarrier* variant.

triggering concurrency bugs depends on both *misalignment* and the size of the time window. While injecting *membarrier* into reproducers can probabilistically increase the size of the time window, it does not eliminate the impact of *misalignment*.

On the other hand, while working on the same condition (not knowing the root cause and no instrumentation), BLINDSHARK can significantly stabilize *all* concurrency bugs, which further outlines the accuracy of the Concurrency Bug Reproduction Model compared to EXPRACE's time-window-centric model.

7 Discussion

7.1 Limitation of CARDSHARK

While CARDSHARK offers a robust solution, its effectiveness is not uniform across all concurrency bugs. This section explores the limitations of CARDSHARK and identifies scenarios where its application may be less effective or impractical, explaining the reasons behind these constraints.

For certain concurrency bugs in the Linux kernel, one (or even both) of the racing execution paths may not be initiated via a system call in the user space. For instance, the concurrency bug fd3d91ab1c [33] emerges from the concurrent execution of a system call in user space and a process triggered by the physical detachment of a USB device. The strength of CARDSHARK lies in its ability to meticulously manage the initiation timing of system calls from user space. Consequently, it becomes inapplicable for concurrency bugs where a racing path is not initiated by a user-space system call.

Moreover, CARDSHARK's reliability diminishes in scenarios involving complex concurrency bugs with more than three interdependent events and in non-inclusive data races, as described in some existing studies [26,40]. While applying the CARDSHARK in these situations is technically feasible, the outcome is uncertain. CARDSHARK allows for the satisfaction of some, but not necessarily all, interleaving constraints. Fully enforcing all the interleaving is challenging due to the inherently random nature of the interleaving between the two execution flows.

In essence, the comprehensive applicability and effectiveness of CARDSHARK for addressing all types of concurrency bugs in the Linux kernel remain unclear, and its performance on certain bugs has not yet been thoroughly evaluated.

7.2 Mitigation

Although CARDSHARK and BLINDSHARK are designed to facilitate analysis and debugging of concurrency bugs, they could theoretically be incorporated into concurrency bug exploitations and improve their success rate. To mitigate this exploitation risk, two approaches are proposed: (1) detecting abnormal frequencies of system call invocations, and (2) restricting non-root users from directly accessing high-resolution timers.

The first approach involves monitoring system call invocations at the kernel level, as exploiting concurrency bugs requires frequent system calls with identical parameters. By introducing small random delays to these system calls, the kernel can disrupt the precise timing control required by CARD-

SHARK and BLINDSHARK, impairing their functionality.

The second approach restricts access to high-resolution timers, which CARDSHARK and BLINDSHARK rely on to inject precise delays and mitigate *misalignment*. Limiting access would straightforwardly render these tools ineffective. However, on some specific hardware platforms, there are high-resolution timers provided at the hardware level (such as the Time Stamp Counter on modern x86 platforms). Restricting these timers may require corresponding hardware-specific patches.

7.3 Combination of EXPRACE and CARDSHARK

As suggested by the Concurrency Bug Reproduction Model, we can further improve CARDSHARK's ability to stabilize concurrency bugs because CARDSHARK focuses on reducing *misalignment* while EXPRACE tries to increase the race time window.

However, unfortunately, these two techniques are not compatible. This is because EXPRACE treats interrupt injections as random occurrences, not ensuring deterministic injection within the time window. Even if CARDSHARK injects a precise *align time* to synchronize the system call invocation and compensate for the *path disparity*, EXPRACE can still inject an interrupt before the *racing event* or *time window*, destroy the precise timing control.

7.4 Future Work

Enhancing CARDSHARK's Capabilities: Section 5.4 points out that CARDSHARK does not resolve all non-determinism factors in concurrency bugs, making deterministic triggering a complex challenge. Identifying and managing these non-deterministic factors is a critical direction for future research.

Furthermore, as Section 7.1 discusses, the effectiveness of CARDSHARK varies when applied to specific types of concurrency bugs. A detailed evaluation of CARDSHARK's applicability and potential refinements for specific bug categories is a priority for future investigations.

Additionally, it is worth noting that CARDSHARK currently represents a basic application of our Concurrency Bug Reproduction Model. The model's broader capabilities still need to be explored. We plan to develop and introduce more refined techniques based on a deeper understanding of it.

Enhancing *align time* Search Algorithm: As discussed in Section 5.3, BLINDSHARK can determine a viable *align time*, not the optimal one. We plan to improve the algorithm of BLINDSHARK to automatically determine the optimal *align time* in the future.

