Detecting Concurrency Vulnerabilities Based on Partial Orders of Memory and Thread Events

Kunpeng Yu
Chenxu Wang*
yukunpeng33@stu.xjtu.edu.cn
cxwang@mail.xjtu.edu.cn
School of Software Engineering
MOE Key Lab of INNS, Xi'an Jiaotong University
Xi'an, Shaanxi, China

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

Xiapu Luo
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Zijiang Yang
Xi'an Jiaotong University
Xi'an, Shaanxi, China
yang@guardstrike.com

ABSTRACT

Memory vulnerabilities are the main causes of software security problems. However, detecting vulnerabilities in multi-threaded programs is challenging because many vulnerabilities occur under specific executions, and it is hard to explore all possible executions of a multi-threaded program. Existing approaches are either computationally intensive or likely to miss some vulnerabilities due to the complex thread interleaving. This paper introduces a novel approach to detect concurrency memory vulnerabilities based on partial orders of events. A partial order on a set of events represents the definite execution orders of events. It allows constructing feasible traces exposing specific vulnerabilities by exchanging the execution orders of vulnerability-potential events. It also reduces the search space of possible executions and thus improves computational efficiency. We propose new algorithms to extract vulnerability-potential event pairs for three kinds of memory vulnerabilities. We also design a novel algorithm to compute a potential event pair's feasible set, which contains the relevant events required by a feasible trace. Our method extends existing approaches for data race detection by considering that two events are protected by the same lock. We implement a prototype of our approach and conduct experiments to evaluate its performance. Experimental results show that our tool exhibits superiority over state-of-the-art algorithms in both effectiveness and efficiency.

CCS CONCEPTS

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

*Corresponding author

KEYWORDS

concurrency vulnerability, multi-threaded programs, partial orders

ACM Reference Format:
Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. 2021. Detecting Concurrency Vulnerabilities Based on Partial Orders of Memory and Thread Events. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468572

1 INTRODUCTION

Programs written in C/C++ are generally memory-efficient and computationally fast because they allow direct memory access without additional safety checks. However, this flexibility sometimes could cause severe vulnerabilities due to wrong memory access. For example, a dereference to a null pointer will cause undefined behaviors in C/C++ programs, which usually leads to severe security problems [25, 29]. Technicians have developed several tools to detect these memory-related errors [30, 33, 35].

With the development of multi-core processors, more and more concurrency (multi-threaded) programs are developed for computational efficiency. When a multi-threaded program runs, operations in various threads are interleaved in unpredictable orders, subject to the constraints imposed by explicit synchronization operations. Although operations in each thread are strictly ordered, the interleaving of operations from a collection of threads is non-deterministic and depends on the program's execution. That is, operations in concurrency programs have various execution orders, and some vulnerabilities only occur in specific thread interleaving. For example, the execution order of two events accessing the same memory address by different threads may be exchangeable because of thread interleaving. If a `read` event to a variable occurs unexpectedly before a `write` one, an error may be caused in some situations [4]. However, it is challenging to detect concurrency memory vulnerabilities due to non-deterministic thread interleaving. Besides, it is computationally intensive to explore all the possibilities of thread interleaving, leading to the efficiency problem in concurrency vulnerability detection.

In this paper, we focus on detecting three kinds of concurrency vulnerabilities, including Use-After-Free (UAF), Double-Free (DF), and Null-Pointer-Dereference (NPD), which are mostly caused by wrong memory-address accesses in stochastic executions [28, 34]. A UAF occurs when a dangling pointer accesses a freed memory [2]. When a program attempts to dereference a NULL pointer, an NPD occurs. DF means that a memory address is improperly freed twice [2]. It is easy to figure out these vulnerabilities in a single-threaded program because all events occur in a definite sequence. However, it is non-trivial to detect these vulnerabilities in multi-threaded programs because only certain execution orders can trigger them.

There are two main approaches to detect concurrency vulnerabilities. An intuitive solution is to cooperate with the *fuzzing* technology by running a target program many times and try to retrieve different execution orders exhaustively [5, 15, 19, 36, 37]. Muzz [5] is a grey-box fuzzer that uses thread-aware instrumentation to get more various execution orders by adjusting thread priorities. Krace [37] is a file system fuzzer aiming to detect data race among multiple threads. It controls thread schedules by delaying a memory access for a random number of accesses in other threads. However, it is extremely expensive to get all possible execution orders due to the explosion of search spaces [22].

