RAProducer: Efficiently Diagnose and Reproduce Data Race Bugs for Binaries via Trace Analysis

Ming Yuan
Tsinghua University
Beijing, China
yuanming18@mails.tsinghua.edu.cn

Yeseop Lee
Tsinghua University
Beijing, China
lee.yeb33@gmail.com

Chao Zhang*
BNRist, Tsinghua University, and
Tsinghua University & QI-ANXIN
Group JCNS
Beijing, China
chaoz@tsinghua.edu.cn

Yun Li
Tsinghua University
Beijing, China
liyun19@mails.tsinghua.edu.cn

Yan Cai
SKLCS, Institute of Software, Chinese
Academy of Sciences, and University
of Chinese Academy of Sciences
Beijing, China
ycai.mail@gmail.com

Bodong Zhao
Tsinghua University
Beijing, China
zbd17@mails.tsinghua.edu.cn

ABSTRACT

A growing number of bugs have been reported by vulnerability discovery solutions. Among them, some bugs are hard to diagnose or reproduce, including data race bugs caused by thread interleavings. Few solutions are able to well address this issue, due to the huge space of interleavings to explore. What's worse, in security analysis scenarios, analysts usually have no access to the source code of target programs and have troubles in comprehending them.

In this paper, we propose a general solution RAProducer to efficiently diagnose and reproduce data race bugs, for both user-land binary programs and kernels without source code. The efficiency of RAProducer is achieved by analyzing the execution trace of the given PoC (proof-of-concept) sample to recognize race- and bug-related elements (including locks and shared variables), which greatly facilitate narrowing down the huge search space of data race spots and thread interleavings. We have implemented a prototype of RAProducer and evaluated it on 7 kernel and 10 user-land data race bugs. Results show that RAProducer successfully reproduces all these bugs. More importantly, *it enables us to diagnose 2 extra real-world bugs which were left unconfirmed for a long time.* It is also efficient as it reduces candidate data race spots of each bug to a small set, and narrows down the thread interleaving greatly. RAProducer is also more effective in reproducing real-world data race bugs than other state-of-the-art solutions.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

*corresponding author

ISSTA '21, July 11–17, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8459-9/21/07…$15.00
https://doi.org/10.1145/3460319.3464831

KEYWORDS

Data Race; Binary; Vulnerability; Reproduction

ACM Reference Format:
Ming Yuan, Yeseop Lee, Chao Zhang, Yun Li, Yan Cai, and Bodong Zhao. 2021. 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 (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3460319.3464831

1 INTRODUCTION

As the number of vulnerability discovery solutions (e.g., fuzzing) grows, more and more bugs are reported together with proof-of-concept (PoC) samples. Developers or analysts have to spend lots of time replaying PoCs and debug target programs, in order to understand the root causes of bugs and the way to fix or exploit them. Some PoCs cannot be stably reproduced due to certain randomness factors, e.g., thread interleaving or environment glitches, making many reported bugs hard to diagnose or reproduce. For instance, Syzkaller [52] found a bug but left it unconfirmed[1] for one year.

Motivation: Among these bugs, data race is one of the most common types of bugs that are hard to diagnose or reproduce. A data race occurs if two threads (without proper synchronizations) *concurrently* access a shared data and at least one access is a *write* operation. The study [30] showed that, compared with other types of bugs, data race bugs are harder to diagnose and reproduce, since thread interleavings in multi-threaded programs are non-deterministic [5, 25, 48, 53]. Specifically, three problems need to be addressed.

First, we need to recognize pairs (denoted as *race pairs*) of data access spots which locate in two threads but concurrently access shared data without proper synchronizations, in order to diagnose the root cause of a data race bug. However, a program may have many shared data (assuming the number is `N`) and each could be

[1] https://groups.google.com/g/syzkaller-bugs/c/kZsmxkpq3UI/m/J35PFexWBgAJ

accessed by many threads (assuming the average number of accessing thread is M). Then, the number of potential race pairs is proportional to $N * M^2$, which is large in practice.

Second, for each candidate race pair, we need to explore the thread interleaving space to check whether this candidate pair is the root cause of the race bug, and identify the exact interleaving that triggers the bug, in order to reproduce the bug. But, exploring the interleaving space is an NP-hard problem [38]. Assuming there are K pairs of threads to schedule, then the number of thread interleavings to explore is 2^K, which is too huge to explore.

Third, we need to support binary programs and those run in user or kernel mode, since (1) the source code or compilation configuration of the vulnerable program may be unavailable, or (2) the source code may slightly differ from the binary program due to compilation optimizations. For instance, analysts may find a potential vulnerability (with a PoC) in a commercial software via automated vulnerability discovery solutions, but cannot reproduce this bug (e.g., the PoC is unstable) in some cases. But, they cannot simply request vendors who have source code to analyze the PoC, because in practice vendors do not have resources allocated for unconfirmed bugs. Instead, analysts have to directly analyze binary programs, which is challenging since many information are missing, to confirm and understand the root cause of this bug.

Therefore, it is highly demanded to design an automated data race bug analysis solution, which can efficiently recognize the bug-related race pair and find the correct bug-related thread interleaving, and is applicable to binary programs.

Challenges: Existing data race diagnosis solutions either fail to precisely locate race pairs or fail to efficiently reduce the interleaving exploration space. And no solutions applicable to binary programs are available yet. For instance, SKI [16] randomly schedules memory access instructions rather than identifying exact race spots first, leading to a relatively large interleaving space to explore. SNORLAX [23] recognizes race spots via static points-to analysis, but does not consider the effects of synchronization, resulting in many false positives. To efficiently diagnose and reproduce data race bugs in binaries, three challenges should be addressed:

C1: Recognizing candidate race pairs and reducing the number. Without source code, it is challenging to recognize shared variables in binary programs and candidate race pairs, i.e., data access spots that access the shared variables concurrently. Further, the number of candidate race pairs of a program in general is large as aforementioned. Thus, it is crucial to filter out infeasible race pairs, in order to develop an efficient data race analysis solution.

C2: Recognizing synchronization primitives in binaries. A common way to filter race pairs is synchronization analysis, as data races occur only if no proper synchronizations are applied. One of the most common synchronization primitives is lock. However, there are an unlimited number of implementations of locks. It is infeasible to recognize locks via prior knowledge (e.g., function names), especially when target programs are stripped binaries without symbols.

C3: Reducing and executing the thread interleaving space. For each candidate race pair, we have to explore a thread interleaving space to find the correct interleaving that can trigger the data race bug. Reducing the interleaving space is crucial for the efficiency of data race analysis approaches. Further, given a specific interleaving to schedule, we also need an executor which can faithfully switch threads for both kernel and user-land binary programs as scheduled.

Our solution: In this paper, we propose a general solution RAProducer to solve these challenges and efficiently diagnose and reproduce data race bugs for binary programs. Its efficiency is achieved by a trace-based analysis, which recognizes race- and bug-related elements and removes unrelated data access spots and interleavings. Given a target binary and a PoC sample which has once triggered a data race, RAProducer can efficiently locate the exact race pair and find the exact thread interleaving that triggers the race.

Specifically, RAProducer replays the PoC sample to get the execution trace, and analyzes the trace to precisely recognize shared variables and candidate race pairs, then relies on lock primitives to remove infeasible race pairs that are well synchronized (i.e., Challenge C1). To recognize locks (i.e., Challenge C2), RAProducer utilizes intrinsic characteristics of locks and performs a two-stage static analysis on binaries. Lastly, it applies a new two-dimension strategy to reduce and schedule thread interleavings by prioritizing race pairs that have high impacts on the target program's control flow. RAProducer could follow the scheduled interleaving and faithfully switch threads for both kernel and user-land binary programs via a customized hypervisor in QEMU/KVM and a novel *bp_queue* model (i.e., Challenge C3).

We have implemented a prototype of RAProducer and evaluated it on 7 known kernel data race bugs and 10 user-land bugs. Results showed that, RAProducer is effective, and could successfully reproduce all these bugs. More importantly, *it enables us to diagnose 2 extra real world bugs which are reported as suspicious but left unconfirmed for a long time.* It is also efficient, and reduces candidate data race spots of each bug to a small set, and narrows down the thread interleaving space. We also compared with three state-of-the-art reproduction (detection) tools, and the results show RAProducer is much more effective in reproducing real-world data race bugs than them.

Contribution: In summary, we make the following contributions:

- We propose a two-stage analysis to identify lock functions in binary programs with high accuracy.
- We propose a trace-based and lockset-based approach to recognize shared variables and race pairs, and efficiently remove infeasible ones.
- We propose a two-dimension thread scheduling solution able to prioritize control-flow related race pairs and greatly reduce the interleaving space, and an executor able to faithfully execute the specified thread interleavings for binary programs via a novel *bp_queue model.*
- We implemented a prototype of RAProducer, and successfully diagnosed and reproduced 7 known kernel and 10 user-land data race bugs, with a much smaller search space. We also confirmed and reported 2 unconfirmed kernel data race bugs which can result in UAF, and gained 2 CVEs.

