Detecting Concurrency Memory Corruption Vulnerabilities

Yan Cai*
State Key Laboratory of Computer Science, Institute of Software Chinese Academy of Science
China
ycai.mail@gmail.com

Biyun Zhu
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, and University of Chinese Academy of Sciences
China
zhuby@ios.ac.cn

Ruijie Meng
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, and University of Chinese Academy of Sciences
China
mengrj@ios.ac.cn

Hao Yun
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, and University of Chinese Academy of Sciences
China
yunhao@ios.ac.cn

Liang He
TCA, Institute of Software, Chinese Academy of Sciences
China
heliang@iscas.ac.cn

Purui Su
TCA/SKLCS, Institute of Software, Chinese Academy of Sciences, and School of Cyber Security, University of Chinese Academy of Sciences
China
purui@iscas.ac.cn

Bin Liang
School of Information, Renmin University of China
China
liangb@ruc.edu.cn

ABSTRACT

Memory corruption vulnerabilities can occur in multithreaded executions, known as concurrency vulnerabilities in this paper. Due to non-deterministic multithreaded executions, they are extremely difficult to detect. Recently, researchers tried to apply data race detectors to detect concurrency vulnerabilities. Unfortunately, these detectors are ineffective on detecting concurrency vulnerabilities. For example, most (90%) of data races are benign. However, concurrency vulnerabilities are harmful and can usually be exploited to launch attacks. Techniques based on maximal causal model rely on constraints solvers to predict scheduling; they can miss concurrency vulnerabilities in practice. Our insight is, a concurrency vulnerability is more related to the orders of events that can be reversed in different executions, no matter whether the corresponding accesses can form data races. We then define exchangeable events to identify pairs of events such that their execution orders can be probably reversed in different executions. We further propose algorithms to detect three major kinds of concurrency vulnerabilities. To overcome potential imprecision of exchangeable events, we also adopt a validation to isolate real vulnerabilities. We implemented our algorithms as a tool ConVul and applied it on 10 known concurrency vulnerabilities and the MySQL database server. Compared with three widely-used race detectors and one detector based on maximal causal model, ConVul was significantly more effective by detecting 9 of 10 known vulnerabilities and 6 zero-day vulnerabilities on MySQL (four have been confirmed). However, other detectors only detected at most 3 out of the 16 known and zero-day vulnerabilities.

CCS CONCEPTS

• **Software and its engineering → Multithreading**; **Software testing and debugging**.

KEYWORDS

Concurrency, Multithreaded, Race Conditions, Vulnerability

ACM Reference Format:
Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338927

1 INTRODUCTION

Memory corruption vulnerabilities are frequently exploited to launch various attacks. Unfortunately, these vulnerabilities can also exist in multithreaded programs due to improper synchronizations [56, 59]. In this paper, we call memory corruption vulnerabilities caused by improper synchronizations in multithreaded programs as

*Yan Cai is the corresponding author.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338927

concurrency memory corruption vulnerabilities or *concurrency vulnerabilities* for short. They are extremely harmful and can be exploited to launch severe attacks. For example, the notorious vulnerability DirtyCow [2] in Linux kernel occurs in multithreaded environment and can be exploited to gain root privileges for a normal user. It has existed in Linux kernel for about ten years (from 2007 to 2016). Hence, it is urgent to detect concurrency vulnerabilities.

However, the detection of concurrency vulnerabilities is challenging. One straightforward approach would be to explore all possible thread interleaving (e.g., model checking approaches [15]). However, unlike sequential programs, the executions of multithreaded programs suffer from interleaving space explosion problem [34]. In practice, it is impossible to explore all executions of a multithreaded, especially on large-scale programs.

A feasible solution is to "borrow" approaches on detecting concurrency bugs [56]. Researchers have adopted data race (race for short) detectors [20]. These detectors have been used to detect all possible candidates for further detection of concurrency vulnerabilities [59]. However, it is ineffective to apply the approaches for race detection to detect concurrency vulnerabilities. This is because the two concepts are not the same one. A data race involves two concurrent accesses to the same memory locations [20]; but a concurrency vulnerability involves two or more memory operations on a set of closely related memory locations [59].

Another kind of feasible approaches is based on maximal causal models [47] to predict additional execution from a single one [19, 23]. Although the model is sound, it is too restricted and a tool based on pure maximal causal model can miss real vulnerabilities [19]. Besides, the prediction relies on constraint solver. Without an efficient solver, the solving process will be time-consuming. In practice, a tool usually adapts a relaxed model and, consequently, compromises its ability (e.g., UFO [23] missed 80% known concurrency Use-After-Free vulnerabilities in our experiment).

In this paper, we target on detecting concurrency vulnerabilities involving memory corruptions. We currently focus on three kinds of concurrency memory corruptions including: Use-After-Free (UAF), Null-Pointer-Dereference (NPD), and Double-Free (DF), which are mostly considered to be caused by orders [46, 52].

Our insight is that, the key to detect concurrency vulnerabilities is to determine whether two or more out of a set of events (operating memory blocks) in a given execution are exchangeable. That is, whether their occurrence orders can be different in alternative executions.If so, they may cause a concurrency vulnerability.

From the above viewpoint, we define *Exchangeable Events* (Section 3) to determine whether the orders of two events can be probably reversed. Although two events in a race defined by happens-before relation [27] are also regarded to be exchangeable, it is strictly based on synchronizations. Our exchangeable events are defined across synchronizations. Hence, our definition has larger coverage. Based on exchangeable events, we design three algorithms to detect three kinds of concurrency vulnerabilities from correct executions. As relaxed exchangeable events are not 100% precise exchangeable, we further isolate real ones via scheduling.

We have implemented our framework as a prototype tool ConVul. To evaluate it, we selected a set of 10 known concurrency vulnerabilities from a CVE database [1] and the latest MySQL server. For comparison, we also selected three well-known and representative race detectors, as well as a recent work UFO that detects concurrency UAFs. The experimental results show that, ConVul significantly outperformed four tools. It detected 9 out of 10 known vulnerabilities. But the three race detectors and UFO only detected 1 or 2 of them. On MySQL, ConVul reported 6 concurrency vulnerabilities and 4 of them have been officially confirmed by MySQL developers. However, the three race detectors only detected 1 of the 6 zero-day concurrency vulnerabilities on MySQL. UFO failed to detect any one of them.

In summary, this paper makes the following contributions:

- It proposes a concept known as (relaxed) exchangeable events to describe pairs of events, indicating that the execution orders of each pair of events can be reversed with a high probability in alternative executions.
- It proposes three algorithms to detect three kinds of concurrency vulnerabilities (UAF, NPD, and DF) from correct executions.
- It develops a framework ConVul and reports a set of experiments. The experimental result shows that, compared with both race detectors and a recent work, ConVul is significant effective on detecting both known concurrency vulnerabilities and zero-day ones.

Our data is available at: https://github.com/mryancai/ConVul.

2 CHALLENGES

Due to the non-determinism property of multithreaded executions, concurrency vulnerabilities only manifest themselves under certain thread interleaving and are extremely difficult to be detected. Hence, traditional approaches (e.g., fuzzing) targeting on sequential programs seldom detect concurrency vulnerabilities. It is also infeasible to explore all possible interleaving.