Integration with Syzkaller: Syzkaller [17] is the state-of-the-art kernel fuzzer aiming to discover kernel bugs, including concurrency bugs. However, many concurrency bugs uncovered by Syzkaller are not reproducible. Many of these bugs cause a kernel crash during the fuzzing process and never reproduce [15, 16]. Analyzing the root cause of such concurrency bugs without a reproducer is highly challenging, leaving many concurrency bugs delayed in fixing or even unfixed.

As an open-source project, we examined Syzkaller's bug reproduction mechanism. We noted its approach to reproducing concurrency bugs: it repetitively and blindly makes race attempts, similar to the baseline settings in our evaluation experiments (see Section 6.2). Given that CARDSHARK can be fully automated with the *align time* determined by BLINDSHARK, we propose that CARDSHARK can be integrated into Syzkaller to enhance its ability to generate concurrency bug reproducers.

Expanding CARDSHARK to Other Systems: While CARDSHARK is primarily designed for addressing concurrency bugs in the Linux kernel, its underlying principles are not exclusive to this context. The core foundation of CARDSHARK, the Concurrency Bug Reproduction Model, is versatile and not inherently limited to the context of the Linux kernel. Consequently, we plan to broaden the scope of CARDSHARK, applying it to other operating systems and user space programs in the future.

8 Related Works

Concurrency Bug Triggering. The latest work in this field, EXPRACE [26], focuses on enhancing the triggerability of non-exclusive data races by utilizing interrupts. EXPRACE initially noted that certain data race bugs are difficult to trigger due to a smaller time window for the racee compared to the racer. It categorized these as non-exclusive data races. EXPRACE's findings suggest that without expanding the racee's time window, these non-exclusive data races remain untriggerable. To address this, EXPRACE increases the racee's time window by introducing interrupts.

While EXPRACE is designed for non-exclusive data races, its principles apply to broader concurrency bug contexts. The approach highlighted the significant influence of time window size on triggering concurrency bugs. EXPRACE's model correlates the success rate of triggering non-exclusive data races with the ratio of the time window sizes. However, as analyzed in our study, the triggering of concurrency bugs depends on the interplay between *misalignment* and time window size. This implies that EXPRACE presupposes a constant alignment of time windows between the racer and racee (*misalignment* equals zero). According to our analysis in Section 4.2, this assumption is not a natural occurrence.

In essence, while EXPRACE's method of invoking interrupts can increase the triggerability of non-exclusive concurrency bugs, it does not fully address the fundamental aspect of *misalignment*. This oversight limits their understanding of the root causes behind the observed improvements in bug triggerability.

Concurrency Bug Discovering. Many research efforts have been dedicated to identifying concurrency bugs in the Linux kernel, employing diverse methodologies ranging from static to dynamic analysis. Works like those by Erickson et al. [14], Engler and Ashcraft [13], Vojdani et al. [51], Ryan et al. [49], Deligiannis et al. [12], Voung et al. [52], and Xu et al. [53] adopt static analysis approaches. These methods scrutinize the code without executing it, aiming to pinpoint potential concurrency issues. In contrast, dynamic analysis techniques, as used in works like SEGFUZZ [23], Razzer [22], PACER [11], and DRDDR [24], involve analyzing the code during its execution to uncover bugs that manifest at runtime.

Among these contributions, Ryan et al. [49] introduced the innovative Probabilistic Lockset Analysis (PLA), which utilizes memory access patterns to predict and rank the likelihood of kernel races. This approach adds a probabilistic dimension to race detection, offering insights into race conditions' potential severity and frequency. Deadline [53] brings a different perspective by precisely defining and pinpointing double-fetch bugs in OS kernels. It employs symbolic checking, systematically identifying and detecting vulnerabilities involving multiple reads, enhancing the reliability of kernel operations.

On the dynamic analysis front, SEGFUZZ [23] dissect thread interleavings into manageable segments, paving the way for exploring new interleavings and identifying potential race conditions. Razzer [22] merges static analysis with deterministic thread interleaving, effectively locating and triggering data races that might otherwise remain undetected.

Additionally, Syzkaller [17] holds a unique position as it not only detects concurrency bugs in the Linux kernel but also automates the generation of reproducers. This feature is particularly valuable as it aids in subsequent bug verification and analysis steps.

Concurrency Bug Diagnosing. Diagnosing concurrency bugs is challenging due to the complex thread interleavings. RAProducer [54] addresses this by dissecting the execution trace of a concurrency bug's Proof of Concept (PoC), providing insightful backtracing. ConCrash [10] efficiently navigates the vast space of test codes using advanced search pruning strategies, pinpointing failure-inducing codes. AITIA [21] clarifies kernel race issues by defining and tracing causality chains, illuminating the root causes of concurrency bugs. ERACE [39] combines dynamic and static analysis to streamline the exploration and exploitation of kernel race vulnerabilities, offering a comprehensive approach to diagnosing these complex issues.