2 BACKGROUND AND RELATED WORK

2.1 an Example of Data Race

We use a real-world vulnerability CVE-2016-8655 in Figure 1 to illustrate the root cause of data races and the complexity of diagnosing and reproducing such bugs.

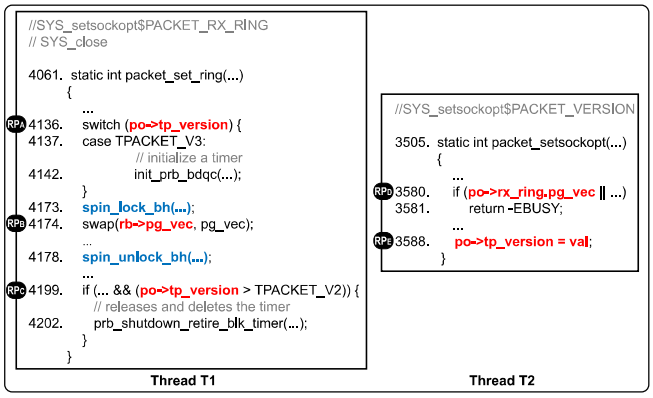

Figure 1: CVE-2016-8655 in Linux 4.4

In Figure 1, three syscalls setsockopt$PACKET_RX_RING, close and setsockopt$PACKET_VERSION on the same socket file run in parallel in two threads; they can result in a UAF crash if scheduled properly. Specifically, if `tp_version` in thread T1 comes to the case of TPACKET_V3 at line 4137, `packet_set_ring` will initialize a `timer_list` object at line 4142. When the socket is closed, thread T1 will check `tp_version` at line 4199, and if its value is great than TPACKET_V2 (which is true for TPACKET_V3), the `timer_list` object will be released. However, if the concurrent thread T2 changes the value `tp_version` to TPACKET_V1 at line 3588 between these two checks, then the `timer_list` object will be initialized but not released and deleted, so the UAF crash will be triggered after the time expires. Besides, if the swap at line 4174 in thread T1 is executed first, then packet_setsockopt in thread T2 will return at line 3581 and the `tp_version` assignment at line 3588 will not be executed. It is obvious that RP_A (at line 4136), RP_C (at line 4199) and RP_E (at line 3588) are race pairs, and RP_B (at line 4174) and RP_D (at line 3580) are race pairs too.

The key to trigger this UAF is to make the program branch into line 4142 but not into line 4202, which is determined by the execution order of RP_A and RP_E, RP_C and RP_E, respectively. Moreover, whether RP_E can be executed depends on the execution order of RP_B and RP_D. Thus, in summary, the order of the race pairs determines whether or not the program can execute as the right control flow that can trigger the crash (line 4142 and line 4202).

To diagnose or reproduce such bugs, we first need to recognize the race pairs (e.g., RP_A - RP_E in this example), and then recognize the locks (e.g., line 4173 and 4178) which may synchronize the execution order of the race pairs, so that we can reduce the search space of race pairs and thread interleavings. After that, we need to recognize the race pairs having influences on the program's control flow (e.g., RP_A, RP_C, and RP_D), and ensure that we will not miss any race pair impacted by other race pairs (e.g., whether RP_E can be executed depends on the execution order of RP_B and RP_D). Lastly, we need to schedule the thread interleavings in an efficient way, to quickly find the interleaving (e.g., RP_D -> RP_B and RP_A -> RP_E -> RP_C) that could trigger the crash.

2.2 Record-and-Replay Systems

A widely used technique of concurrency bug reproduction is deterministic record-and-replay, e.g., [10, 19, 20, 33, 36, 41–43, 45, 50, 61]. Through faithfully recording online runtime information of programs (memory access, instructions, thread interleaving, etc.), such solutions could deterministically replay the execution (including non-deterministic behaviors) to reproduce concurrency bugs.

However, record-and-replay systems have some drawbacks. First, a successful reproduction relies on the prerequisite that the PoC must trigger the bug during recording, which in general is not stable to achieve. Generally, the PoC has to be run for thousands of times to trigger the race. Second, such systems suffer from high runtime performance penalty to support multi-threaded programs in modern multiprocessor architectures. For example, a state-of-the-art technique [45] introduces 2.3× overheads.

In particular, some works record symbolic traces and utilizes model checking or SMT (Satisfiability Modulo Theories) solvers to recover the replay order. Hardware-assisted solutions [26, 27] utilize SMT solvers to replay concurrent programs, which heavily relies on hardware support to suppress the overheads. CLAP [20] and Symbiosis [31] start from a failed execution, use guided symbolic execution to obtain per-thread symbolic traces, and utilize a SMT solver to solve the expected schedule. Based on CLAP and Symbiosis, Cortex [32] explores potential crashing paths and introduces path constraints to the SMT solver, therefore does not need to start from a failed execution. Based on Cortex, ConCrash [3] could start from crashing stack contexts to explore interleavings. However, such SMT-based solutions are usually applicable to a bounded window of events in practice, as larger windows will cause constraint-solving infeasible. Besides, such solutions rely on the delicate and complicated definitions of vulnerability constraints (e.g., out-of-bound and use after free) to guide SMT solvers. As a result, the soundness and completeness of such solutions are highly dependent on the quality of vulnerability modeling (which is discussed in [9, 17]).

Unlike record-and-replay systems, our work focuses on recognizing the exact race pairs and predicting the exact thread interleaving, which makes it possible to successfully reproduce the vulnerability within only a few executions (or even just one in some cases).

2.3 Race Pair Recognition

To diagnose data race bugs, the core task is locating data race spots, which concurrently access a shared variable without proper synchronizations and at least one of them is a write access.

In the past decades, there has been lots of work on data race detection. A large number of solutions, e.g., [4, 9, 15, 18, 44, 55, 59], utilize happens-before analysis to locate race pairs. In general, they try to recognize happens-before relationships between concurrent memory accesses during dynamic analysis. Memory accesses without happens-before relationships are marked as data race spots. Particularly, based on the happens-before analysis, ConVul [9, 35] can identify exchangeable events and is able to detect concurrency memory corruption vulnerabilities. However, they in general have false negatives due to incomplete dynamic testing. For instance, some memory accesses may have happens-before relationship in the current execution trace, but not in another non-explored trace. The data race bug may depend on the memory accesses in the non-explored trace, making the analysis fails.

Another type of popular solutions is lockset analysis, e.g., [12, 39, 46, 47, 51]. Assuming lock is the only synchronization primitive, these approaches in general mark memory access instructions that access a shared variable but are not guarded by the same lock as data race spots. Compared to happens-before approaches, lockset analysis has no false negatives and lower overhead. However, lockset analysis is prone to have false positives, since it does not consider other synchronization primitives (e.g., semaphores, reference counters, etc.) or special use cases where synchronization is not needed.

As it is hard to recognize complex synchronization primitives in large programs, some sampling-based approaches are proposed, e.g., [8, 13, 28, 34], to detect data races instead of inferring happens-before relation or locksets. Specifically, a thread is delayed for a certain duration of time at particular memory accesses, meanwhile, the system monitors whether the same memory is accessed by others during this period. Although this approach will not introduce false positives, it may miss real data race due to incomplete sampling.

Synchronization Primitive Identification. Several studies [21, 49, 54, 56] have been proposed to identify synchronization primitives (e.g., locks) in source code. SyncFinder [56] and BARRIERFINDER [54] use static approaches to identify ad-hoc synchronization primitives. However, these solutions cannot be simply ported to binaries. SyncTester [60] can identify synchronization pairs in user-land binaries through an on-the-fly test; however, it cannot recognize synchronization primitives that are not covered in dynamic testing. Instead of identifying ad-hoc synchronization primitives, we focus on the most common synchronization scheme *locks* in binaries, which facilitate reducing the number of race pairs.

2.4 Thread Scheduling

Identifying data race spots is insufficient to reproduce data race bugs; we have to find out the proper thread interleaving and schedule threads accordingly to reproduce the real race.

2.4.1 Passive Thread Scheduling. Some solutions [23, 24] passively record thread interleaving order during the execution of the target program, and then obtain the correct thread interleaving based on the running results (e.g., crash). However, due to the small race window between race spots, it is hard to execute the target program just once to trigger the crash. These solutions have to execute thousands of times to uncover the correct thread interleaving.