Recently, researchers discussed [56] and tried to apply approaches on detecting concurrency bugs to detect concurrency vulnerabilities [59]. The most promising ones are race detectors; because races are widely considered as a cause to concurrency vulnerabilities and there is also a category of vulnerabilities known as "Race condition" [1], where a race is theoretically defined to be two concurrent accesses (unordered and with at least one write) to the same memory blocks [20]. Although this is indeed true, by focusing on races, we actually focus on the correctness property of multithreaded executions. Unfortunately, there is no gold rule to define races in practice [7, 18, 32]. For example, among popular race detectors, some are based on happens-before relations [20, 27] and some are based on the lockset discipline [3, 45] as well as the hybrid of them [49]. Hence, it is still unclear whether and how a race detector can be well adopted to detect concurrency vulnerabilities. For example, in a recent work [59], among all 24, 645 races in Linux kernel reported by various race detectors, only 36 races are finally confirmed to be related to concurrency vulnerabilities. There are other disadvantages by directly applying existing race detectors to detect concurrency vulnerabilities.

Firstly, race detectors not only report false positives but also miss true positives [33]. Even in our experiment (Section 6), 80%

[1]Note, race conditions are not completely the same as races, although some works do not distinguish them [20].

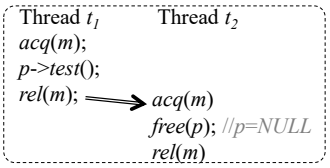

Figure 1: A concurrency vulnerability involving no race.

of known concurrency vulnerabilities cannot be detected by three popular race detectors.

Secondly, not all concurrency vulnerabilities can be detected by race detectors. This is because that a race is restricted to two concurrent accesses that are expected to occur in parallel. However, a concurrency vulnerability does not have such a restriction; it can even occur in a race-free multithreaded program, as long as the involved accesses can have a different execution order. For example, Figure 1 shows two threads: a thread t_1 dereferences a pointer via *p->test()*, and a second thread t_2 frees the same pointer *free(p)*. And a concurrency UAF can occur if thread t_2 executes all its three lines before thread t_1. However, the two accesses in the UAF (*p->test()* and *free(p)*) are well ordered by the same lock m, no matter which thread executes first. Hence, no race detectors can report a race on the two accesses (suppose there is a write in *free(p)* to p).

Lastly, concurrency vulnerabilities particularly refer to those causing memory corruptions due and can be exploited to launch severe attacks [52]. However, only about 8% to 10% races are harmful [18, 38]. That is, a larger number of races are actually benign and even some races are deliberately introduced [45]. If we rely on race detectors, we have to pay additional effort to identify harmful ones.

Other kinds of major concurrency bugs like atomicity violations and order violations may also cause concurrency vulnerabilities [56]. Given an atomic region or an order between two events, it will be easy to further identify any violations to them. However, the challenge is, how to identify atomic regions or expected orders of two events [30, 36, 40].

Another direction to detect concurrency vulnerabilities is based on scheduling prediction. The representative one is based on the Maximal Casual Model [47]. There have been works to predict concurrency NPDs [19], and concurrency UAFs [23]. ExceptioNull relaxes this model to detect concurrency NPDs which can miss vulnerabilities [19]. Another limitation of this model is that, it relies on constraint solver to determine the feasibility of each potential scheduling. This introduces performance challenges as it is usually time consuming to solve a large set of constraints. Hence, in practice, these tools have to relax their model based on heuristics. This not only affects their precision but also makes them to miss real vulnerabilities. For example, to improve performance of the constraint solver Z3, UFO adopts a sub-optimal model which can miss UAFs (i.e., *"does not encode all possible UAFs"*, see the two paragraphs right above Section 4.3 in [23]) as also verified in our experiments.

3 BACKGROUND AND DEFINITIONS

3.1 Background

This paper focuses on multithreaded programs and is based on sequential consistency memory model [28]. A multithreaded program consists of a set of threads that execute a set of events concurrently and coordinate their paces via synchronizations on locks (other synchronization primitives can be defined similarly). Particularly, we focus on the following types of events: (1) Memory **read** and **write**: *read(t, m)* and *write(t, m)*, a thread t reads from or writes to a memory block m and (2) Lock **acquisition** and **release**: *acq(t, l)* and *rel(t, l)*, a thread t acquires or releases a lock l.

For simplicity, we may omit the thread identifiers in above events (e.g., to use $read(m)$ instead of $read(t, m)$). We also use $Tid(e)$ to extract the thread identifier that executes event e.

An execution trace σ is a sequence of all events; σ_{t_i} is a projection of σ on thread t_i (i.e., all events from thread t_i).

The **happens-before relation** (HBR for short, denoted as $\rightarrow$) [27] is defined as three rules:

(1) Program order: given two events α and β from the same thread, if the event α is executed before β, then $\alpha \rightarrow \beta$.
(2) Synchronization order: if a lock l is released by a thread, denoted as $rel(l)$, and is later acquired by another thread, denoted as $acq(l)$, then $rel(l) \rightarrow acq(l)$.
(3) Transition property: if $\alpha \rightarrow \beta$ and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$.

HBR can be tracked via vector clocks [20, 27]. A vector clock VC of size n is an array of n integers. The corresponding algorithms are well-known [3, 7, 20, 32] and we will not discuss them in details. The basic idea is to set three kinds of basic vector clocks VC_t, VC_l, and VC_m for each thread t, for each lock l, and for each memory block m, respectively. They are maintained on various events and checked to determine the happens-before orders of two events.

3.2 Exchangeable Events

The execution of multithreaded program exhibits non-determinism. Given two events, their execution orders may be different in different executions. However, there are still many events where their execution orders are fixed among all executions. We are more interested in the former. If their execution orders can be different from the observed orders, they may cause a concurrent vulnerability. To describe such pairs of events, we define **exchangeable events**.

Definition 1. *Given two events e_1 and e_2, among all executions, if both $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_1$ are observed, we say that e_1 and e_2 are a pair of exchangeable events.*

The orders of two exchangeable events must be reversible as both orders are already observed. And we call such two events as **strict** exchangeable events. However, it is extremely difficult to observe all two orders in limited executions.

We then propose a concept of relaxed exchangeable event. Before that, we firstly present two concepts to describe such kinds of events. A **sync-edge** ($\Rightarrow$) is an edge either (1) from an event $acq(m)$ to its paired event $rel(m)$ in the same thread or (2) from an event $rel(m)$ to a later event $acq(m)$ by two different threads. Based on sync-edge, we define the **sync-distance** (or **distance** for short) of two event e_1 and e_2 as the minimal number of sync-edges that order the two events, denoted as $D(e_1, e_2)$.

Figure 2 shows three threads and their synchronizations that totally execute 11 non-synchronization events, where each gray block indicates a pair of acquisition and release on the same lock. On the right of Figure 2, we also list the distances of all pairs of

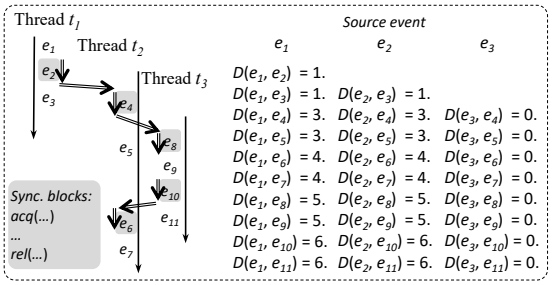

Figure 2: A demonstration on sync-distances.

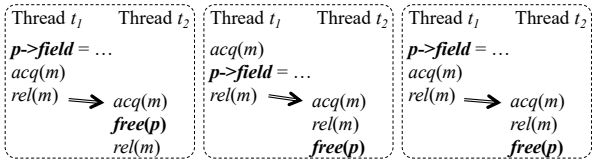

Figure 3: Three variants of the example in Figure 1.

events from e_1, e_2, and e_3. For example, there are totally 4 sync-edges that order e_2 and e_6; hence, we have $Dist(e_2, e_6) = 4$ (where the edges are along the path e_2, e_4, e_5, e_6 which is short than the path $e_2, e_4, e_8, e_9, e_{10}, e_6$). And we have $D(e_3, e_4) = 0$ as there is no sync-edge ordering them.