Another way is to predict possible executions from observed ones. Huang's UAF Finder Optimal (UFO) [13] is underpinned by a data race detection model based on constraint solving algorithms. UFO relies on the constraint solver Z3 [7], which is computationally intensive for complex constraints. ExceptioNULL [11] rearranges the trace to expose NPD vulnerabilities. It solves several constraints to guarantee the feasibility of the constructed traces. The scheduling constraints are employed to represent necessary execution orders of a multi-threaded program, and then Z3 is adopted to solve the constraints. However, this approach may cost a mass of time to solve complicated constraints. ConVul [4] uses a heuristic method based on *sync-edge* and *sync-distance* to determine whether two events are exchangeable. A sync-edge is an edge either from a lock event to its unlock event in the same thread or from an unlock event to a lock event on the same lock in the other thread. The sync-distance of two events is defined as the minimal number of sync-edges ordering the two events [4]. ConVul generates candidate traces that trigger vulnerabilities by exchanging the execution orders of events. It assumes that two events with short sync-distance are more likely to be exchangeable for computational efficiency improvement. If the sync-distance of two events is less than a predefined threshold (default is 3), ConVul schedules the trace and tries to exchange their execution order for vulnerability detection. However, it may miss some exchangeable events with long sync-distances. In summary, there is a lack of effective and efficient methods to detect concurrency vulnerabilities.

This paper presents ConVulPOE, a Concurrency Vulnerability detector based on Partial Orders of Events. Similar to ConVul [4] and UFO [13], ConVulPOE attempts to generate execution traces exposing vulnerabilities. However, ConVulPOE follows constructed partial orders of memory and thread events to generate traces. First, we record the concerned thread operations and memory access events in an execution of a program through Intel Pin [20]. We represent the sequence of events as a trace. Second, we develop new algorithms to extract vulnerability-potential event pairs that may cause vulnerabilities by passing through the trace. We also design a novel algorithm to compute an event pair's feasible set, which contains the relevant events required by a feasible trace exposing a vulnerability.

Third, we construct a partial order for a feasible event set to generate a new trace exposing a vulnerability. The partial orders of an event set represent the deterministic execution orders between contextual events. Compared with the heuristic algorithm used in ConVul [4], following a partial order of events to generate a trace allows us to detect vulnerabilities without false positive errors. It also reduces the search spaces of possible executions, which improves computational efficiency.

Finally, we implement a prototype of ConVulPOE for pthread-based multi-threaded C/C++ programs based on Intel Pin. We conduct experiments to evaluate the performance of ConVulPOE. Experimental results show that our tool can find more vulnerabilities than state-of-the-art detectors in less time.

2 PRELIMINARIES

A multi-threaded program $\mathcal{P}$ consists of various threads $\{t_1, t_2, \ldots, t_n\}$, where n is the number of threads. Each thread has a sequence of operations under sequential consistency semantics [17]. Operations in different threads are interleaved randomly as the program runs. Like prior works [4, 13, 23, 26], this paper focuses on variable and lock accessing operations because most concurrency vulnerabilities are caused by improper synchronization on variables. We refer to these operations as *events*. An access to a variable is a `read` or `write` event. We use $r(x)$ and $w(x)$ to denote a `read` and `write` event on variable x, respectively. Operations on a lock include `acquire` and `release`. Similarly, we employ $acq(l)$ and $rel(l)$ to denote an `acquire` and `release` event on lock l, respectively.

We use a sequence of events to represent a program's execution and call it *trace*. Events in a trace are ordered sequentially. Given a trace σ, we use $\mathcal{E}_\sigma$ to denote the set of events in σ and $\mathcal{W}(\mathcal{E}_\sigma)$ (resp., $\mathcal{R}(\mathcal{E}_\sigma)$, $\mathcal{L}^A(\mathcal{E}_\sigma)$, $\mathcal{L}^R(\mathcal{E}_\sigma)$) to denote the `write` (resp., `read`, `acquire`, `release`) event set of σ. Given an event e, we use $tid(e)$ to denote the thread that executes e and $loc(e)$ to denote the variable (or lock) that e accesses. If event e is an `acquire` (`release`) event, we use $match_\sigma(e)$ to denote the first `release` (`acquire`) event accessing the same lock following (previous to) e in the same thread.

Given a pointer-type variable x, we use $free(x)$ to denote an event freeing x, and $deref(x)$ to denote the `dereference` operation of x. Note that $free(x)$ and $deref(x)$ are two special cases of $read(x)$. Given a trace σ, we use $Free(\mathcal{E}_\sigma)$ and $Deref(\mathcal{E}_\sigma)$ to denote the `free` and `dereference` event sets of σ, respectively.

The following presents two critical definitions relevant to the execution of concurrency programs.

Critical sections: Given a lock l, a critical section is the set of sequential events in the same thread starting from $acq\ (l)$ and ending with $rel\ (l) = match_\sigma\ (acq\ (l))$ (including $acq(l)$ and $rel(l)$).

Conflicting events: Two events are said to be conflicting if they belong to different threads and access the same variable, and at least one of them is a `write` event. We also say that two lock accessing events are conflicting if they access the same lock. We use $e_1 \asymp e_2$ to denote that e_1 and e_2 are conflicting.