2.4.2 Active Thread Scheduling. On the contrary, active thread scheduling approaches determine thread execution order in advance and schedule threads as planned. In general, they fall into two categories: random scheduling and enumeration-based scheduling. Random scheduling methods [6, 7, 16] schedule threads of the target program in a random order. So, each concurrency vulnerability could be triggered with a certain probability. However, when scheduling complex programs, the space of interleavings to explore increases exponentially with the number of race pairs, and the probability drops dramatically. Enumeration-based scheduling solutions [14, 37, 58, 62] usually collect candidate concurrency instructions, enumerate all orders that match bug patterns (e.g., WR, RW, WRW, etc.), and test each order one by one. However, the efficiency of these methods depends on the number of bug patterns used. More bug patterns lead to a larger interleaving space to explore and higher overheads, while fewer bug patterns lead to more false negatives and a lower possibility to trigger the bug.

2.4.3 Thread Interleavings Execution. To perform active thread scheduling, we need to control the target program's thread execution order according to our schedule. Many prior work [29, 37, 40, 58] implemented its interleaving executor through user-land tools like Intel PIN or JVM, which are not applicable to kernels. SKI [16] uses an adapted virtual machine monitor to determine the execution status of the various threads in kernels. However, it introduces more overheads than hardware-based approaches [13, 22]. In order to effectively execute interleavings in both user mode and kernel mode, we adopt a customized hardware-based method.

3 OUR SOLUTION: RAPRODUCER

3.1 Design Overview

Our goal is to develop a solution able to efficiently diagnose and reproduce data race bugs. As aforementioned, the core challenges are recognizing and reducing candidate race pairs, as well as reducing and scheduling thread interleavings, for binary programs.

Intuitively, we can analyze binary programs via trace analysis, which provides relatively precise information including shared variables that could be accessed concurrently. Then, we recognize locks in binaries by matching several intrinsic characteristics of locks, and use them to find spots that access shared variables but are not synchronized, and mark them as candidate race pairs (denoted as *RootPair*). Lastly, we prioritize RootPairs that may affect the program's control flow (denoted as *CheckPair*), and then explore remaining RootPairs (denoted as *DataPair*), and finally schedule the thread interleavings accordingly to reduce the interleaving space.

We thereby present a novel solution RAProducer to analyze data race bugs. As Figure 2 shows, it consists of three major components: lock function extractor, RootPair extractor, and thread interleaving analyzer and controller.

3.2 Lock Function Identification

As one of the most important synchronization primitives, lock is widely used to quickly and roughly determine the concurrency of two data access spots. However, there are an unlimited number of implementations of locks. It is infeasible to recognize locks via prior knowledge (e.g., function names), especially when target programs are stripped binaries without symbols.

We present a two-stage (i.e., local stage and global stage) static analysis method to recognize locks in binaries.

3.2.1 Observation: Although there are many potential implementations of locks (some of them are wrappers of simple locks), a lock implementation usually has the following intrinsic characteristics:

(1) An atomic instruction (e.g., `cmpxchg`) is involved, to check (and update) the status the lock variable without disturbance.
(2) A loop wraps the atomic instruction and breaks only if the lock is acquired, in order to wait for the lock release.
(3) The functionality of this loop and its parent function is single-purposed, and thus has a small number of instructions.

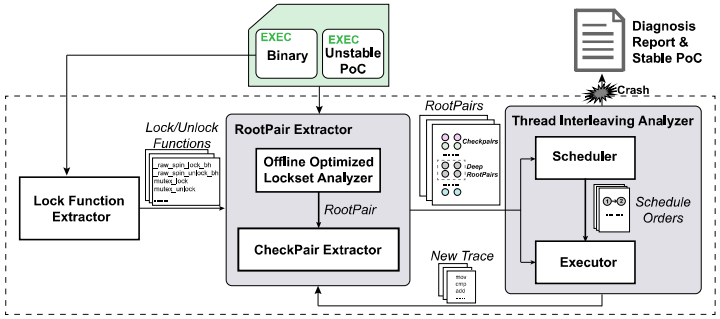

Figure 2: Overall architecture of RAProducer.

Listing 1: A classic lock function

```
1 void lock(void *lockval) {
2   while(!compareAndSet(lockval,0,1)) wait();
3 }
```

Listing 2: An optimistic lock in Linux 4.15

```
1 void msr_event_update(struct perf_event *e)
2 {...
3 again:
4     prev = local64_read(&e->hw.prev_count);
5     now = msr_read_counter(e);
6     if (local64_cmpxchg(&e->hw.prev_count, prev,
              now) != prev)
7         goto again;
8 ...}
```

In addition, in most cases (4) a lock function also has a corresponding unlock function which will release the lock.

Listing 1 shows a classical lock satisfying all 4 characteristics. Data access spots guarded by such a lock function and its counterpart unlock function (with the same lock variable) in general will not cause race. Listing 2 shows an optimistic lock (line 3-7) which does not need an unlock function. Data access spots guarded by such a lock function in general will not cause race either.

Based on these observations, we present a two-stage static analysis to recognize lock functions and optimistic locks in binaries.

3.2.2 Local Stage: In this stage, we utilize the 3 intrinsic characteristics of locks to recognize lock functions and optimistic locks.

As shown in Algorithm 1, we first find all functions for a given binary (line 16) and analyze each loop in each function (line 20). For each loop, if it consists of less than a pre-defined number (i.e., 50) of instructions (line 2); and at least one of its paths contains an atomic instruction (line 4) whose return value acts as the break-out condition of the loop (line 5), we recognize this loop as a candidate *optimistic lock* (line 23), and mark the parent function as a candidate lock function (line 27).

3.2.3 Global Stage: For a given candidate lock function recognized in the local stage, we will apply a global stage analysis to remove false positives and mitigate false negatives, as shown in Algorithm 2.

Note that, a lock function may have wrappers which can be used as locks as well. Thus, to mitigate false negatives, we take wrappers of each candidate lock function into consideration as well (line 20).

As aforementioned, in most cases a lock function has a corresponding unlock function, and they will be invoked together with the same parameter. We could utilize this observation to remove false lock functions. However, the invocation of unlock may not directly exist in the same function which invokes the lock function.

Algorithm 1 Local Stage

```
1: function Check_Lock(loop)
2:     if GetLoopIns(loop) <= 50 then # max number of instructions in a lock loop
3:         for each path ∈ loop do
4:             if HasAtomicIns(path) then
5:                 if AtomicValueForLoop(path) then
6:                     return Ture
7:                 end if
8:             end if
9:         end for
10:    end if
11:    return False
12: end function
13: function Get_Lock_Candidates(binary)
14:    lock_candidate := ∅
15:    opt_lock_candidate := ∅
16:    all_func_set := Get_All_Function(binary)
17:    for each func ∈ all_func_set do
18:        func_loops := Get_Loops(function)
19:        is_lock_func := false
20:        for each loop ∈ func_loops do
21:            if Check_Lock(loop) then
22:                is_lock_func := true
23:                Add loop to opt_lock_candidate
24:            end if
25:        end for
26:        if is_lock_func then
27:            Add func to lock_candidate
28:        end if
29:    end for
30:    return lock_candidate, opt_lock_candidate
31: end function
```

Algorithm 2 Global Stage

```
1: function GetUnlock(lock_func)
2:     unlock_func := 0
3:     xref_funcs := GetXrefFunction(lock_func)
4:     for each func ∈ xref_funcs do
5:         unlock_cand := SameParaCalleeAfterLock(func)
6:         if exist unlock_cand then
7:             pair_number := GetLockPairNumber(lock_func, unlock_cand)
8:             xref_numer := Len(xref_funcs)
9:             if pair_number / xref_numer >= 0.5 then
10:                unlock_func := unlock_cand
11:                break
12:            end if
13:        end if
14:    end for
15:    return unlock_func
16: end function
17: function GetLockFuncs(lock_candidate)
18:    lock_unlock_set := ∅
19:    for each cand ∈ lock_candidate do
20:        lock_and_wrapper_set := GetWrapperFunction(cand)
21:        Add cand to lock_and_wrapper_set
22:        for each lock_func ∈ lock_and_wrapper_set do
23:            unlock_func := GetUnlock(lock_func)
24:            if unlock_func! = 0 then
25:                Add < lock_func, unlock_func > to lock_unlock_set
26:            end if
27:        end for
28:    end for
29:    return lock_unlock_set
30: end function
```

Thus, for each candidate lock, we apply a probability-based analysis to recognize its corresponding unlock function (line 23).

Specifically, we first find parent functions which have invoked the candidate lock (line 3), and locate candidate unlock functions that have the same parameters as the lock and are invoked after the lock in the same parent functions (line 5). Then, we calculate how many times the candidate unlock is invoked together with the lock in the whole program (line 7), and mark it as a real unlock function if they are invoked together more frequently than not together (line 9). Further, this unlock function is grouped together with the lock function as a set of locks (line 25).

3.3 RootPair Recognition

After recognizing locks in binaries, RAProducer performs an optimized trace-based lockset analysis to recognize candidate race pairs (i.e., RootPairs). Different from traditional lockset analysis which operates on a whole program (with source code), our lockset analysis for binaries works on execution traces, which has better accuracy and can be optimized in several ways.