Obviously, the minimal distance of two events is zero, indicating that no sync-edge ordering the two events; and they are expected to execute in parallel (if they are from two threads). In all other cases, there is at least one sync-edge ordering them. Hence, one of them actually happens-before the other. That is, the minimal non-zero distance of two events (from two threads) is 3. [2] This case is shown in Figure 1, which further has three variants as shown in Figure 3 (where, in each of the three, thread t_1 executes first and the distance of two accesses in bold is 3 [3]). We are interested in these cases where the distance is larger than zero. In such cases, the order of the two events may still be reversed as shown in Figures 1 and 3.

Intuitively, given two ordered events, if their distance is smaller, there will be a higher probability to reverse the execution order of the two events. Besides, if there is a third event such that the both distances from two events to it are smaller, then there will be a higher probability to reverse the execution order of the two events. Based on these two heuristics, we propose ***d*-relaxed** exchangeable events that is predictable from a single execution, as illustrated in Figure 4.

Definition 2. *Given an execution trace σ and its projections on two different threads t_1 and t_2 as $\sigma_{t_1} = \langle ..., e_1, ... \rangle$ and $\sigma_{t_2} = \langle ..., e_{any}, ..., e_2, ... \rangle$, respectively, let $d = D(e_1, e_2)$, if either (1) $d = 0$ or (2) $d \geq 1 \wedge D(e_{any}, e_2) \leq 1 \wedge D(e_{any}, e_1) = 0$ holds, then two events e_1 and e_2 are d-relaxed exchangeable events, denoted as $e_1 \leftrightarrow_d e_2$.*

For example, in Figure 1, there is a pair of 3-relaxed exchangeable events, where e_1 = *p->test()*, e_2 = *free(p)*, and e_{any} is any event right

[2]Note, for two events from the same thread, their minimal distance can be 1 and no two events have a distance of 2.

[3]Unlike the example in Figure 1, for all cases in Figure 3, a race detector can report a race on two accesses if the lock acquisition order on m by two threads can be reversed.

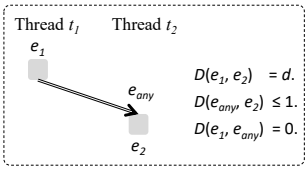

Figure 4: Illustration on d-relaxed exchangeable events.

before $acq(m)$ of thread t_2 (not shown). Generally, for a consecutive lock acquisition and release by two threads, if the HBR between two events are only determined by the two acquisitions, then the two events are d-relaxed exchangeable events for some value of d. In the rest of this paper, without explicit explanation, all exchangeable events refer to d-relaxed exchangeable events.

Admittedly, like HBR, our above definition is not 100% precise. A precise approach to determine whether the order of two ordered events observed in one execution requires analyzing all memory read and write accesses of all threads between the two events [33, 50]. This might be (partially) feasible in theory but ineffective in practice, especially on large-scale programs [26]. Besides, in most cases, any memory read or write in between two events does not affect the order of them; hence, such algorithms will further limit the interleaving coverage on inferring exchangeable events, i.e., reporting false negatives. Hence, our definition by considering synchronizations plus an "evidence" (i.e., e_{any}) is more practical and provides larger coverage.

Intuitively, with increasing value of d, the probability to reverse the order of a pair of d-exchangeable events may decrease due to other constraints besides synchronizations. Hence, in this paper, we restrict the value of d to be 3, i.e., the minimal non-zero distance for two events from two threads. It is also the default value when the distance of two exchangeable events is not mentioned in this paper. In experiment, we show a result under additional difference distances.

4 OUR APPROACH: CONVUL

4.1 Overview

ConVul dynamically analyzes executions and identifies sets of critical operations related to memory operations. If they can form any of three kinds of concurrency vulnerabilities by exchanging their orders, they are reported as potential vulnerabilities. Next, ConVul tries to isolate real ones by scheduling executions to trigger their occurrence.

In the rest of this section, we firstly present a basic tracking algorithm for exchangeable events in Subsection 4.2. Then, we analyze how three kinds of concurrency vulnerabilities can occur in multithreaded executions as well as how to detect them by our ConVul.

4.2 Check Exchangeable Events

ConVul detects concurrency vulnerabilities according to a given distance value of d (i.e., consider d-relaxed exchangeable events). The naïve way to calculate the distance d is straight-forward. It only requires tracking all events and all synchronizations as a graph, where memory access events and lock acquisitions are nodes and

Algorithm 1 Check Exchangeable Events

1: $PredVC_e$ maps an event e to a predicted vector clock that should be formed when thread $Tid(e)$ releases a lock l.
2: **function** onAnyEvent(Event e)
3: $PredVC_e \leftarrow VC_t$, where $t = Tid(e)$.
4: **end function**
5: **function** AreExchangeable(e_1, e_2)
6: **if** $\neg(e_1 \rightarrow e_2 \vee e_2 \rightarrow e_1)$ **then** ▷ i.e., $D(e_1, e_2) = 0$
7: Return $True$
8: **end if**
9: Let σ be the current execution trace.
10: Let t_1, t_2 be the thread IDs of e_1 and e_2, respectively.
11: Let $e_{rel} = rel(l)$ be the first release on l after e_1 in σ_{t_1}, and $e_{acq} = acq(l)$ be the last acquisition on l before e_2 in σ_{t_2}.
12: Let e_{any} be the event right before e_{acq} in σ_{t_2}.
13: ▷ i.e., limit $D(e_{any}, e_2) \leq 1$
14: **if** $\neg(e_{any} \rightarrow e_1 \vee e_1 \rightarrow e_{any})$ and $VC_l = PredVC_{e_1}$ **then**
15: ▷ i.e., $D(e_{any}, e_1) = 0$ and $D(e_1, e_2) \leq 3$
16: Return $True$
17: **end if**
18: Return $False$
19: **end function**

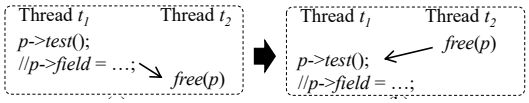

Figure 5: How a concurrency UAF occurs.

Algorithm 2 Detect Concurrency UAF

1: **function** onMemAccess(m, t)
2: Let e_m be this event.
3: $VC_{e_m}[t] \leftarrow VC_t[t]$ ▷ Track status of each event
4: **end function**
5: **function** onFree(p, t) ▷ p: a pointer to a memory block.
6: Let e_{fp} be this event.
7: $sz \leftarrow$ DerefSize(p)
8: **for** i from 0 to $sz - 1$ **do**
9: $m' \leftarrow Deref(p + i)$
10: **if** $\exists\, e_{m'}$, such that $(e_{m'} \leftrightarrow e_{fp})$ **then**
11: Report a UAF.
12: **end if**
13: **end for**
14: **end function**

lock acquisition orders (including both thread-local ones and across-threads ones) are edges. The distance of two events will the be the number of edges on a shortest path from one event to another on the graph. However, this implementation consumes too much memory. In this paper, we design a different one for checking 3-relaxed exchangeable events, by utilizing HBR tracking based on vector clocks, shown in Algorithm 1.

Given two events e_1 and e_2, Algorithm 1 firstly checks whether they are ordered by HBR; if not, they are exchangeable events. Otherwise, it further extracts three required events e_{rel} and e_{any} as well as e_{acq} (lines 11–12) and checks whether they satisfy the conditions in Definition 2 (lines 13–14).