Both critical sections and conflicting events can be got by passing through a trace once.

2.1 Memory Vulnerabilities

C/C++ programs adopt a series of functions or operators (e.g., `malloc`, `free`, `new`, `delete`) for dynamic memory management. A block of heap memory must be allocated before it is accessed and freed after use. Accesses to a memory address that is not allocated yet or previously freed will lead to undefined behaviors of a program and usually cause crashes and security problems. This paper focuses on three kinds of concurrency memory vulnerabilities, including Use-After-Free (UAF), Null-Pointer-Dereference (NPD), and Double-Free (DF).

Use-After-Free (UAF): A UAF occurs when a block of previously freed memory is used. It can cause serious consequences, ranging from corruptions of valid data to the execution of arbitrary codes [9].

NULL-Pointer-Dereference (NPD): An NPD occurs when a NULL pointer is dereferenced. It typically causes a crash or an abnormal exit. In rare circumstances, it also leads to the execution of arbitrary codes [10].

Double-Free (DF): A DF occurs if a program calls `free` twice with the same argument (i.e., the memory address). It can cause programs' crashes or make them vulnerable to buffer overflow attacks under specific conditions [8].

These vulnerabilities are caused by several flaws, such as race conditions and other exceptional circumstances. The non-deterministic execution of multi-threaded programs makes these vulnerabilities more frequent to occur and harder to be detected.

2.2 Feasible Traces

Not all reordered traces of an observed one are valid and feasible. Even a few changes to the events' execution orders may cause the program to execute other branches. Hence, we cannot use an arbitrarily reordered trace to represent an execution. Every `read` event should get the same value as it did in the original trace to ensure the validity of the reordered trace. Given a trace σ and a `read` event r, we say that r observes a `write` event w in σ if w is the nearest event before r that writes the same variable. We use $obs_\sigma(r) = w$ to denote that r observes w. We have $obs_\sigma(r) = w$, iff $loc(r) = loc(w)$ and $w <_\sigma r$, and there is no other `write` event w' that $loc(w') = loc(r)$ and $w <_\sigma w' <_\sigma r$. For simplicity, we assume that every variable has a `write` event initializing it. We first define the projection of a trace onto a thread and then define feasible traces.

Projection: Given a trace σ and a thread t, the projection of σ onto thread t is defined as the event sequence of t and is denoted by $\sigma\upharpoonright_t$. That is, $\sigma\upharpoonright_t$ is a sub-sequence of σ satisfying that:

(1) $\sigma\upharpoonright_t$ only contains the events of thread t. That is, $\mathcal{E}_{\sigma\upharpoonright_t} \subseteq \mathcal{E}_\sigma$ and $\forall e \in \mathcal{E}_{\sigma\upharpoonright_t}, tid(e) = t$.
(2) Event orders in $\sigma\upharpoonright_t$ is the same as that in σ. That is, $\forall (e_1, e_2 \in \mathcal{E}_\sigma \wedge tid(e_1) = tid(e_2) = t \wedge e_1 <_\sigma e_2), e_1 <_{\sigma\upharpoonright_t} e_2$.

Given an observed trace σ, we say another trace σ' is *feasible* if the following properties hold:

(1) Trace σ' only contains the events in σ. That is, $\mathcal{E}_{\sigma'} \subseteq \mathcal{E}_\sigma$. σ' need not contain all the events in σ because we just want the program to execute to the point where a vulnerability occurs and do not care about what happens after it.
(2) Events in the same thread must execute in the same order in σ and σ'. That is, for every thread t, $\sigma'\upharpoonright_t$ is a prefix of $\sigma\upharpoonright_t$.
(3) Critical sections on the same lock cannot overlap. That is, given two `acquire` events acq_1 and acq_2, and $loc(acq_1) = loc(acq_2)$, if $acq_1 <_{\sigma'} acq_2$, then $match_\sigma(acq_1) <_{\sigma'} acq_2$.
(4) Every `read` event r should observe the same event in σ' as it does in σ. That is, $\forall r \in \mathcal{R}(\mathcal{E}_{\sigma'}), obs_{\sigma'}(r) = obs_\sigma(r)$.

2.3 Partial Orders

Multi-threaded programs usually have different execution orders due to non-deterministic thread interleaving. The execution orders of some events could be exchanged in different runs of a multi-threaded program. In this paper, we are concerned about exchangeable event pairs because they are more likely to cause memory vulnerabilities. Given two events e_1 and e_2, we say that (e_1, e_2) is exchangeable if two valid traces σ and σ' exist, and the execution orders of e_1 and e_2 in the two traces are different (i.e., $e_1 <_\sigma e_2 \wedge e_2 <_{\sigma'} e_1$).