3.3.1 Optimized Trace-based Lockset Analysis. We perform lockset analysis on execution traces. Specifically, we run the PoC in the record-and-replay platform Panda [11] to record its execution trace, and then perform an inter-procedural and flow-sensitive lockset analysis in the replay stage.

First, given the execution trace, we could recognize the exact memory being accessed by each instruction, and thus find out all shared variables in the trace. It thus has much better accuracy than static solutions, since the latter relies on alias analysis and call graph analysis which are inaccurate.

Further, we recognize data access spots that may access shared variables concurrently from the trace, and utilize lock functions that we have recognized to remove well-synchronized data access spots. Notably, unlike traditional lockset analysis, we also remove data access spots protected by *optimistic locks* that we have recognized along with traditional lock functions (e.g., pessimistic locks), thus reducing false positives.

Lastly, we apply another scope optimization to further reduce the analysis scope. Specifically, we only focus on data access instructions between thread start and thread exit, but not instructions of thread start and exit itself. This is because the data race spots usually appear inside the function of the new thread, not during the thread creation or exit phases.

3.3.2 CheckPair Recognition. In addition to RootPairs, we also try to recognize a subset of race pairs which affect the program's control flow, i.e., CheckPairs. Such pairs are important for thread interleavings, since the target data race bug/crash has to follow certain program paths which may be affected by CheckPairs. Thus, the thread scheduler should explore interleavings related to CheckPairs first to efficiently explore the interleaving space.

Specifically, we examine each RootPair to verify whether it is a CheckPair. First, if one instruction in the RootPair is a branch instruction (such as `CMP`, `TEST`, etc.), then it is a CheckPair. Second, if a branch instruction is affected by the data flow from one instruction in the RootPair, then this RootPair is also a CheckPair. An intra-procedural data flow analysis will be applied in the second case. The pair (R_1, W_1) in Figure 3 is an example CheckPair.

3.3.3 Missing Deep RootPairs. Due to the non-deterministic behaviors of multi-threaded programs, our trace-based analysis may miss the real race pairs. For instance, the RootPair (W_1, R_2) in Figure 3 depends on the first RootPair (R_1, W_1). It is possible that W1 is executed before R1 sometimes, then the pair (W_1, R_2) will not exist in the corresponding traces. For simplicity, we denote a RootPair that depends on the interleaving order of another RootPair as a *Deep RootPair*. We rely on the thread interleaving scheduler to uncover such Deep RootPairs. More details will be discussed in the following section.

3.4 Thread Scheduling and Execution

After obtaining RootPairs, we further try to determine thread interleavings that can trigger target race bugs, and execute the thread interleavings as planned to verify and reproduce target bugs.

3.4.1 Two-dimensional Thread Scheduling. Note that, the crash caused by the data race PoC sample depends on a specific control flow and data flow, which may be affected by some race pairs. Thus, we adopt active thread interleaving and prioritize RootPairs that affect the program's control flow or data flow (i.e., CheckPair and DataPair respectively), which turns to be very effective.

Control-flow Dimensional Scheduling: First, we prioritize interleavings related to CheckPairs which affect programs' control flow. Specifically, for each CheckPair <I1, I2>, RAProducer schedules thread interleavings of two orders (i.e., I1->I2 and I2->I1) respectively, and checks whether a crash is triggered. If yes, the bug is successfully reproduced and the analysis will stop. Otherwise, the analysis continues and other interleavings will be explored.

Deep RootPair Discovery and Scheduling: During exploring the control-flow dimensional scheduling, we also check whether a new RootPair is found in the new trace, in addition to checking crashes. If yes, the new RootPair is denoted as a Deep RootPair. After exploring all interleavings related to CheckPairs, we will get a list of Deep RootPairs. Then, we will schedule thread interleavings related to these Deep RootPairs, to check crashes. Note that, when scheduling a Deep RootPair, the interleaving of its dependent CheckPair will be kept so that the Deep RootPair shows in the new interleaving. If no crashes are triggered, then the analysis continues and other interleavings will be explored.

Data-flow Dimensional Scheduling: After Deep RootPairs are explored, RAProducer will schedule DataPairs which might change data flow. For example, the execution order of (R_2, W_1) in Figure 3 has no effect on branches but on the value of `len`. We will traverse candidate thread interleavings for each DataPair. If a crash is triggered, the exploration stops. Otherwise, the analysis continues.

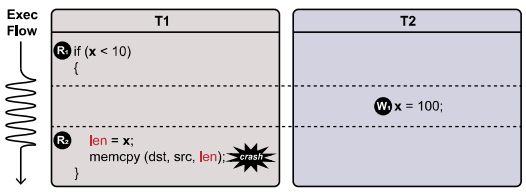

Figure 3: An example of Deep RootPair and CheckPair

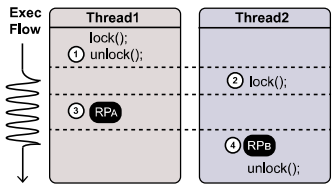

Figure 4: An example RootPair affected by lock order.

Multi-pair Scheduling: Finally, if neither of the above steps can trigger a crash, the data race bug is likely to depend on the order of multiple race pairs. Thus, RAProducer will schedule multiple RootPairs at the same time. Specifically, RAProducer first performs multi-pair scheduling on CheckPairs, by randomly sampling an incremental number of CheckPairs and combining different scheduling orders of every sampled CheckPair. If no crash is triggered, then RAProducer performs multi-pair scheduling on CheckPairs and DataPairs together, until a crash is found.

RAProducer uses hardware breakpoints (by modifying QEMU and KVM) to schedule each RootPair. To determine the runtime concurrency between each pair of instructions to be scheduled, RAProducer adopts the breakpoint-wait scheme, a widely used methodology in data race detection (such as Datacollider, RAZZER, etc.). Concretely, if a pair of instructions can pause at breakpoints at the same time, then we can schedule them. Otherwise, we can view them guaranteed by other synchronization primitives (e.g., completions) instead of lock that they will not be executed concurrently, so we will no longer schedule them.

Lock Scheduling: In practice, there are some corner cases which would make the aforementioned thread scheduling infeasible to achieve. For instance, as shown in Figure 4, assuming we are planning to schedule ③->④, then we must follow the order ①->② first. Otherwise, Thread2 acquires the lock before Thread1 acquires and releases it, then the data access spot RP_A will never execute before RP_B, i.e., ③->④ cannot be scheduled. Therefore, we will add the order ①->② together with the order ③->④ to the scheduling queue.

Thus, when we explore the interleaving RP_A->RP_B of a RootPair, if there is a lock on the RP_B but not the same lock on RP_A, we will check the trace of Thread1 to see whether there is a lock acquiring and releasing operation for the same lock before executing RP_A. If so, the lock of the two threads will also be scheduled together with the expected interleaving of the RootPair.

3.4.2 Thread Interleavings Execution. For each candidate scheduling order, RAProducer lastly faithfully executes the thread interleavings to verify the target data race bugs. To control the execution of thread interleavings, we implement a hypervisor in QEMU and KVM, which takes charge of installing hardware breakpoints in the guest OS and resuming execution after breakpoints are triggered. Hardware breakpoints will be inserted at RootPairs and get resumed in the specified scheduling order.

But the straightforward hardware breakpoint solution cannot control thread interleavings as we wish. A major drawback of the hardware breakpoint mechanism is that it can only set 4 breakpoints at most, and for each CPU (or vCPU) it cannot ensure that the breakpoints are triggered in the specified order.

To overcome these shortcomings, we designed a novel *bp_queue* (breakpoint priority queue) scheme to assist thread scheduling. In this model, each processor (vCPU) has its own bp_queue consisting of a set of schedule points, and each schedule point consists of the address and execution order of an instruction. The bp_queue works as follows:

(1) Only the breakpoint at the head of the bp_queue could be inserted and triggered.
(2) When a breakpoint is triggered, only the breakpoint with the minimal order among the heads of all bp_queues will be resumed and dequeued.

With the bp_queue model, we can handle multiple breakpoints imposed by multiple RootPairs. To sum up, RAProducer controls the complex thread interleavings by the following two steps:

Preparation bp_queue for each thread: RAProducer hooks the thread-creation function (e.g. `pthread_create`) to prepare bp_queue information. When starting a new thread, RAProducer calls a customized transition function then resumes execution of the original start_routine function. The transition function first sets the CPU affinity mask of each thread, and then sets up the bp_queue information for every processor. With the aid of hardware breakpoints, we implement a general hypercall to interact with the hypervisor. Compared with RAZZER [22], which modifies kernel code to implement a hypercall syscall handler, RAProducer is much more portable and flexible as it places the hypercall function in user-land without making any modification to the target kernel.