Note that, in Algorithm 1, given line 12, the checking on the conditions in line 14 is an effective implementation of the conditions in Definition 2. Let's explain it below.

If the two events are ordered, there must be a lock release event e_{rel}; and a meaningful e_{any} must be in a different synchronization block from event e_2 (otherwise, the condition $D(e_{any}, e_1) = 0$ will not be satisfied). Hence, there will be at least one lock acquisition event e_{acq}. Besides, as we already limit the value of d in d-exchangeable event to be 3, there will be at most one sync-edge between two threads (e.g., see Figures 1 and 3).

As a result, there must be the same lock l, such that its release and acquisition are the above said lock release after e_1 and the lock acquisition e_{acq} in between e_{any} and e_2. This also indicates that the vector clock of the lock l (right after e_{rel} and right before e_{acq}) keeps unchanged. Hence, to reduce tracking effort, Algorithm 1 introduces a new vector clock $PredVC_e$ for each event e to keep a predicted vector clock of a lock l being held by $Tid(e)$ when this thread releases lock l (line 1). It is maintained when event e occurs (line 3). If there is no such a lock l, it becomes empty. Thus, the checking $VC_l = PredVC_{e_1}$ determines whether lock l is the one in above events e_{rel} and e_{acq}, which further determines whether $D(e_1, e_2)$ is 3 (line 14).

4.3 Analyze and Detect Concurrency UAFs

A UAF vulnerability occurs when a freed memory block is re-accessed [9]. It is known that more than 88% UAFs can be exploited to launch zero-day attacks [29]. A UAF vulnerability is caused by at least two memory accesses: (1) free a memory block pointed to by a pointer p and (2) later access the memory block via the pointer p. Therefore, it is possible that the two accesses can be performed by two threads.

Figure 5 shows an example of how a concurrency UAF can occur. There are two statements: *p->test()* and *free(p)* where p is a pointer. In correct executions (shown in Figure 5(a)), thread t_1 first dereferences pointer p and then, thread t_2 frees the memory block via pointer p. However, if the memory location is freed by thread t_2 first before its use in thread t_1, a UAF occurs, as shown in Figure 5(b).

The challenge is that, given the execution order in Figure 5(a), how we can know the existence of the order in Figure 5(b). This is essentially the difficulty in analyzing multithreaded programs. For example, there might exist the same locks protecting the two accesses in a concurrency UAF; and hence, no race detector can report a race on it. For other cases, race detector may also be ineffective, as discussed in Section 2. Of course, if these accesses are all protected by the same locks, no race detector can detect the UAF.

Our algorithm. Considering the semantics of UAFs and the multithreaded executions, we propose Algorithm 2 to detect possible concurrency UAFs. Given a correct execution trace, it tracks all memory accesses (lines 1–4) and updates corresponding vector clocks (line 3). Later, if there is any free call on p (denoted as event e_{fp}), Algorithm 2 checks whether there is any event accessing memory blocks pointed by p (denoted as event $e_{m'}$, lines 9–10). The function *DerefSize(p)* returns the size of the memory block pointed by p. For any such event $e_{m'}$, if $e_{m'}$ and e_{fp} are exchangeable events, Algorithm 2 reports a potential concurrency UAF (lines 10–11).

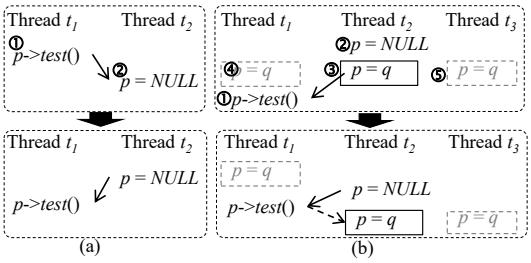

Figure 6: How a concurrency NPD occurs.

Algorithm 2 overcomes limitations of race detectors by checking whether two events are exchangeable events, no matter they are well ordered by HBR or not.

4.4 Analyze and Detect Concurrency NPDs

When a pointer is set to be NULL and is dereferenced later, a NULL pointer dereference (NPD) occurs. It can be usually exploited to launch various attacks [4, 5]. A NPD involves a memory write to a pointer with a NULL value and a later dereference (e.g., *p->test()*). The two events in a NPD can be produced by two different threads as shown in Figure 6(a), where a thread t_1 dereferences a pointer p and a second thread t_2 later sets the pointer to be NULL. However, if the order of two events can be reversed so that thread t_2 firstly writes the pointer to be NULL and then thread t_1 tries to dereference this pointer, causing a concurrency NPD. To ease our presentation, we use circled number to denote each statement/event.

In practice, however, it is not straightforward to analyze a concurrency NPD. This is because there are often other events in between the two events ① and ②. We have systematically analyzed all possible cases and identify three unique ones. Besides the simple case shown in Figure 6(a), another three cases are shown in Figure 6(b) where one more event (i.e., ③, ④, or ⑤) writing the pointer (i.e., $p = q$) exists in between the two events ① and ②. And the pointer writing event $p = q$ can be executed by either of two threads or a third thread (i.e., forming an event ③, ④, or ⑤, respectively) as long as it is executed in between the two events.

In these cases, if the pointer writing event $p = q$ occurs before the NULL-writing event by thread t_2 (as ④) or after the pointer dereference by thread t_1 (as ③ or ⑤), a concurrency NPD occurs. On detecting concurrency NPDs, race detectors can precisely report a race on ① and ② in Figure 6(a). However, for three cases in Figure 6(b), although they can report races, these races cannot be directly related to concurrency NPDs.

Our algorithm. Our algorithm (Algorithm 3) tracks all recent NULL value writes to all pointers (i.e., *OnMemWrite(p, val, t)*) as well as all pointer dereferences (i.e., *onMemRead(p, ins, t)*, where *ins* is the instruction containing the read event to pointer p). In order to detect all four types of concurrency NPDs in Figure 6, Algorithm 3 maintains three additional data structures: $VC_p^{W_{Nul}}$, $VC_p^{W_{nNul}}$, and VC_p^R, to track NULL value writes, Non-NULL values write, and reads (i.e., dereferences) to a pointer p, respectively.

During runtime, for each memory write, Algorithm 3 checks the value to write to pointer p (line 5). If the value is NULL and there was a read event to the same pointer p, it will report a concurrency NPD if this read event and the current write event p (lines 7–8) are

Algorithm 3 Detect Concurrency NPD

```
1:  VC_p^{W_Nul}: Track NULL value writes to pointer p.
2:  VC_p^{W_nNul}: Track Non-NULL value writes to pointer p.
3:  VC_p^R: Track reads to pointer p.
4:  function onMemWrite(p, val, t)
5:      if val = NULL then
6:          Let e_p^{W_Nul} be this event.
7:          if ∃ e_p^R such that e_p^{W_Nul} ↔ e_p^R then
8:              Report a NPD.                          ▷ Case (a)
9:          end if
10:         VC_p^{W_Nul}[t] ← VC_t[t]                  ▷ Null-value writes
11:     else
12:         VC_p^{W_n}[t] ← VC_t[t]                    ▷ Non-null-value writes
13:     end if
14: end function
15: function onMenRead(p, ins, t)
16:     Let e_p^R be this event.
17:     if isDeref(ins) ≠ True then
18:         Return
19:     end if
20:     if ∃ e_p^{W_Nul} ∧ e_p^{W_nNul} then
21:         if e_p^{W_Nul} ↔ e_p^{W_nNul} then
22:             Report a NPD.                          ▷ Case (b): ② || ③/④/⑤
23:         else if e_p^R ↔ e_p^{W_nNul} then
24:             Report a NPD.                          ▷ Case (b): ③/④/⑤ || ①
25:         end if
26:     end if
27:     VC_p^R[t] ← VC_t[t]
28: end function
29: function IsDeref(ins)
30:     Let ins' ← NextIns(ins)
31:     if hasR(ins) ∧ hasRW(ins') ∧ BaseReg(ins') = OprandReg(ins) then
32:         Return True
33:     end if
34:     Return False
35: end function
```

exchangeable events. This type of concurrency NPD corresponds to the one shown in Figure 6(a). Next, Algorithm 3 tracks this write event by updating $VC_p^{W_{Nul}}$ (line 10) or $VC_p^{W_{nNul}}$ (line 12), according to whether the value to write to pointer p is NULL.