Some events must execute before others to make a reordered trace feasible. We use partial orders to represent the execution orders of events. Given an event set E, a (strict) partial order $P(E)$ is an irreflexive, transitive, and asymmetry relation over E. If E is clear from the context, we will use P for simplification. There are two basic partial orders over $\mathcal{E}_\sigma$ for a trace σ. The first is the execution order ($<_\sigma$) we mentioned above, which represents an execution of a program. We denote the execution order by $tr(\mathcal{E}_\sigma)$, which is determined by the trace σ. The execution of a single thread can also be represented by a partial order. If event e_1 is performed before event e_2 in σ, and both events are in the same thread, we say e_1 is *thread-ordered* before e_2, denoted by $e_1 <_{\sigma\upharpoonright_t} e_2$. We denote the thread order by $TO(\mathcal{E}_\sigma)$. Note that any two events of σ are ordered by $tr(\mathcal{E}_\sigma)$, but only events in the same thread are ordered by $TO(\mathcal{E}_\sigma)$. Given a partial order P, two events e_1, e_2 are *unordered*, denoted by $e_1 \parallel_P e_2$, if neither $e_1 <_P e_2$, nor $e_2 <_P e_1$. For example, any two events in different threads are unordered by $TO(\mathcal{E}_\sigma)$. On the other hand, if a trace σ' is feasible, events in the same thread must execute in the same order in σ. That is, given $e_1, e_2 \in \mathcal{E}_\sigma$ and $e_1, e_2 \in \mathcal{E}_{\sigma'}$, if $e_1 <_{\sigma\upharpoonright_t} e_2$, then $e_1 <_{\sigma'\upharpoonright_t} e_2$.

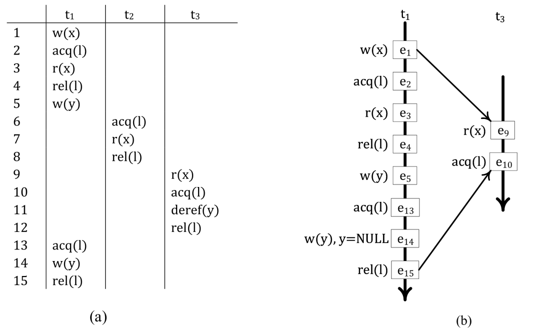
3 FEASIBLE TRACE CONSTRUCTION

In this section, we first give an example to illustrate the motivation of our method. Then, we present the algorithm to generate new feasible traces based on an observed one.

3.1 A Motivating Example

Figure 1(a) shows a trace containing three threads. We use e_i to denote the i-th event. Let e_{14} assign NULL to y. Note that e_{11} is a pointer-dereference event, and (e_{11}, e_{14}) is an event pair protected by the same lock l. If e_{14} executes before e_{11}, then an NPD occurs. To prove that, we need to reorder these events and get a new trace in which e_{11} is not included to make e_{14} execute first. Since the new trace is feasible and includes the event right before e_{11}, we can append e_{11} to the end of the new trace and keep it feasible.

The first step is to figure out what events should be included in the new trace. For example, the events in t_2 are not needed in the new trace because there is no `write` event, and other events need not observe them. Therefore, the event set should be $E = \{e_i\}_{i=1}^{5} \cup \{e_i\}_{i=9}^{10} \cup \{e_i\}_{i=13}^{15}$. Events in the same thread have already been ordered. To get a feasible trace, we need to make some events in different threads reordered. We introduce partial order to solve the problem. Figure 1(b) is a partial order graph constructed based on E. We can see that there are two additional edges from e_1 to e_9 and from e_{15} to e_{10}. The first edge makes e_9 read the value written by e_1 as it does in the origin trace. And the second one makes the critical sections on lock l not overlap. Then we can get a feasible trace $e_1, e_9, e_2, e_3, e_4, e_5, e_{13}, e_{14}, e_{15}, e_{10}, e_{11}$, where e_{14} executes before e_{11}. Thus, we can prove that there is an NPD vulnerability.



	t1	t2	t3
1	w(x)		
2	acq(l)		
3	r(x)		
4	rel(l)		
5	w(y)		
6		acq(l)	
7		r(x)	
8		rel(l)	
9			r(x)
10			acq(l)
11			deref(y)
12			rel(l)
13	acq(l)		
14	w(y)		
15	rel(l)		

Figure 1: An example of a trace and one of its partial orders.

In the rest of this section, we first define a feasible event set for an execution trace of a multi-threaded program. A feasible event set contains necessary events that build a new trace. Then, we build a partial order based on the feasible event set to add some ordering between events. Finally, we present the algorithm to generate new feasible traces.

3.2 Feasible Event Sets

As we mentioned above, a feasible trace σ' need not contain all the events in the original trace σ. Besides, not all partial orders can produce a feasible trace. The first problem of getting a feasible trace from a partial order is to extract a subset of $\mathcal{E}_\sigma$. We say that a subset of $\mathcal{E}_\sigma$ is *feasible* if it is prefix-closed, lock-feasible, and observation-feasible for σ.