Dynamically adjust breakpoints according to scheduling order: After the original start_routine function starts, the hypervisor inserts the breakpoint at the head address (i.e., the instruction address stored in the head schedule point) of the bp_queue. If the breakpoint is triggered, the hypervisor will check whether its order is the minimal one among the heads of all vCPUs' bp_queues. If not, the thread will get blocked until its order becomes the minimal one. Otherwise the hypervisor will perform the following steps: (1) remove the breakpoint from the processor, (2) dequeue the breakpoint from the bp_queue, (3) insert a new breakpoint at the head address to the processor and resume execution. With the above steps, the hypervisor can complete thread scheduling accurately as planned, even though multiple race pairs are scheduled at the same time, and thus excels existing solutions.

Figure 5 shows an example of bp_queue. RAProducer first sets breakpoints at I1 and I2, and executes the target program. Firstly, the breakpoint at I2 is triggered, but it has no priority to be scheduled since its order number isn't minimal in the heads of two bp_queues, so it will wait for the breakpoint at I1. Once I1 is triggered and RAProducer finds it has the minimum order number, the tool will perform a single step execution at I1, removes the breakpoint at I1, dequeues I1 from bp_queue1 and sets a breakpoint at I4. After that I2's order number becomes the minimal one in the two queue heads, so RAProducer repeats the above procedure to I2. Then all vCPUs resume execution, and handle I3/I4 like I1/I2. Finally, RAProducer completes scheduling when all bp_queues get empty.

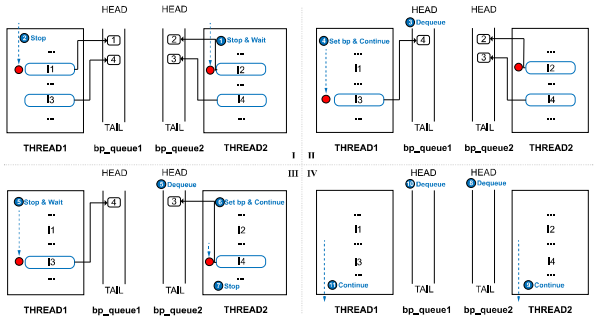

Figure 5: A simplified bp_queue example. There are two bp_queues and each has two schedule points. Numbers in the box are the scheduling order; red and blue solid dots represent breakpoints and scheduling actions respectively.

4 EVALUATION

4.1 Implementation Details

We implemented a prototype of RAProducer with 2773 lines of Python code and 2218 lines of C code, and we will release our source code.

Lock Functions Extractor. To identify lock/unlock functions through local-global strategy, RAProducer needs control flow and data dependency information, which can be obtained by IDA Pro [1]. With the obtained information, we implemented an IDA plugin as Lock Functions Extractor by IDA Python.

RootPair Extractor. In this module, RAProducer uses Panda to record and replay the target binary. Panda can record all the executed instructions and data in the guest OS, so we implemented replay plugins to get control-flow and data-flow information in the replay stage. Then RAProducer performs lockset-based analysis with this information to get RootPairs. We implemented an IDA plugin to disassemble the instructions and perform intra-procedural data-flow analysis from RootPairs to condition instructions to get CheckPairs. Finally, RAProducer extracts RootPairs and CheckPairs, and stores them in JSON format.

Thread Interleaving Analyzer. Given the RootPairs described with JSON, we implemented a two-dimension active scheduling algorithm (Section 3.4) in Scheduler to generate scheduling order. Then we implemented a bp_queue scheme in hypervisor to handle multiple RootPairs scheduling. Once triggering the crash, RAProducer generates the diagnosis and a reproduction report.

4.2 Experimental Setup

We further evaluate RAProducer to answer the following questions:

(1) How effective is it at extracting lock functions? (Section 4.3)
(2) How effective is it at extracting RootPairs? (Section 4.4)
(3) How effective is it at uncovering thread interleaving that triggers a race? (Section 4.5)
(4) How effective is it at diagnosing and reproducing real world known and unconfirmed data race, comparing to state-of-the-art solutions? (Section 4.6 and 4.7)

To answer the above questions, we have evaluated RAProducer on user-land programs and kernels respectively. We searched for data race bugs that can cause memory corruption in MITRE CVE/oss-security/exploit-db/ctftime, and removed those either (1) of other race condition types (i.e., atomicity violation [40]), or (2) without public PoC. It costs us 95 man-hours to process all the data and yields 17 bugs, including 7 real-world CVEs in different-version kernels, and 3 real-world data race bugs in two user-land pervasive applications Apache and MySQL, and 7 user-land Pwn challenges in CTF (Capture The Flag) competitions.

All the experiments are conducted on a x86-64 PC with 8 Intel-i9-9900@4.0G processors and 16-GB memory.

Table 1: Results of extracting lock function

Lock functions	Linux 4.4	Linux 4.10	Linux 4.15	Apache 2.0.48	Glibc 2.23
Extracted	33	26	28	9	8
Real	28	22	25	7	6

4.3 Extracting Lock Functions

4.3.1 Experiment of Lock Extraction. RAProducer uses the local-global approach to extract lock/unlock functions in the target program. We tested the approach on Linux 4.4, Linux 4.10, Linux 4.15, Apache 2.0 and Glibc 2.23. The results are shown in Table 1.

RAProducer can identify most of the commonly used primitive lock functions (such as *spin_lock*, *mutex_lock*, *upwrite* and etc), accounting for nearly all lock functions in the execution traces of the evaluated programs. In fact, most lock functions rely on these primitive lock functions. For example, *bh_lock_sock* is actually a *spin_lock*, and *tty_lock* calls *mutex_lock* to construct a lock. Hence, in the evaluated binary, we could skip *bh_lock_sock* and *tty_lock*, and use *spin_lock* or *mutex_lock* to perform lockset analysis instead.

We compiled a kernel (Linux 4.10.1) with symbols; based on the symbols and our experience, we manually identified and verified all the lock functions as ground-truth to check the effectiveness of RAProducer. In addition, to demonstrate its effectiveness more straightforwardly, we only focus on unique locks and neglect lock functions that rely on other lock functions (such as the *tty_lock* mentioned earlier). The result is in Table 4 in Appendix. It shows that the local-global approach is effective at extracting lock functions and could be used for lockset analysis. And we will discuss the false positives and negatives of locks extraction in Section 4.3.2.

In the following sections we will discuss several types of recognized locking functions.

Typical lock function. A typical lock function is one that satisfies the above requirements in the local stage, and in most cases is followed by another function with the same arguments (that is, the unlock function). Taking the *__lll_lock_wait_private* in Glibc 2.23 as an example, Listing 3 shows its source code.

Listing 3: __lll_lock_wait_private in Glibc 2.23

```
void __lll_lock_wait_private (int *futex)
{
  do{
      int oldval = atomic_compare_and_exchange_val_24_acq (
          futex, 2, 1);
      if (oldval != 0)
          lll_futex_wait (futex, 2, LLL_PRIVATE);
  }while (atomic_compare_and_exchange_val_24_acq (futex, 2, 0)
        != 0);
}
```

Listing 4: spin_lock in assembly code

```
1  _raw_spin_lock
2      push    rbp
3      mov     rbp, rsp
4      xor     eax, eax
5      mov     edx, 1
6      lock cmpxchg [lock], edx
7      test    eax, eax
8      jnz     label1
9      pop     rbp
10     ret
11 label1:
12     mov     esi, eax
13     call    queued_spin_lock_slowpath
14     pop     rbp
15     ret
```

The break of the loop (line3 - line9) depends on the result of the atomic exchange of the lock variable *futex* in line 9. Obviously, it meets the requirements of the local stage. Furthermore, according to our statistics, more than a half of the invoked *__lll_lock_wait_private* appear prior to the unlock function with the same parameters. So it also satisfies the conditions of the global stage. As a result, RAProducer identifies it as a lock function.

In fact, as an ad-hoc lock function in GNU C library, *__lll_lock_wait_private* is widely invoked in standard library functions (e.g., *malloc*, *free*, *gets*, and so on). For prior lockset analysis work which only focuses on standard APIs (e.g., *pthread_mutex_lock*), ignoring *__lll_lock_wait_private* inevitably introduces lots of unnecessary false positives.

Wrapper lock function. The global stage not only prunes fake lock functions, but also identifies more real lock functions. Taking the *spin_lock* in Linux 4.15 as an example, its assembly code is shown in Listing 4. In addition to identifying *queued_spin_lock_slowpath* in the local stage, we also identify its wrapper function *spin_lock* (denoted as wrapper lock function) in the global stage. Although *spin_lock* mainly relies on *queued_spin_lock_slowpath* to implement its lock, the inline function *do_raw_spin_trylock* (line 5-6 in the assembly code) is invoked before *queued_spin_lock_slowpath*. As shown in line 6, it tries modifying the lock variable from 0 to 1 directly, and the change of the lock variable indicates the acquirement of the lock. So in the subsequent dynamic trace recording, when entering the *spin_lock* function, if the lock variable is updated successfully, the function will exit directly because the register eax in the test instruction (line 7) is zero, and the call of *queued_spin_lock_slowpath* (line 13) will not be recorded. Hence, to effectively utilize the identified lock functions to perform lockset analysis, it is necessary to extract the *raw_spin_lock* function, not just the *queued_spin_lock_slowpath*. Table 4 shows that our strategy works very well for identifying this kind of lock functions.