For the concurrency NPDs shown in Figure 6(b), Algorithm 3 detects them when a pointer is read (in *onMemRead(p, ins, t)*). For a memory read event e_p^R to a pointer p, if there are two events ($e_p^{W_{Nul}}$ and $e_p^{W_{nNul}}$ which are, respectively, expected to be ② and one of ③, ④, and ⑤ in Figure 6(b)) that write a NULL value and a Non-NULL value to p, Algorithm 3 further checks the following conditions to detect concurrency NPDs:

- If the two events $e_p^{W_{Nul}}$ and $e_p^{W_{nNul}}$ are exchangeable events, a concurrency NPD is reported (lines 21–22). Here, the two events are ② and ④ or ② and ⑤.
- If the two events e_p^R and $e_p^{W_{nNul}}$ are exchangeable events, a concurrency NPD is also reported (lines 23–24). Here, the two events are ① and ③ or ① and ⑤.

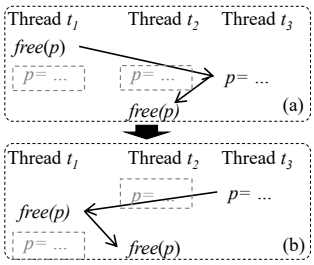

Figure 7: How a concurrency DF occurs.

In Algorithm 3, we check whether a memory read is a pointer dereference in *IsDeref(ins)*. It is based on the following heuristic: a pointer dereference usually corresponds to two consecutive binary instructions *ins* and its next instruction *ins′* (line 31), where the Operand Register of the first memory read operations *ins* is used as the Base Register of its next memory access instruction *ins′*. Of course, this heuristic might be imprecise and can be improved.

4.5 Analyze and Detect Concurrency DFs

A double free (DF) vulnerability occurs when a memory location is freed twice [9]. DF is also a kind of serious vulnerability and can be exploited to launch remote code execution attacks [17]. A DF involves two events and can occur in multithreaded programs. Unlike UAF and NPD, a DF involves the same type of two memory free events. Hence, once two free events on the same memory location occur, a DF must occur.

In practice, a concurrency DF may be hidden by other pointer assignments in between two free events. Hence, the two frees via the same pointer actually free two different memory blocks. The assignment can be either from one of the two threads or a third thread. We show an example in Figure 7(a). In the example, two threads t_1 and t_2 execute two frees on the same pointer p. And in between two *free(p)* calls, a write to p exists from either thread t_1 (Case 1)), or thread t_2 (Case 2)), or thread t_3 (Case 3)). As a result, although the pointer p is freed twice, no DF occurs.

However, given a different thread schedule as shown in Figure 7(b), if the assignment to pointer p occurs after both *free(p)* (as Case 1)) or before both *free(p)* (as Cases 2) and 3)), a concurrency DF occurs.

Given the concurrency DF in Figure 7, a race detector may report a race between the read to p in *free(p)* and the write to p. This, however, is insufficient to infer a DF. Similarly, when it involves lock protection, no race detector can detect the DF.

Our algorithm. Algorithm 4 assumes, if there is any potential concurrency DF, there must exist three events: two free events on the same pointer by two different threads and one assignment to the same pointer (as shown in Figure 7).

Algorithm 4 maintains two data structures: a map *Pts* that maps a memory m to a set of pointers to memory block m, and a map *Frs* that maps a pointer p to a set of free events on p. The structure *Pts* is maintained by fully tracking any assignment from a memory block m to a pointer p (in *onPointerAssign(p, m)*). That is, when an address of a memory m is assigned to a pointer p, all mapped items to p from other memory m' are removed (line 5) and p is remapped from memory m (line 7).

Algorithm 4 Detect Concurrency DF

```
1:  Pts: a map from a memory m to a set of pointers to m.
2:  Frs: a map from a pointer to an event that frees p.
3:  function onPointerAssign(p, m)
4:      for ∀ m' such that Pts(m') ≠ ∅ do
5:          Pts(m') ← Pts(m') \ {p}
6:      end for
7:      Pts(m) ← Pts(m) ∪ {p}
8:  end function
9:  function onFree(m, t)          ▷ m: the 1st addr of a memory block.
10:     Let e_f be this event.
11:     for each p ∈ Pts(m) do
12:         Let e_p^W be the event associating pointer p to m.
13:         if ∃ e_p^F ∈ Frs(p) then
14:             if e_p^F ↔ e_p^W then
15:                 Report a DF.                          ▷ Case (a)
16:             else if e_p^W ↔ e_f then
17:                 Report a DF.                          ▷ Case (b)
18:             end if
19:         end if
20:         Frs(p) ← Frs(p) ∪ {e_f}
21:     end for
22: end function
```

The core part of Algorithm 4 is how it checks *free* events and detects concurrency DFs, as shown in function *onFree(m, t)*. Given a free event *free(p)* that actually frees memory m by thread t (denoted as event e_f), it tries to find a previous assignment event to pointer p from a memory m (denoted as event e_p^W), such that there is also a previous free event on the same pointer p (denoted as e_p^F) (lines 11–13). Then, a concurrency DF is reported in one of two cases:

- Case (a): The two events e_p^F and e_p^W are exchangeable events (line 14), i.e., the case where thread t_2 has an assignment to p in Figure 7.
- Case (b): The two events e_p^W and e_f are exchangeable events (line 16), i.e., the cases where thread t_1 or t_3 has an assignment to p in Figure 7.

4.6 Schedule Executions

HBR is not guaranteed to be 100% precise [20, 33]. Similarly, our relaxed exchangeable events are also not 100% precise. To overcome this imprecision we adopt scheduling techniques with aim to trigger occurrences of all reported concurrency vulnerabilities.

Given a set of ordered events from a trace, we try to produce a set of new orders (our target) among these events. The basic idea is to suspend the first event in each order to enforce the targeted order to

Table 1: Scheduling rules (~~Cancelled~~ events are not required to appear).

	Detected Orders	Targeted Orders
UAF	$\langle e_{use}, e_{free} \rangle$	$\langle e_{free}, e_{use} \rangle$
NPD	$\langle e_{deref}, e_{wNull} \rangle$	$\langle e_{wNull}, e_{deref} \rangle$
	$\langle e_{wNull}, e_{reAssign}, e_{deref} \rangle$	$\langle e_{reAssign}, e_{wNull}, \sout{e_{deref}} \rangle$ $\langle e_{wNull}, e_{deref}, \sout{e_{reAssign}} \rangle$
DF	$\langle e_{free1}, e_{reAssign}, e_{free2} \rangle$	$\langle e_{free1}, e_{free2}, \sout{e_{reAssign}} \rangle$ $\langle e_{reAssign}, e_{free1}, \sout{e_{free2}} \rangle$

occur. Table 1 shows both the events and the orders of each reported concurrency vulnerability by ConVul as well as the targeted orders among these events. There are totally six rules. In the last four rules, if the targeted order of the first two events is satisfied, then the third event may not appear (i.e., become meaningless as the vulnerability already occurs).