Prefix-closed: For every event pair $e_1, e_2 \in \mathcal{E}_\sigma$, we say that $\mathcal{E}_{\sigma'}$ is prefix-closed if $e_2 \in \mathcal{E}_{\sigma'}$ and $e_1 <_{\sigma\restriction_t} e_2$, then $e_1 \in \mathcal{E}_{\sigma'}$. It ensures that $\sigma' \restriction_t$ is a prefix of $\sigma \restriction_t$ for every thread t. We can see that the event set in Figure 1(b) is prefix-closed.

Lock-feasible: We say that $\mathcal{E}_{\sigma'}$ is lock-feasible if (i) $\forall rel \in \mathcal{L}^R_{\sigma'}$, $match_\sigma(rel) \in \mathcal{E}_{\sigma'}$, and (ii) $\forall acq_1, acq_2 \in OpnAcq_\sigma(\mathcal{E}_{\sigma'})$, $loc(acq_1) \neq loc(acq_2)$; where $OpnAcq_\sigma(\mathcal{E}_{\sigma'}) = \{acq \in \mathcal{L}^A_{\sigma'} | match_\sigma(acq) \notin \mathcal{E}_{\sigma'}\}$ is an event set that contains open `acquire` events whose matching `release` events do not belong to $\mathcal{E}_{\sigma'}$. The first condition guarantees that the matching `acquire` event for each `release` event in $\mathcal{E}_{\sigma'}$ is also in $\mathcal{E}_{\sigma'}$. The second ensures that there is at most one open `acquire` event for every lock. In Figure 1(b), $match_\sigma(e_{15}) = e_2$ and e_{10} is the only open acquire event. So the event set is lock-feasible.

Observation-feasible. We call $\mathcal{E}_{\sigma'}$ is observation-feasible if $\forall r \in \mathcal{R}(\mathcal{E}_{\sigma'})$, $obs_\sigma(r) \in \mathcal{E}_{\sigma'}$. It guarantees that there is a `write` event writing the value read by later events. In Figure 1(b), e_1 writes the variable x, which is read by later events e_3 and e_9. So the event set is observation-feasible.

3.3 Trace-Closed Partial Orders

In this paper, we use partial orders to represent the execution order of events. Given two partial orders P and Q over an event set E, we say that Q *refines* P, if $\forall(e_1, e_2 \in E \bigwedge e_1 <_P e_2)$, then $e_1 <_Q e_2$. We denote it as $Q \sqsubseteq P$ and say that P is *weaker* than Q. A weaker partial order means less constraints on the execution orders of events, and we can construct more possible traces based on it. We give the following conditions to construct a partial order required by a feasible trace. The insight here is to first construct a basic partial order that makes some events ordered. And then, considering the newly ordered events, we add new ordering between two events incrementally to meet all the constraints.

Given a trace σ and a feasible event set $E \subseteq \mathcal{E}(\sigma)$, we use P to represent a partial order over E that makes a trace feasible. First, $P \sqsubseteq TO(E)$ ensures that every event pair in the same thread is ordered by P. Then, for every `release` event $rel \in \mathcal{L}^R(E)$, if $OpnAcq_\sigma(E) \neq \emptyset$, then for each event $acq \in OpnAcq_\sigma(E)$ and $loc(acq) = loc(rel)$, we have $rel <_P acq$. It ensures that every open `acquire` event is the last one that operates the lock. Otherwise, the lock will not be released. Finally, for every `read` event $r \in \mathcal{R}(E)$, we have $obs_\sigma(r) <_P r$. It ensures that every `read` event executes after the `write` event it observes in trace σ. We say that such a partial order *respects* trace σ and use $Rs_\sigma(E)$ to denote the weakest partial order respecting σ. The trace-respecting partial order P offers basic constraints on execution orders between event pairs required by a feasible trace.

Let σ be a trace, $E \subseteq \mathcal{E}(\sigma)$ be a feasible event set, and P be a partial order over E that respects σ. To ensure that the critical sections on the same lock do not overlap, for every `release` event pair $rel_1, rel_2 \in \mathcal{L}^R(E)$, let $acq_1 = match_\sigma(rel_1)$ and $acq_2 = match_\sigma(rel_2)$, if $acq_1 <_P rel_2$ and $loc(rel_2) = loc(acq_1)$, we have $rel_1 <_P acq_2$ (see Figure 2(a)). We say that such a partial order is *lock-closed.* To ensure that every `read` event observes the same event as it does in σ, for every `read` event $r \in \mathcal{R}(E)$, let $w = obs_\sigma(r)$, and for every `write` event $w' \in \mathcal{W}(E) \setminus \{w\}$ such that $w' \asymp r$, we have that (i) if $w' <_P r$, then $w' <_P w$ (see Figure 2(b)) and (ii) if $w <_P w'$, then $r <_P w'$ (see Figure 2(c)). We say that such a partial order is *observation-closed.* The two rules make w the nearest event writing the same variable as r reads. A partial order is *trace-closed* (or simply *closed*) if it is lock-closed and observation-closed. We define the closure of P as the weakest partial order P' which is closed and refines P. That is, $P \sqsubseteq P'$ and there is no closed partial order Q such that $P' \sqsubseteq Q$. If P' does not exist, P has no closure.