9 Conclusion

In our research, we spotlighted *misalignment*, a critical but previously underappreciated factor influencing the triggering of Linux kernel concurrency bugs. This insight reshapes our understanding of concurrency bug triggering, revealing that the intricate interplay between the time window and *misalignment* dictates bug triggering. We encapsulated this concept in the Concurrency Bug Reproduction Model, a predictive tool for assessing the success probability of a race attempt in triggering a bug. Leveraging this model, we introduced CARDSHARK, a strategy to enhance the probability of triggering concurrency bugs by manipulating *misalignment*. We believe that our insights into the concurrency bug-triggering process represent a significant step toward unraveling the unpredictable nature of concurrency bugs. Furthermore, the introduction of CARDSHARK could significantly improve the stability of concurrency bug triggering to ease concurrency bug fixing and analyzing.

10 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd for their insightful comments. This work was supported by the research on vulnerability discovery and impact assessment for desktop operating system from Ascent Project of Institute of Information Engineering, Chinese Academy of Sciences.

References

[1] [CVE-2022-1786] a journey to the dawn | kylebot's blog. https://blog.kylebot.net/2022/10/16/CVE-2022-1786/.

[2] CVE-2022-29582 - computer security and related topics. https://ruia-ruia.github.io/2022/08/05/CVE-2022-29582-io-uring/.

[3] Lexfo's security blog - cve-2017-11176: A step-by-step linux kernel exploitation (part 1/4). https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html.

[4] Linux kernel < 4.10.15 - race condition privilege escalation - linux local exploit. https://www.exploit-db.com/exploits/43345.

[5] NVD - CVE-2016-8655. https://nvd.nist.gov/vuln/detail/CVE-2016-8655.

[6] NVD - CVE-2017-15649. https://nvd.nist.gov/vuln/detail/cve-2017-15649.

[7] syzbot. https://syzkaller.appspot.com/upstream.

[8] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1):1–15, 2017.

[9] Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Effective detection of sleep-in-atomic-context bugs in the linux kernel. *ACM Transactions on Computer Systems (TOCS)*, 36(4):1–30, 2020.

[10] Francesco A Bianchi, Mauro Pezzè, and Valerio Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 705–716, 2017.

[11] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. Pacer: Proportional detection of data races. *ACM Sigplan Notices*, 45(6):255–268, 2010.

[12] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2015.

[13] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review*, 37(5):237–252, 2003.

[14] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective {Data-Race} detection for the kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[15] Google. Syzbot invalid bugs. https://syzkaller.appspot.com/upstream/invalid.

[16] Google. Syzbot open bugs. https://syzkaller.appspot.com/upstream.

[17] Google. Syzkaller. https://github.com/google/syzkaller.

[18] Zunchen Huang, Shengjian Guo, Meng Wu, and Chao Wang. Understanding concurrency vulnerabilities in linux kernel. *arXiv preprint arXiv:2212.05438*, 2022.

[19] Intel. Intel speedstep technology. https://download.intel.com/design/network/papers/30117401.pdf.

[20] Intel. Time stamp counter. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf.

[21] Dae R Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Diagnosing kernel concurrency failures with aitia. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 94–110, 2023.

[22] 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. IEEE, 2019.

[23] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2104–2121. IEEE Computer Society, 2023.

[24] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. Drddr: a lightweight method to detect data races in linux kernel. *The Journal of Supercomputing*, 72:1645–1659, 2016.

[25] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. Pspray: Timing {Side-Channel} based linux kernel heap exploitation technique. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6825–6842, 2023.

[26] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. {ExpRace}: Exploiting kernel races through raising interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2363–2380, 2021.

[27] Linux. CFS. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[28] Linux. commit log 20f2e4c228. https://github.com/torvalds/linux/commit/20f2e4c228c712158113583947f4e16691e951f6.

[29] Linux. commit log 3c4f8333b5. https://github.com/torvalds/linux/commit/3c4f8333b582487a2d1e02171f1465531cde53e3.

[30] Linux. commit log 423ecfea77. https://github.com/torvalds/linux/commit/423ecfea77dda83823c71b0fad1c2ddb2af1e5fc.

[31] Linux. commit log 458c15dfbc. https://github.com/torvalds/linux/commit/458c15dfbce62c35fefd9ca637b20a051309c9f1.