ConVul adopts the existing scheduling work [11, 12] to validate each reported vulnerability by including support of memory access events. As novelty of ConVul is at its predictive detection of concurrency vulnerabilities, we omit the detailed scheduling algorithm. Readers may refer to the work [11, 12].

5 DISCUSSION

ConVul is a two-phase approach to detect three kinds of concurrency vulnerabilities in real-world programs. It does not adopt heavy strategies (e.g., via maximal casual models) to find all vulnerabilities but relies on a practical prediction and a practical validation. Hence, it can miss vulnerabilities. However, its validation guarantees that no false positive is reported. In future, we will extend ConVul to detect more concurrency vulnerabilities.

6 EXPERIMENTS

6.1 Experiment Setup

6.1.1 Implementation. We implemented ConVul on top of Pin [31] for C\C++ programs with Pthread. ConVul instruments both synchronizations and memory accesses during program loading time to produce various events. During runtime, these events are passed to the algorithms of ConVul.

For comparison purpose, we also selected three representative race detectors: FastTrack (FT) [20], Helgrind (Hel) [3, 39], and Thread Sanitizer (TSan) [49], as well as a recent work UFO [23]. FT and Hel are two well-known Happens-before based race detector [27] and such detectors usually report fewer false positives but miss real races. TSan is a practical hybrid race detector based on both Happens-before relation and Lockset discipline [45], which is claimed to report the largest number of races without reporting false positives and benign races [49]. Both Hel and TSan are available online; for FT, we adopted the implementation from a recent work [22]. UFO targets on detecting concurrency UAF and is available online. ExceptioNull [19] detects NPDs based on the similar idea as UFO; unfortunately, it is for Java and is not available. Besides above tools, we adapted UFO on known vulnerabilities to detect concurrency NPDs, denoted as UFO_{NPD} (see the next subsection) .

We did not compare ConVul with detectors like Address Sanitizer [48] as which only detects occurred vulnerabilities but cannot predict ones from correct executions.

6.1.2 Benchmarks. We aim to evaluate the ability of ConVul at detecting not only known but also zero-day concurrency vulnerabilities, including on real-world large-scale programs. We then searched the NVD vulnerability database (i.e., at the site https://nvd.nist.gov). We restricted all CVEs under category of "Race" published in the past ten years (from 2008 to 2017). This results in 545 records. We then manually identified those caused by UAF, NPD, and DF, which resulted in 82 CVEs. Next, we excluded those on non-Linux platform (i.e., on "Android", "Apple", "Java", "Windows kernel", and "Qualcomm") and those with no clear descriptions or without source and inputs (like POC). Finally, there are 10 CVEs left. We extracted the code involving vulnerabilities. We also replaced non-Pthread synchronizations with Pthread ones. Table 2 shows the details of these CVEs, including their CVE IDs, Category, Program, and Detection Results by all detectors. We further selected MySQL database server (of the latest version 5.7.20 at the time of evaluation). MySQL is a widely-used large-scale programs consisting of 2, 244, 927 SLOC. It mainly adopts Pthread model with several customized synchronizations and contains 933 test cases.

Note, as UFO only detects UAFs, we revised the 5 known NPD vulnerabilities by changing the NULL pointer assignment as frees (i.e., change *"p=NULL;"* to *"free(p);"*). Thus, UFO (as UFO_{NPD}) are expected to detect the 5 NPDs as 5 UAFs. However, on MySQL, we are unable to do such changes for its whole code; and it also crashed after we applied above changes to all NPDs detected by ConVul. Hence, UFO_{NPD} was only "evaluated" on the 5 known NPDs.

6.1.3 Setting. The default distance of ConVul is 3 (i.e., to check 3-relaxed exchangeable events). We also configured it with other distances and present the results in Section 6.4. For other tools, we adopted their default configurations.

Our experiment was conducted on a ThinkPad workstation W541 with an i7-4710MQ processor, installed with Ubuntu 14.04, GCC 4.8, LLVM 3.6. We run all tools for 10 times and collected their results (but 100 times for collecting time and memory overhead).

6.2 Effectiveness on Known Vulnerabilities

Table 2 shows the results of the all detectors on 10 concurrency vulnerabilities. Note that, the three detectors only detect races; hence, we count a reported race as a "vulnerability" if it reflects a concurrency vulnerability; otherwise, we further check whether they report any race on the variable of a concurrency vulnerability; if so, we append a star marker (⋆) in the corresponding cell in Table 2. UFO only detects UAFs and UFO_{NPD} only detects NPDs; we put a "-" to indicate the case where they are not applicable.

From the table, ConVul successfully detected 9 out of 10 vulnerabilities. However, others only detected 1 to 2 vulnerabilities. In other words, all three race detectors missed at least 80% known vulnerabilities. These results are consistent with our claim that, not all concurrency vulnerabilities can be detected by race detectors. Even simply counting any race on variables involved in vulnerabilities (indicated by ⋆), three detectors still missed 4 to 6 (about 50%) vulnerabilities. UFO only detected 1 out of 4 known UAFs; and UFO_{NPD} only detected 2 out of 5 NPDs. Obviously, on 10 unknown CVEs, ConVul was significantly effective than others.

As ConVul only missed one vulnerability (which was also missed by all other detectors), we investigated it and outline its code below:

```
1  //Initialization
2  head->mm=NULL;
3  head->next=node;
4
5     Thread t_exit
6  acq(mmlist);
7  head->next = head;
8  rel(mmlist);
```

```
9     Thread t_worker
10 tmp=...;
11 if(head->next != head){
12
13    acq(mmlist);
14    tmp = head->next;
15    rel(mmlist);}
16 tmp->mm->... //dereference
```

Table 2: Descriptive statistics and detection results.

CVE ID	Category	Program	Detection Results					
			ConVul	FT	Hel	TSan	UFO	UFO_{NPD}
cve-2009-3547	NPD	Linux-2.6.32-rc6	✓	✓	✓	✓	-	✓
cve-2011-2183	NPD	Linux-2.6.39-3	✗	✗	✗	✗	-	✗
cve-2013-1792	NPD	Linux-3.8.3	✓	✗	✗	✗	-	✗
cve-2015-7550	NPD	Linux-4.3.4	✓	✗	✗	✗	-	✓
cve-2016-1972	UAF	Firefox-45.0	✓	✗	✗ (★)	✗ (★)	✗	-
cve-2016-1973	UAF	Firefox-45.0	✓	✗	✗	✗	✗	-
cve-2016-7911	NPD	Linux-4.6.6	✓	✗ (★)	✗ (★)	✗ (★)	-	✗
cve-2016-9806	DF	Linux-4.6.3	✓	✗ (★)	✗	✗ (★)	-	-
cve-2017-6346	UAF (DF)	Linux-4.9.13	✓	✗ (★)	✗ (★)	✗ (★)	✗	-
cve-2017-15265	UAF	Linux-4.13.8	✓	✗	✗	✓	✓	-
		Total	**9**	**1**	**1**	**2**	**1**	**2**

Table 3: Six zero-day concurrency vulnerabilities detected by ConVul on MySQL and a comparison with other detectors.