3.4 Trace Construction Algorithm

Algorithm 1 presents the pseudo-code of constructing a feasible trace from a feasible event set or deducing its nonexistence. The inputs include a trace σ and a feasible event set $E \subseteq \mathcal{E}(\sigma)$. We represent a partial order P as a graph $\mathcal{G}$ where nodes are events,

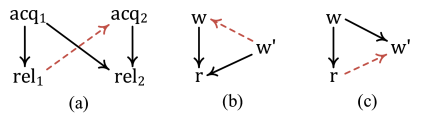


Figure 2: Illustrations of lock and observation-closed partial orders. Solid arrows are observed execution orders and dashed ones are partial orders added to make a trace closed.

and edges represent their execution orders. Figure 3 shows an example of a partial order graph. The indices of events in Figure 3 represent the execution order in trace σ, i.e., if $i < j$, $e_i <_\sigma e_j$. The thick arrows throughout the events of each thread are thread-order edges. Initially, we construct P over E (line 1) by connecting every thread's events. Then, we insert edges to $\mathcal{G}$ to make P lock-closed and observation-closed (line 2). The solid line arrows are observation edges ($e_1 \rightarrow e_7$, $e_5 \rightarrow e_{10}$), and the dashed ones are lock edges ($e_4 \rightarrow e_9$). If a cycle is formed, it means that conflicts occur among the constraints to construct a trace-closed partial order. That is, P has no closure, and the algorithm returns $\perp$ (lines 3-4). Note that a closed partial order may still contain unordered conflicting events, preventing us from getting a feasible trace. If we get a closed partial order Q, for every conflicting event pair, we insert an edge between the two events to Q. Note that after inserting the new edge, Q may not be closed (see Figure 2). We need to insert additional edges and get a new closure of Q (lines 5-8). We make two conflicting events keep their original execution order (lines 6-7). The insight here is that it is mostly like to avoid forming a cycle [26], although it may fail in some situations [23]. If no cycle is formed, we build a feasible trace from Q. Given an event e and a thread i, if $\exists e' \in E, tid(e') = i, e' <_Q e$ and $\nexists e'' \in E$ such that $tid(e'') = i$, $e'' <_Q e$ and $e' <_Q e''$, then e' is a predecessor of e in thread i. For example, as shown in Figure 3, the predecessors of e_8 in t_1 and t_2 are e_1 and e_7, respectively, and e_8 has no predecessor in t_3. Initially, T is an empty trace. If an event e has no predecessor in Q or its predecessors are already in T, we add e to the new trace T and remove it from E (lines 11-14).

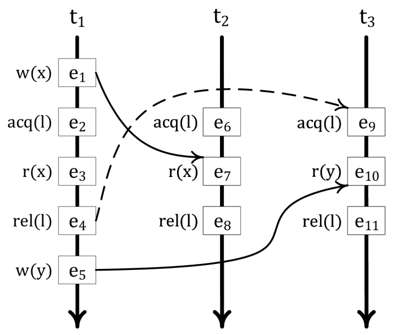


Figure 3: An example of a partial order graph. Solid lines are observation edges and dashed lines are lock edges.

Algorithm 1: ConstructFeasibleTrace

```
Input: A trace $\sigma$, and a feasible set $E$
Output: A feasible trace if it exists, otherwise $\perp$
1  $P \leftarrow Rs_\sigma(E)$
2  $Q \leftarrow$ GetClosure($t, P, E$)
3  if $Q = \perp$ then                         // A cycle is formed
4      return $\perp$
5  while $\exists e_1, e_2 \in E$ s.t. $e_1 \asymp e_2$ and $e_1 \parallel_Q e_2$ do
6      if $e_1 >_{tr} e_2$ then
7          swap($e_1, e_2$)
8      $Q \leftarrow$ InsertAndGetClosure($Q, e_1 \rightarrow e_2$)
9      if $Q = \perp$ then                     // A cycle is formed
10         return $\perp$
11 Let $T$ be an empty list
12 while $\exists e \in E$ s.t. $e$ has no predecessor in $Q$ or its predecessors
     have been added to $T$ do
13     $T$.append($e$)
14     $E \leftarrow E \backslash \{e\}$                        // Remove e
15 return $T$
```

4 VULNERABILITY DETECTION

This section describes the approach to construct a feasible trace exposing a concurrency vulnerability. We first extract vulnerability-potential event pairs and then compute their feasible event sets. Then we attempt to construct a feasible trace by Algorithm 1.