Ad-hoc lock function. Prior work [56, 60] shows that ad-hoc lock functions can be harmful. However, the functions are usually not described in documents, and thus are difficult to find. The majority of false positives in lockset analysis results from lacking analysis of ad-hoc lock function's effect on synchronization. As one of our contributions, RAProducer can identify two types of ad-hoc functions: (1) the ones relying on basic lock functions to implement its feature, such as *tty_lock* or *bh_lock_sock* mentioned above; (2) the ones satisfying the requirements of local and global stage policies, such as *__lll_lock_wait_private* in Glibc or *tty_ldisc_lock* in kernel.

Table 2: Results of RootPairs

		AccessPair	AccessPair$_{lock}$	RootPair	CheckPair
Kernel	CVE-2018-12232	5485	1423	387	39
	CVE-2017-10661	4689	1323	331	16
	CVE-2017-17712	5038	1536	328	5
	CVE-2016-8655	4282	1024	81	6
	CVE-2017-2636	1947	473	69	3
	CVE-2017-15265	5097	1131	550	25
	CVE-2014-0196	2202	847	146	9
User	Apache-21287	124	96	85	13
	Apache-25520	115	37	27	6
	MySQL-3596	430	119	50	21
	GOT(icq)	45	42	4	4
	zero_task(TCTF2019)	25	15	7	4
	unknow(BYTECTF2019)	15	15	6	2
	race(RHG)	113	37	5	1
	mulnote(BYTECTF2019)	15	10	3	1
	MultiHeap(TWCTF2019)	12	6	4	0
	note_sys(WHCTF2017)	10	10	10	4

4.3.2 Discussion of False Positive and Negative. The false negatives shown in Table 4 in the Appendix are mostly caused by insignificant invocation counts in the global analysis stage, i.e., they are invoked less than 3 times in the trace. In this case, it is difficult to use statistics to find associated lock/unlock pairs. Fortunately, such missing locks are rarely called in the trace and thus have low impacts on the lockset analysis too. Besides, we also cannot handle some special lock-acquiring functions that do not have any arguments, such as *rcu_read_lock* or *console_lock*. Despite this small number of false negatives, RAProducer is able to recognize most lock functions and reliably facilitate the lockset analysis to greatly reduce the number of RootPairs, as shown in Table 2. Thus, the false negative issue is negligible, and we leave it for future work to recognize more lock functions.

However, some non-lock functions also satisfy the requirements of local and global stage policies, which may introduce false positives. As Table 1 shows, the number of false positives is very small, and it's almost impossible to trigger calls to these functions (meaning they don't appear in trace), so this is acceptable for us, too.

4.4 Extracting RootPairs

We evaluated multiple user-land and kernel programs to extract RootPairs and Checkpairs, and the results are shown in Table 2. In this table, **AccessPair** denotes the number of instruction pairs accessing the same memory extracted from the trace; **AccessPair$_{lock}$** denotes the result with regular lockset analysis (without optimization) while **RootPair** denotes the number of RootPairs extracted by our optimized lockset analysis. **CheckPair** denotes the number of CheckPairs extracted from RootPairs. It is obvious that, through offline optimized lockset-based analysis RAProducer can greatly reduce the scope of candidate race pairs and effectively find the RootPairs and CheckPairs causing the data race.

As Table 2 shows, lockset analysis on binary can greatly reduce the number of pairs to be scheduled (the fourth column), and our optimized lockset analysis can further reduce more RootPairs by a great margin (the fifth column).

Binary-based dynamic analysis like the breakpoint-wait scheme can also determine whether a pair of instructions is concurrent, and as mentioned in Section 3.4.1, RAProducer also utilizes breakpoint-wait scheme to confirm runtime concurrency before scheduling

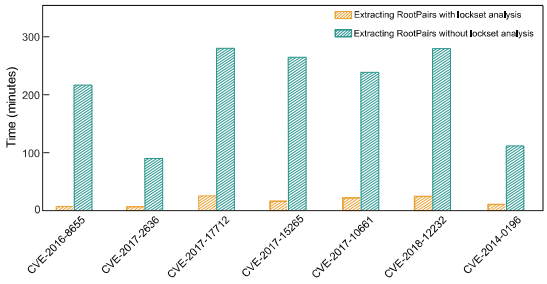

Figure 6: Contribution of lockset analysis. The vertical axis represents the time of extracting all scheduled pairs

RootPairs. However, the overhead of RAProducer with lockset analysis is much lower than that of just using the breakpoint-wait scheme (such as Datacollider). To better understand the value of optimized lockset analysis, we have implemented another tool that dropping the optimized lockset analysis to extract RootPairs. In other words, it only uses breakpoint-wait scheme to detect the concurrency of instruction pairs. We compare this tool with full-fledged RAProducer on extracting RootPairs of kernel CVEs. As shown in Figure 6, RAProducer has 11× to 30× performance improvement over the tools only using dynamic analysis to detect concurrency in extracting RootPairs.

CheckPair. We can also learn from Table 2 that the CheckPairs only account for a small portion of the RootPairs, which means that there are few concurrent instruction pairs that really affect the control flow of the program. Therefore, instead of scheduling all the RootPairs without distinction, we should identify and prioritize these CheckPairs during scheduling. In fact, the subsequent scheduling experiments clearly demonstrate the effectiveness of prioritizing CheckPairs, especially for those complex programs. Among the ten common software vulnerabilities (including seven kernel vulnerabilities, two Apache vulnerabilities, and one MySQL vulnerability), seven can trigger a crash simply by scheduling their CheckPairs (and the Deep RootPairs they generate, if any).

Deep RootPair. RAProducer can effectively handle the situation of Deep RootPair as well. Taking CVE-2017-10661 as an example, we show its RacePairs in Figure 7. To trigger the crash, the scheduling order should be RP_A-> RP_B-> RP_C-> RP_D. If RP_B is executed before RP_A when running the PoC, the RP_C will not be recorded by Panda, so we only have RP_A, RP_B and RP_D in the trace log. After performing scheduling with these three instructions, we find that there is no scheduling order that is able to trigger crashes stably. Therefore RAProducer performs Deep RootPair analysis, and eventually finds instruction RP_C by arranging for RP_A to execute prior to RP_B. This ensures that stable reproducible thread interleaving can be uncovered in the subsequent analysis.

4.5 Scheduling Threads

Since we accurately obtained RootPairs and CheckPairs in the previous steps, the thread interleavings space can be greatly reduced. To better illustrate the advantage of RAProducer on thread scheduling in binaries, we compare it to SKI, another binary-based tool

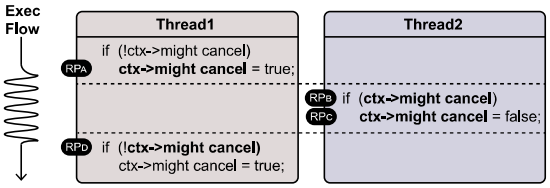

Figure 7: CVE-2017-10661 in Linux 4.10.1

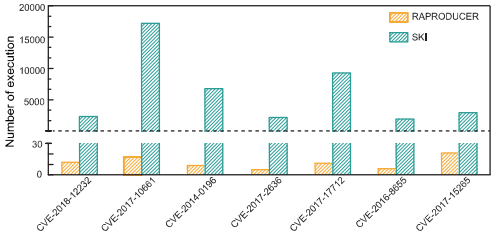

Figure 8: Efficiency of SKI and RAProducer in uncovering thread interleaving that triggers a race.

applying PCT algorithm to scheduling. SKI works on the kernel, so we test it against RAProducer on seven kernel CVEs.

However, we do not have access to the full source code of SKI, so we implemented SKI_{Emu} by extending RAProducer thread interleaving analyzer with the idea of random thread interleaving in PCT algorithm. Instead of performing thread interleavings between RootPairs, PCT algorithm randomly selects the reschedule points from executed instructions. Particularly, SKI has optimized the selection range. It randomly selects reschedule points from instruction pairs that access the same address (denoted as communication points). Corresponding to that, SKI_{Emu} randomly selects reschedule points from the data in the third row of Table 2, and interleaves the execution when CPU reaches the reschedule points.

We use times of executions to trigger data race during scheduling to evaluate the efficiency of RAProducer and SKI_{Emu}. As shown in Figure 8, compared to SKI, RAProducer requires 191× to 750× fewer executions to successfully trigger data race.