Bug ID	Category	Status	Detected by Others?			
			FT	Hel	TSan	UFO
MySQL-88311	UAF	Confirmed	✗	✗	✗	✗
MySQL-88911	NPD	Submitted	✗	✗	✗	✗
MySQL-88914	NPD	Submitted	✗	✗	✗	✗
MySQL-91448	NPD	Confirmed	✗	✓	✓	✗
MySQL-91449	NPD	Confirmed	✗	✗	✗	✗
MySQL-91896	NPD	Confirmed	✗	✗	✗	✗
		Total:	**0**	**1**	**1**	**0**

From the code, we see that, a NPD will occur if thread t_{exit} (lines 6, 7, and 8) executes in between line 11 and line 13 of thread t_{worker}. This NPD is caused by a NULL value write to pointer *mm* at line 2 and a dereference on the same pointer *mm* at line 16. In a normal execution, thread t_{worker} executes first followed by thread t_{exit}. This case falls out of our models shown in Figure 6 and ConVul failed to detect it. For three race detectors, as a lock *mmlist* is used to order two writes to *head->next*, no race was reported by them.

6.3 Effectiveness on Unknown Vulnerabilities

6.3.1 Summary of Results. Table 4 shows the number of concurrency vulnerabilities or races reported on MySQL by all tools. ConVul predicted 9 concurrency vulnerabilities and 6 of them was triggered. FT reported up to 2,208 races. Both Hel and TSan reported almost the same number of races: 536 and 546, respectively. Surprisingly, UFO detected no UAF.

6.3.2 Result Analysis. From Table 4, we see that the three race detectors reported many races (from hundreds to thousands). It is unknown whether all these reported races are real ones and how many of them reveal concurrency vulnerabilities. However, ConVul reported 9 concurrency vulnerabilities and 6 of them were zero-day vulnerabilities. We have reported these 6 vulnerabilities to MySQL developers. And 4 of them have been confirmed as real vulnerabilities; the remaining 2 are waiting for confirmation. We first discuss the 6 zero-day vulnerabilities and then discuss the three not triggered ones (in the next subsection).

Table 4: # of concurrency vul. or races reported on MySQL.

	ConVul	FT	Hel	TSan	UFO
#Vul. or #Races	9/6	2,208 *	536	546	0

* Due to limited debug information by Pin, this result contains duplicated races.

Table 3 shows all 6 zero-day concurrency vulnerabilities, consisting of 1 UAF and 5 NPDs. The table includes (1) Bug IDs allocated by MySQL Bugzilla after we submitted our result, (2) vulnerability category, and (3) vulnerability status. In the last major column, we indicate whether FT, Hel, TSan, and UFO detected the 6 vulnerabilities or not.

From the last major column, we see that, FT and UFO detected none of the 6 vulnerabilities; Hel and TSan both detected only one of them. This further confirms that race detectors are ineffective in detecting concurrency vulnerabilities and UFO can miss UAFs. Compared to them, our tool ConVul is more effective.

6.3.3 Study on False Positives from Prediction. For the three false positives, we have investigated them and found that all were caused by the similar reason, as illustrated below.

```
1      Thread t1
2  //may be a lock here
3  for(each node in list){
4    node -> ...;
5  }
```

```
6      Thread t2
7  //sometimes a lock here
8  for(each node in list){
9    if(node = ...){
10     remove(list, node);
11     free(node);}}
```

In the simplified code, there is a list structure. In testing executions, thread t_1 executes first to iterate each node in *list*. Hence, the pointer *node* is firstly dereferenced (line 4) and then freed by thread t_2 (line 11). And ConVul predicted this as a concurrency vulnerability. However, when thread t_2 executes first, it will remove the node from the list and then frees the memory pointed by *node*. Next, when thread t_1 executes, it will never read the node freed by thread t_2 again. As a result, the dereference is actually missing (as the node is no longer in the *list*).

We further verified that, for three variables in three false positives, FT reported three races while both Hel and TSan reported two races on two variables. Based on our above analysis, all these reports are false positives. Actually, such false positives cannot be easily excluded without scheduling runs, even for pure HBR based race detectors. And this is still a hot topic [26, 33, 50].

6.4 ConVul with Different Distances

Besides 3-relaxed exchangeable events, we also set other different distances (from 4 to 10) to ConVul and applied it to both 10 known vulnerabilities and MySQL.

On the 10 known vulnerabilities, ConVul did not detect them with $d > 3$. This is understandable as each of them only contains

code producing vulnerabilities. On MySQL, with $d = 4$, one NPD was detected; with $d = 5$, two NPDs were detected. They are a subset of the 6 vulnerabilities listed in Table 3. In other cases (i.e., $6 \leq d \leq 10$), no vulnerability was detected.

6.5 Ineffectiveness Analysis of Other Detectors

The three widely used race detectors were significantly ineffective in our experiment. After detailed investigation, we found that the reasons were still those which we have discussed in Sections 1 and 2. UFO adopted a sub-optimal model and hence can generates simple constraints that are accepted by high performance constraint solvers like Z3, which prevents it to detect all UAFs (see the last paragraph of Section 4.2 [23]). It only detected 1 UAF out of 5 known and unknown UAFs.

6.6 Performance Evaluation

Table 5: Time overhead (an average of 100 runs).

	Pin	ConVul	FT	Hel	UFO	TSan
CVE Avg.	269.8x	330.9x	289.8x	235.8x	482.4x	7.0x
MySQL	3.4x	117.1x	17.87x	117.7x	7.0x	4.1x

Table 6: Memory overhead (an average of 100 runs).

	Pin	ConVul	FT	Hel	UFO	TSan
CVE Avg.	81.0x	151.8x	144.6x	525.2x	433.9x	1.7x
MySQL	2.2x	42.3x	19.6x	41.3x	121.4x	11.1x

Pin [31] and Valgrind [39] are two heavy dynamic binary instrumentation framework. ConVul and FT are built on Pin and Hel is built on Valgrind. TSan is integrated into a program during compilation time. UFO collects traces at runtime and offline detects UAFs. To compare their performance, we collected both time overhead and memory overhead of all. On 10 CVE programs, all detectors incurred the similar time and memory overhead; we only show the average data on them. The data is shown in Table 5 and Table 6, including the overhead of Pin (with no tool) for reference.

From two tables, we see that, TSan incurred the least time overhead and memory overhead, which is reasonable due to its integration implementation. And we will not discuss it. Other four detectors incurred heavy time and memory overhead on 10 CVE programs, where the time overhead of UFO and the memory overhead of both Hel and UFO are extremely heavy (i.e., from >430x to >520x). On MySQL, both ConVul and Hel incurred almost the same time and memory overhead; however, UFO incurred the least time overhead but the largest memory overhead.

Overall, compared to both FT, Hel, and UFO, the performance of ConVul is acceptable to us.

7 RELATED WORKS

Detection of concurrency vulnerabilities has begun to gain more focus recently. The work [55] first shows important features of concurrency attacks and reveals that many of concurrency bugs can lead to severe attacks, such as privilege escalation, malicious code execution and security check bypassing. 2AD [54] focuses on the concurrent attacks on databases but cannot handle other programs efficiently. OWL [59] detects concurrency attacks based on race detectors and we have discussed it in this paper. It automatically identifies the real concurrency bugs and eliminate false positive produced by existing tools. And then, based on inter-procedural analysis and attack input fuzzer, OWL can detect the attack sites derived from shared memory corrupted by the concurrency bugs. Compared with OWL, ConVul does not need to identify the concurrency bugs, but directly focuses on concurrency vulnerabilities.