4.1 Potential Event Pairs

Given a trace σ, and an exchangeable event pair $e_1, e_2 \in \mathcal{E}_\sigma$, if $e_1 <_\sigma e_2, tid(e_1) \neq tid(e_2)$, and the exchange of their execution order may cause a vulnerability (e.g., if e_1 writes a memory address and then e_2 frees the memory, the exchange of their execution may cause an UAF), we call them a *vulnerability-potential* event pair (or simply *potential* pair). The following describes the algorithms to get the potential pairs of UAF, NPD, and DF.

4.1.1 UAF. A UAF occurs when a block of previously freed memory is used. Figure 4 shows how thread-interleaving causes a UAF. There are two events in Figure 4(a). Pointer p is dereferenced (①) first in thread t_1 and is then freed (②) in thread t_2. If their execution order is exchanged, and p is freed first in t_2, then an UAF occurs when it is being dereferenced in t_1. Figure 4(b) shows other cases where a UAF occurs. There are three events in two threads. In Figure 4(b), a pointer is freed (①) first, and then a not NULL value q is written to it (②), at last it is dereferenced (④). In this case, if $p = q$ executes after ④, when p is dereferenced (④) after it is free (①), then an UAF occurs. Note that $p = q$ may also be in other threads, making the problem more complex. For example, let $p = q$ in thread t_1, the original trace is ①, ③, ④, if there is a trace ③, ①, ④, then an UAF occurs. Therefore, event pairs (①, ②) in Figure 4(a) and (①, ③) or (②, ④) in Figure 4(b) are UAF vulnerability-potential pairs.

Algorithm 2 shows the pseudo-code to get potential event pairs in a trace σ that may cause UAFs. Given a trace σ and an event e, we use the function $Before^{\mathcal{R}}(e, i)$ ($Before^{\mathcal{W}}(e, i)$, $Before^{Free}(e, i)$)

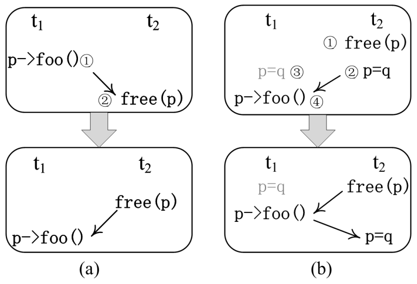


Figure 4: Illustrations of how UAFs occur. The gray events are other possible events in different thread.

or $Before^{Deref}(e, i)$) to denote the first read (write, free or dereference) event e' before e that $tid(e') = i$ and $loc(e') = loc(e)$. First, for every free event $e_{free} \in Free(\mathcal{E}_\sigma)$ on a pointer, we get the first dereference e_{deref} event before e_{free} on the same pointer in each thread i. If the event exists and $i \neq tid(e_{free})$, we add $\left(e_{deref},\ e_{free}\right)$ to the potential pair set S (lines 2-7). Then for every dereference event $e_{deref} \in Deref(\mathcal{E}_\sigma)$ on a pointer and every thread i, we get the first write event w before e_{deref} on the same pointer in thread i (lines 8-10). If w does not write NULL, then for every thread j, we get the first free event e^j_{free} in the thread that frees the same pointer before w such that $\nexists w' \in \mathcal{W}(\mathcal{E}_\sigma)$, $tid(w') = j$ and $e^j_{free} <_\sigma w' <_\sigma w$ (lines 11-17). If there is no such an event e^j_{free} for every thread j, it continues (line 18). Otherwise, we add $\left(w, e_{deref}\right)$ to S if $tid(w) \neq tid(e_{deref})$ (lines 19-20). For every thread j and e^j_{free} (if exists), if $j \neq tid(w)$, we add $\left(e^j_{free}, w\right)$ to S (lines 21-23).

4.1.2 NPD. An NPD occurs when a NULL pointer is dereferenced. As shown in Figure 5(a), a pointer p is dereferenced first and then set to be NULL. If the execution order of them is exchanged, then an NPD occurs. Like Figure 4(b), Figure 5(b) shows how three events cause an NPD. If the execution order of the two write events (①, ③) is exchanged, then an NPD occurs. And if the dereference event (④) executes before the write event that sets p a new value (②), an NPD also occurs. The difference between Figure 4(b) and 5(b) is that the first event in Figure 4(b) is a free event while in Figure 5(b), the first event is a write event and assigns NULL to p. The algorithm that gets potential NPD event pairs in a trace σ is similar to Algorithm 2. We omit it for space concerns.