4.6 Analyzing Known Data Race Bugs

We test 10 real-world programs and 7 CTF challenges to evaluate the effectiveness of RAProducer; all of them are successfully diagnosed and reproduced, as shown in Table 3. On the one hand, without any preliminary knowledge about the target programs or PoC, RAProducer is able to precisely extract all the RootPairs causing the races, and identify the thread interleavings that triggered the crashes. These diagnoses help security researchers understand the cause of the vulnerabilities and guide them to patch or experimentally attack the programs. On the other hand, RAProducer can steadily reproduce all of these data race bugs, which powerfully demonstrates its effectiveness and compatibility with multiple kernels and user-land programs.

We further compare RAProducer to several state-of-the-art solutions, including ConVul, Cortex and ConCrash. These tools all rely on an explicit (and manually provided) model of program

Table 3: Results of Reproduction (or Detection)

	Bug Name	Vulnerability Type	RAProducer	ConVul	Cortex
Real World	CVE-2016-8655	Use After Free	✓	×	×
	CVE-2017-2636	Double Free	✓	×	✓
	CVE-2017-15265	Double Free	✓	✓	×
	CVE-2017-17712	Uninitialized Use	✓	×	×
	CVE-2014-0196	Buffer Overflow	✓	×	×
	CVE-2017-10661	Use After Free	✓	✓	×
	CVE-2018-12232	NULL Pointer Deref	✓	✓	×
	Apache-21287	Double Free	✓	✓	✓
	Apache-25520	Buffer Overflow	✓	×	✓
	MySQL-3596	NULL Pointer Deref	✓	✓	✓
CTF Challenge	GOT	Use After Free	✓	✓	✓
	zero_task	Use After Free	✓	✓	✓
	unknow	Use After Free	✓	✓	✓
	race	Use After Free	✓	✓	✓
	mulnote	Use After Free	✓	✓	✓
	Multi Heap	Use After Free	✓	✓	✓
	note_sys	Double Free	✓	✓	✓

semantics, e.g., synchronization primitives and callsites of the free/assert functions, while RAProducer does not need this information. In order to compare these tools with ours, we overcome their limitation manually and provide this semantics information for them.

As for ConCrash, it relies on Cortex to explore interleavings. In our scenario, the input PoC is given, and we only care about the thread interleavings that trigger the crash. Thus, it is sufficient to compare with Cortex rather than directly comparing to ConCrash. As for Cortex, it targets at assertions (which in general do not exist in programs) rather than general memory corruption vulnerabilities. In order to better measure Cortex's reproduction ability, we provide it with data sets where the memory vulnerabilities are simplified as assertions. As a result, this simplification relatively enhances the ability of Cortex because modeling an assertion is much easier than modeling a memory corruption vulnerability into SMT constraints (that is a prerequisite of the original Cortex as discussed in the Section 2).

For each PoC test case, we ran RAProducer, Cortex and ConVul for 24 hours, and the bug reproduction results are shown in the fourth, fifth, and sixth columns of Table 3, where "✓" indicates a successful reproduction or detection, and "×" means a failed one. Although both ConVul and Cortex perform well as RAProducer does on simple CTF challenges, these two tools have trouble handling real-world programs. ConVul can only detect three specific vulnerability types (UAF/DF/NPD), while RAProducer is designed to reproduce (and detect) all concurrency bugs that once resulted in crashes. However, we can see that even for UAF vulnerabilities (such as CVE-2016-8655 and CVE-2017-2636), ConVul cannot detect them because of its poor scheduling implementation (focusing primarily on scheduling between synchronizations rather than that between read/write). Further, even though we have lowered the difficulty by providing assertions, Cortex still fails to reproduce many bugs, because it limits the detection into a bounded window of events to avoid heavy constraint-solving costs. In our test case, we observed that when the involved events (read/write, lock) exceed 10,000, Cortex cannot reproduce after 24 hours.

4.7 Analyzing Unconfirmed Data Race Bugs

We also applied RAProducer to two data race vulnerabilities which were reported as suspicious but left unconfirmed for a long time, named CVE-2020-25668 and CVE-2020-25656. RAProducer successfully diagnosed and reproduced these two bugs, which were fixed after being reported to the maintainers. One of the two vulnerabilities was found by syzbot [2], a famous public fuzzer, and the crash report was published for nearly one year. However, due to the difficulty of diagnosis and reproduction of the data race, it has not been confirmed and fixed until we report it to the maintainers. Here we present a case study to explain the complexity of reproduction.

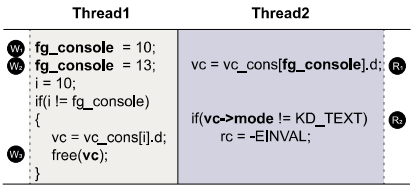

Figure 9: Case study: CVE-2020-25668

Case Study on CVE-2020-25668. Figure 9 shows a simplified version of this bug. This is a concurrency use-after-free vulnerability in vt_console driver of linux kernel, and the race is on the shared variable `fg_console` and `vc`. To trigger the UAF, we need to ensure that `fg_console` equals to 10 when Thread2 reads its value at R_1, and that the free operation on `vc` is executed before the read operation at R_2. After analysis, RAProducer is able to find that the order W_1->R_1->W_2 and W_3->R_2 can trigger a crash successfully. By contrast, this is intractable for SKI since it adopts the PCT algorithm for scheduling. The more reschedule points the PCT algorithm has, the larger its scheduling space will be. The original SKI is implemented with only two reschedule points [16], but this bug requires three, so it cannot be reproduced successfully by SKI. Even if we adjusted the rescheduling points to three, SKI still failed to reproduce the bug after running for 24 hours.

5 DISCUSSION

5.1 False Positive and Negative Analysis

5.1.1 False Positive by Benign Data Race. Some concurrent memory accesses are benign. For example, the reference count variable is

allowed to be updated concurrently. We notice that quite a few benign races occur in optimistic locks, and RAProducer's optimistic lock optimization can prune some of these false positives. In fact, most of the false positives in the fifth column of Table 2 are benign data races, and the number of them is acceptable to us.

5.1.2 False Negative by Deep RootPair Extraction. RAProducer can extract Deep RootPairs caused by the scheduling order of RootPairs. However, there are other factors affecting the extraction, such as the order of synchronization primitives. In fact, this involves concurrency bugs such as order violations or atomic violations; since RAProducer only targets data race, we leave this for future work.

5.2 Lock Functions Extraction

We notice that some unlock functions are inlined, whose invocations are translated to one or more instructions in the target binary. In this case, instead of identifying unlock functions in the binary, RAProducer identifies unlock instructions in the trace obtained by running the PoC in Panda. Specifically, when a lock function in this trace is called and recorded, RAProducer sets its parameters as lock variables, and then analyzes the subsequent instructions following this call. Intuitively, we take the first instruction modifying the lock variable as the unlock instruction. In this case, the scope of the acquired lock starts from the instruction calling the lock function and ends with the unlock instruction. The results of lockset analysis indicate that we can well handle this case.

5.3 Scalability

RAProducer mainly focuses on diagnosis and reproduction of data race that can result in memory corruption, but could be easily extended to a general data race detector, as it can identify race pairs in RootPair recognition stage. Besides, given a PoC, RAProducer can find out the thread scheduling that can trigger a crash in the target binary, but it cannot generate a PoC. In the future, we can use methods like directed fuzz [57], combined with crash and patch reports to automatically generate a multi-threaded PoC, and apply RAProducer to schedule it to trigger the crash stably.

6 CONCLUSION

In this paper, we propose a practical approach RAProducer to effectively diagnose and reproduce data race bugs in user-land or kernel binaries. RAProducer utilizes three key techniques: (1) a two-stage strategy to identify lock functions in binaries; (2) an optimized trace-based lockset approach to extract RootPairs precisely and missing Deep RootPairs; (3) a two-dimension active thread scheduling strategy to uncover the correct thread interleavings, and an effective bp_queue model to control thread interleavings execution. The evaluation of RAProducer on 10 user-land programs and 7 kernel bugs showed that, this approach is both effective and efficient at diagnosing and reproducing known bugs. We also showed that RAProducer could facilitate us diagnosing and reproducing unconfirmed data race bugs.

ACKNOWLEDGEMENT

This work was supported in part by National Natural Science Foundation of China under Grant U1736209, 61972224 and 61772308, BNRist Network and Software Security Research Program under Grant BNR2019TD01004 and BNR2019RC01009, and the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006).