Among concurrency bugs, data race is the mostly closed one to concurrency vulnerabilities, as explained in this paper. There have been many works to detect them [10, 13, 20, 22, 25, 37, 41, 42, 45, 51, 53]. Razzer [24] focuses on fuzzing harmful races in Linux kernel. One hot topic on race detection is to improve the detection coverage in single executions [26, 33, 50] but not to explore all possible executions like Model checking [16]. Other topics are to focus on a small portion of interleaving space [35, 57] and randomized scheduling with guarantees [8]. These research directions can also benefit our ConVul to detect additional concurrency vulnerabilities.

Lastly, fuzzing techniques become one of most effective methods to detect sequential vulnerabilities by simply mutating inputs, including AFL [58] and Peach [6] which have found a huge number of notable security vulnerabilities in thousands of large-scale programs on various systems. Symbolic execution and taint propagation can be further used to improve code coverage and bypass some special input validation checks in fuzzing [14, 21, 44]. Stress testing can be an alternative one to increase thread parallelism [43]. ConVul can be integrated into these tools.

8 CONCLUSION

This paper studies how the order of two events can be reversed in different executions. It further proposes ConVul implementing three algorithms to detect concurrency vulnerabilities. The experiment results on 10 known concurrency vulnerabilities and on a database server demonstrated that ConVul is significantly more effective than existing works on detecting both known vulnerabilities and 0-day vulnerabilities.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China (NSFC) (Grant No. U1736209, U1836209, and 61602457), the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (Grant No. QYZDJ-SSW-JSC036), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. 2017151), the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2017QNRC001), and the Blockchain Technology and Application Joint Laboratory, Guiyang Academy of Information Technology (Institute of Software Chinese Academy of Sciences Guiyang Branch).

REFERENCES

[1] Feb. 2019 (last accessed). Common Vulnerabilities and Exposures. https://cve.mitre.org/.
[2] Feb. 2019 (last accessed). Dirty Cow (CVE-2016-5195) . https://dirtycow.ninja/.
[3] Feb. 2019 (last accessed). Helgrind: a thread error detector. http://valgrind.org/docs/manual/hg-manual.html.
[4] Feb. 2019 (last accessed). Much ado about NULL: Exploiting a kernel NULL dereference. https://blogs.oracle.com/linux/much-ado-about-null%3a-exploiting-a-kernel-null-dereference-v2.

[5] Feb. 2019 (last accessed). OpenSSL NULL Pointer Dereference Vulnerabilities. http://www.securiteam.com/securitynews/5FP3B00HQE.html.

[6] Feb. 2019 (last accessed).. PeachTech. https://www.peach.tech.

[7] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[8] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1736020.1736040

[9] Juna Caballero, Custavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: Early Detection of Dangling Pointers in Use-After-Free and Double-Free Vulnerabilities. In *the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[10] Yan Cai and Lingwei Cao. 2015. Effective and Precise Dynamic Detection of Hidden Races for Java Programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 450–461. https://doi.org/10.1145/2786805.2786839

[11] Y. Cai and Q. Lu. 2016. Dynamic Testing for Deadlocks via Constraints. *IEEE Transactions on Software Engineering* 42, 9 (Sep. 2016), 825–842. https://doi.org/10.1109/TSE.2016.2537335

[12] Yan Cai, Shangru Wu, and W. K. Chan. 2014. ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 491–502. https://doi.org/10.1145/2568225.2568312

[13] 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 (FSE 2016)*. ACM, New York, NY, USA, 810–821. https://doi.org/10.1145/2950290.2950310

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[15] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. Springer-Verlag, Berlin, Heidelberg, 52–71. http://dl.acm.org/citation.cfm?id=648063.747438

[16] Edmund M Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT press.

[17] Matthew Conover. Feb. 2019 (last accessed). Double Free Vulnerabilities. https://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-1.

[18] 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 (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 151–162. http://dl.acm.org/citation.cfm?id=1924943.1924954

[19] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting Null-pointer Dereferences in Concurrent Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 47, 11 pages. https://doi.org/10.1145/2393596.2393651

[20] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[21] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (January 2012).

[22] Yu Guo, Yan Cai, and Zijiang Yang. 2017. AtexRace: Across Thread and Execution Sampling for In-house Race Detection. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 315–325. https://doi.org/10.1145/3106237.3106242

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

[24] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2018. Razzer: Finding Kernel Race Bugs through Fuzzing. In *Razzer: Finding Kernel Race Bugs through Fuzzing*. IEEE, 0.

[25] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. Fast and Accurate Static Data-race Detection for Concurrent Programs. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 226–239. http://dl.acm.org/citation.cfm?id=1770351.1770386

[26] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[27] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[28] L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[29] Dyoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-After-Free with Dangling Pointers Nullification. In *Network and Distributed System Security Symposium (NDSS)*.

[30] Brandon Lucia and Luis Ceze. 2009. Finding Concurrency Bugs with Context-aware Communication Graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 553–563. https://doi.org/10.1145/1669112.1669181

[31] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[32] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: Effective Sampling for Lightweight Data-race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 134–143. https://doi.org/10.1145/1542476.1542491

[33] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018, to appear. What Happens-After the First Race? Enhancing the Predictive Power of Happens-Before Based Dynamic Race Detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'18)*. https://arxiv.org/abs/1808.00185

[34] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 446–455. https://doi.org/10.1145/1250734.1250785

[35] 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 (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 267–280. http://dl.acm.org/citation.cfm?id=1855741.1855760

[36] A. Muzahid, N. Otsuki, and J. Torrellas. 2010. AtomTracker: A Comprehensive Approach to Atomic Region Inference and Violation Detection. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. 287–297. https://doi.org/10.1109/MICRO.2010.32

[37] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective Static Race Detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/1133981.1134018

[38] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 22–31. https://doi.org/10.1145/1250734.1250738

[39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[40] Chang-Seo Park and Koushik Sen. 2008. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 135–145. https://doi.org/10.1145/1453101.1453121

[41] Eli Pozniansky and Assaf Schuster. 2003. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*. ACM, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

[42] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2006. LOCKSMITH: Context-sensitive Correlation Analysis for Race Detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 320–331. https://doi.org/10.1145/1133981.1134019

[43] K. Qiu, Z. Zheng, K. S. Trivedi, and B. Yin. 2019. Stress Testing With Influencing Factors to Accelerate Data Race Software Failures. *IEEE Transactions on Reliability* (2019), 1–19. https://doi.org/10.1109/TR.2019.2895052

[44] Snajay Rawat, Vivek Jain, Ashish Kumar, Lucain Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium (NDSS)*.

[45] 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

[46] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE Symposium on Security and Privacy*.

[47] Traian Florin Serbanuta, Feng Chen, and Grigore Rosu. 2012. Maximal Causal Models for Sequentially Consistent Systems. In *Proceedings of the third International Conference on Runtime Verification (RV'12) (LNCS)*, Vol. 7687. Springer, 136–150. https://doi.org/10.1007/978-3-642-35632-2_16

[48] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[49] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09)*. ACM, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[50] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[51] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[52] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy (S&P)*.

[53] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. 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 (ESEC-FSE '07)*. ACM, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

[54] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM SIGMOD International Conference on Mangement of Data*.

[55] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attack. In *4th USENIX Workshop on Hot Topoics in Parallelism (HOTPAR)*.

[56] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=2342788.2342803

[57] 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 (OOPSLA '12)*. ACM, New York, NY, USA, 485–502. https://doi.org/10.1145/2384616.2384651

[58] Michal Zalewski. Feb. 2019 (last accessed). American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[59] S. Zhao, R. Gu, H. Qiu, T. O. Li, Y. Wang, H. Cui, and J. Yang. 2018. OWL: Understanding and Detecting Concurrency Attacks. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 219–230. https://doi.org/10.1109/DSN.2018.00033