[32] Linux. commit log 4ccf11c4e8. https://github.com/torvalds/linux/commit/4ccf11c4e8a8e051499d53a12f502196c97a758e.

[33] Linux. commit log fd3d91ab1c. https://github.com/torvalds/linux/commit/fd3d91ab1c6ab0628fe642dd570b56302c30a792.

[34] Linux. CPU performance scaling. https://docs.kernel.org/admin-guide/pm/cpufreq.html.

[35] Linux. Linux clock source. https://www.kernel.org/doc/Documentation/timers/timekeeping.txt.

[36] Linux. Linux github repository. https://github.com/torvalds/linux.

[37] Linux. Linux HPET device. https://docs.kernel.org/timers/hpet.html.

[38] Linux. Linux intel_pstate cpu performance scaling driver. https://docs.kernel.org/admin-guide/pm/intel_pstate.html.

[39] Danjun Liu, Pengfei Wang, Xu Zhou, and Baosheng Wang. Erace: Toward facilitating exploit generation for kernel race vulnerabilities. *Applied Sciences*, 12(23):11925, 2022.

[40] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[41] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, 2017.

[42] MITRE. CVE-2017-15265. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15265.

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

[44] MITRE. CVE-2017-26708. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-26708.

[45] MITRE. CVE-2017-7533. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7533.

[46] MITRE. CVE-2021-32606. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-32606.

[47] MITRE. CVE-2022-1729. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-1729.

[48] MITRE. CVE-2023-31083. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-31083.

[49] Gabriel Ryan, Abhishek Shah, Dongdong She, and Suman Jana. Precise detection of kernel data races with probabilistic lockset analysis. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2086–2103. IEEE Computer Society, 2023.

[50] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288, 2009.

[51] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 391–402, 2016.

[52] Jan Wen Voung, Ranjit Jhala, and Sorin 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*, pages 205–214, 2007.

[53] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 661–678. IEEE, 2018.

[54] 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. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 593–606, 2021.

[55] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for {K (H) eaps}: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 71–88, 2022.

11 APPENDIX

11.1 CARDSHARK

The demonstration of CARDSHARK is provided below in pseudo-C code. The *SYNCHRONIZE* primitive refered in Algorithm 1 is achieved by busy waiting on global variables. The *DELAY* primitive relies on continuously checking the time stamp counter within an infinite loop for timing delays. Finally, the *PIN_CPU* functionality is achieved through the *sched_setaffinity* system call to bind a thread to a specific CPU.

```
unsigned long long racer_delay, racee_delay;
unsigned char flag1, flag2, flag3;

static void delay(unsigned long long interval)
{
  unsigned long long end;

  end = rdtsc() + interval;
  while (1) {
    if (rdtsc() > end)
      return;
  }
}

static void pin_cpu(int cpu)
{
  cpu_set_t cset;

  CPU_ZERO(&cset);
  CPU_SET(cpu, &cset);
}

void* thr_racee(void* arg)
{
    flag1 = 1;
    while(!flag3);
    delay(racee_delay);
    sys_racee();
}

void* thr_racer(void* arg)
{
    flag2 = 1;
    while(!flag3);
    delay(racer_delay);
    sys_racer();
}

void race_attempt(void)
{
    pthread_t thrs[2];

    pin_cpu(0);
    pthread_create(&thrs[0], NULL, thr_racee, NULL);
    pin_cpu(1);
    pthread_create(&thrs[1], NULL, thr_racer, NULL);
    pin_cpu(2);
    while (!flag1 || !flag2);
    flag3 = 1;

    pthread_join(thrs[0], NULL);
    pthread_join(thrs[1], NULL);
}

void cardshark(long long align_time)
{
    if (align_time > 0)
        racer_delay = align_time;
    else
        racee_delay = -align_time;

    race_attempt();
}
```

11.2 BLINDSHARK

Below is a sample implementation of BLINDSHARK in pseudo-C code. The *MEASURE* primitive measures execution time by reading the time stamp counter immediately before and after the system call invocation.

```
unsigned long long measure(void (*syscall)())
{
    unsigned long long start, end;
    start = rdtsc();
    syscall();
    end = rdtsc();

    return end-start;
}

void blindshark(void)
{
    unsigned long long racer_len, racee_len;
    long long i;

    racer_len = measure(sys_racer);
    racee_len = measure(sys_racee);

    for (i=0; i<racer_len; i+=100) {
        cardshark(-i);
    }

    for (i=0; i<racee_len; i+=100) {
        cardshark(i);
    }
}
```