A APPENDIX

Table 4: Identification of lock functions in Linux 4.10.1

Function	Identification
_raw_spin_lock	✓
_raw_spin_lock_irq	✓
_raw_spin_lock_bh	✓
_raw_spin_lock_irqsave	✓
_raw_write_lock	✓
_raw_write_lock_bh	✓
mutex_lock	✓
mutex_lock_killable	✓
down_write	✓
_raw_write_lock_irqsave	✓
_raw_write_lock_irq	✓
_raw_read_lock	✓
_raw_read_lock_irq	✓
_raw_read_lock_bh	✓
_raw_read_lock_irqsave	✓
atomic_dec_and_mutex_lock	✓
cgroup_kn_lock_live	✓
ext4_lock_group	✓
native_queued_spin_lock_slowpath	✓
__mutex_lock_slowpath	✓
queued_write_lock_slowpath	✓
_atomic_dec_and_lock	✓
acpi_ev_acquire_global_lock	×
usermodehelper_read_lock_wait	×
console_lock	×
task_rq_lock	×
get_dap_lock	×
lock_mount	×
spi_bus_lock	×
ipc_lock	×
lock_trace	×
rio_lock_device	×
usb_lock_device_for_reset	×
__srcu_read_lock	×

REFERENCES

[1] [n.d.]. IDA Pro. https://www.hex-rays.com.
[2] [n.d.]. syzbot. https://syzkaller.appspot.com/upstream.
[3] Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. 2017. Reproducing Concurrency Failures from Crash Stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 705–716. https://doi.org/10.1145/3106237.3106292
[4] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. (2010), 255–268. https://doi.org/10.1145/1806596.1806626
[5] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. 2005. Applications of Synchronization Coverage. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 206–212. https://doi.org/10.1145/1065944.1065972
[6] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. (2010), 167–178. https://doi.org/10.1145/1736020.1736040
[7] Yan Cai and Zijiang Yang. 2016. Radius Aware Probabilistic Testing of Deadlocks with Guarantees *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 356–367. https://doi.org/10.1145/2970276.2970307
[8] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A Deployable Sampling Strategy for Data Race Detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 810–821. https://doi.org/10.1145/2950290.2950310
[9] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting Concurrency Memory Corruption Vulnerabilities *(ESEC/FSE*

2019). Association for Computing Machinery, New York, NY, USA, 706–717. https://doi.org/10.1145/3338906.3338927

[10] Yufei Chen and Haibo Chen. 2013. Scalable Deterministic Replay in a Parallel Full-System Emulator. *SIGPLAN Not.* 48, 8 (Feb. 2013), 207–218. https://doi.org/10.1145/2517327.2442537

[11] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (Los Angeles, CA, USA) *(PPREW-5)*. Association for Computing Machinery, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/2843859.2843867

[12] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: A Race and Transaction-Aware Java Runtime. (2007), 245–255. https://doi.org/10.1145/1250734.1250762

[13] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162.

[14] Mahdi Eslamimehr and Jens Palsberg. 2014. Race Directed Scheduling of Concurrent Programs. (2014), 301–314. https://doi.org/10.1145/2555243.2555263

[15] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. (2009), 121–133. https://doi.org/10.1145/1542476.1542490

[16] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 415–431.

[17] Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

[18] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[19] Jeff Huang and Charles Zhang. 2012. LEAN: Simplifying Concurrency Bug Reproduction via Replay-Supported Execution Reduction. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 451–466. https://doi.org/10.1145/2384616.2384649

[20] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. *SIGPLAN Not.* 48, 6 (June 2013), 141–152. https://doi.org/10.1145/2499370.2462167

[21] Ali Jannesari and Walter F. Tichy. 2010. Identifying ad-hoc synchronization for enhanced race detection. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 1–10. https://doi.org/10.1109/IPDPS.2010.5470343

[22] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

[23] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 582–598. https://doi.org/10.1145/3132747.3132767

[24] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of in-Production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 344–360. https://doi.org/10.1145/2815400.2815412

[25] Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. 2011. Coverage Metrics for Saturation-Based and Search-Based Testing of Concurrent Software. In *Proceedings of the Second International Conference on Runtime Verification* (San Francisco, CA) *(RV'11)*. Springer-Verlag, Berlin, Heidelberg, 177–192. https://doi.org/10.1007/978-3-642-29860-8_14

[26] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. 2011. Offline symbolic analysis to infer total store order. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 357–358. https://doi.org/10.1109/HPCA.2011.5749743

[27] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. 2009. Offline Symbolic Analysis for Multi-Processor Execution Replay. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 564–575. https://doi.org/10.1145/1669112.1669182

[28] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding Thousands of Concurrency Bugs during Testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 162–180. https://doi.org/10.1145/3341301.3359638

[29] Christopher Lidbury and Alastair F. Donaldson. 2019. Sparse Record and Replay with Controlled Scheduling. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 576–593. https://doi.org/10.1145/3314221.3314635

[30] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News* 36, 329–339. https://doi.org/10.1145/1346281.1346323

[31] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency Debugging with Differential Schedule Projections. *SIGPLAN Not.* 50, 6 (June 2015), 586–595. https://doi.org/10.1145/2813885.2737973

[32] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2016. Production-Guided Concurrency Debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 29, 12 pages. https://doi.org/10.1145/2851141.2851149

[33] Nuno Machado, Paolo Romano, and Luís Rodrigues. 2018. CoopREP: Cooperative record and replay of concurrency bugs. *Software Testing, Verification and Reliability* 28, 1 (2018), e1645.

[34] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-Race Detection. *SIGPLAN Not.* 44, 6, 134–143. https://doi.org/10.1145/1543135.1542491

[35] Ruijie Meng, Biyun Zhu, Hao Yun, Haicheng Li, Yan Cai, and Zijiang Yang. 2019. CONVUL: An Effective Tool for Detecting Concurrency Vulnerabilities. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 1154–1157. https://doi.org/10.1109/ASE.2019.00125

[36] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. 2009. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/1508244.1508254

[37] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 267–280.

[38] Robert H. B. Netzer and Barton P. Miller. 1992. What Are Race Conditions? Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.* 1, 1 (March 1992), 74–88. https://doi.org/10.1145/130616.130623

[39] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. 2011. Efficient Data Race Detection for Distributed Memory Parallel Programs. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 51, 12 pages. https://doi.org/10.1145/2063384.2063452

[40] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1508244.1508249

[41] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 177–192. https://doi.org/10.1145/1629575.1629593

[42] Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. 2013. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 643–654. https://doi.org/10.1145/2485922.2485977

[43] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. 2011. CoreRacer: A Practical Memory Race Recorder for Multicore X86 TSO Processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 216–225. https://doi.org/10.1145/2155620.2155646

[44] Sri Varun Poluri and Murali Krishna Ramanathan. 2015. Deterministic dynamic race detection across program versions. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 181–190. https://doi.org/10.1109/ICSM.2015.7332464

[45] Shiru Ren, Le Tan, Chunqi Li, Zhen Xiao, and Weijia Song. 2016. Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 551–564. https://doi.org/10.1145/2797022.2797028

[46] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[47] Ohad Shacham, Mooly Sagiv, and Assaf Schuster. 2005. Scaling Model Checking of Dataraces Using Dynamic Information. (2005), 107–118. https://doi.org/10.1145/1065944.1065958

[48] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

[49] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. 2008. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the 2008 international symposium on Software testing and analysis*. 143–154. https://doi.org/10.1145/1390630.1390649

[50] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2012. DoublePlay: Parallelizing Sequential Logging and Replay. *ACM Trans. Comput. Syst.* 30, 1, Article 3 (Feb. 2012), 24 pages. https://doi.org/10.1145/2110356.2110359

[51] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. (2001), 70–82. https://doi.org/10.1145/504282.504288

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

[53] Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage Guided Systematic Concurrency Testing. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/1985793.1985824

[54] Tao Wang, Xiao Yu, Zhengyi Qiu, Guoliang Jin, and Frank Mueller. [n.d.]. BARRIERFINDER: Recognizing Ad Hoc Barriers. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 323–327.

[55] James R. Wilcox, Parker Finch, Cormac Flanagan, and Stephen N. Freund. 2015. Array Shadow State Compression for Precise Dynamic Race Detection *(ASE '15)*. IEEE Press, 155–165. https://doi.org/10.1109/ASE.2015.19

[56] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. 2010. Ad Hoc Synchronization Considered Harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 163–176.

[57] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. 2017. SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2139–2154. https://doi.org/10.1145/3133956.3134085

[58] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 485–502. https://doi.org/10.1145/2384616.2384651

[59] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1095810.1095832

[60] Xiang Yuan, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Wenwen Wang, Jianjun Li, and Di Xu. 2013. Synchronization Identification through On-the-Fly Test. In *Proceedings of the 19th International Conference on Parallel Processing* (Aachen, Germany) *(Euro-Par'13)*. Springer-Verlag, Berlin, Heidelberg, 4–15. https://doi.org/10.1007/978-3-642-40047-6_3

[61] Xiang Yuan, Chenggang Wu, Zhenjiang Wang, Jianjun Li, Pen-Chung Yew, Jeff Huang, Xiaobing Feng, Yanyan Lan, Yunji Chen, and Yong Guan. 2015. ReCBuLC: Reproducing Concurrency Bugs Using Local Clocks. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 824–834. https://doi.org/10.1109/ICSE.2015.94

[62] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs through Sequential Errors. (2011), 251–264. https://doi.org/10.1145/1950365.1950395