4.1.3 DF. A DF occurs when a block of memory is freed twice. As shown in Figure 6, a pointer is assigned with a new value (②) after being freed (①) and then be freed once again (④). The exchanges of ① and ② may cause an DF. Note that the assigning instruction can also be in thread t_1 (see ③ in Figure 6), which make the situation more complex. Algorithm 3 shows the pseudo-code to get potential event DF event pairs. For every free event $e_{free} \in Free(\mathcal{E}_\sigma)$ on a pointer and every thread i, we get the first write event w on the same pointer before e_{free} in thread i (lines 2-5). If w does not write NULL, we get the free events $e^j_{free'}$ in every thread j that frees

Algorithm 2: GetPotentialUAF

```
   Input: A trace $\sigma$
   Output: Event pairs that may cause UAFs
1  Let $S$ be an empty set
2  foreach free event $e_{free} \in Free(\mathcal{E}_\sigma)$ do
3      foreach thread $i$ do
4          if $i \neq tid(e_{free})$ then
5              $e_{deref} \leftarrow Before^{Deref}(e_{free}, i)$
6              if $e_{deref} \neq \perp$ then
7                  $S \leftarrow S \cup \{(e_{deref}, e_{free})\}$
8  foreach dereference event $e_{deref} \in Deref(\mathcal{E}_\sigma)$ do
9      foreach thread $i$ do
10         $w \leftarrow Before^{\mathcal{W}}_{loc(e_{deref})}(e_{deref})$
11         if $w \neq \perp$ then
12             if $w.getWriteValue() = NULL$ then continue;
13             let $frees$ be an empty set
14             foreach thread $j$ do
15                 $e_{free} \leftarrow Before^{Free}(w, j)$
16                 if $e_{free} \neq \perp$ and there is no $w' \in$
                     $\mathcal{W}(\mathcal{E}_\sigma), tid(w') = j \wedge e_{free} <_{tr} w' <_{tr} w$
                     then
17                     $frees \leftarrow frees \cup \{e_{free}\}$
18             if $frees.isEmpty()$ then continue ;
19             if $tid(w) \neq tid(e_{deref})$ then
20                 $S \leftarrow S \cup \{(w, e_{deref})\}$
21             foreach $e_{free} \in frees$ do
22                 if $tid(e_{free}) \neq tid(w)$ then
23                     $S \leftarrow S \cup \{(e_{free}, w)\}$
24 return S
```

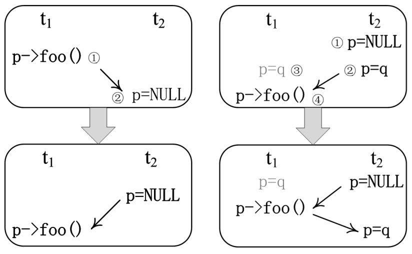


Figure 5: Illustrations of how NPDs occur. The gray events are other possible events in different thread.

the same pointer before w such that $\nexists w' \in \mathcal{W}(\mathcal{E}_\sigma)$, $tid(w') = j$ and $e^j_{free'} <_{tr} w' <_{tr} w$ (lines 6-11). If there is no such an event $e^j_{free'}$ for every thread j, we continue (lines 12). Otherwise, we add $\left(w, e_{free}\right)$ to the potential pair set S if they are not in the same thread (lines 13-14). And for every thread j and $e^j_{free'}$ (if exists), if $j \neq tid\,(w)$, then we add $\left(e^j_{free'}, w\right)$ to S (lines 15-17).

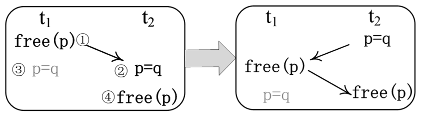


Figure 6: Illustrations of how DFs occur. The gray events are other possible events in different thread.

```
Algorithm 3: GetPotentialDF
  Input: A trace $\sigma$
  Output: Event pairs that may cause DFs
1 let $S$ be an empty set
2 foreach free event $e_{free} \in Free(\mathcal{E}_\sigma)$ do
3     foreach thread $i$ do
4         $w \leftarrow Before^{\mathcal{W}}(e_{free}, i)$
5         if $w \neq \perp$ then
6             if $w.getWriteValue() = NULL$ then continue;
7             let $frees$ be an empty set
8             foreach thread $j$ do
9                 $e_{free} \leftarrow Before^{Free}(w, j)$
10                if $e_{free} \neq \perp$ and there is no $w' \in \mathcal{W}(\mathcal{E}_\sigma), tid(w') = j \wedge e_{free} <_{tr} w' <_{tr} w$ then
11                    $frees \leftarrow frees \cup \{e_{free}\}$
12            if $frees.isEmpty()$ then continue ;
13            if $tid(w) \neq tid(e_{free})$ then
14                $S \leftarrow S \cup \{(w, e_{free})\}$
15            foreach $e_{free'} \in frees$ do
16                if $tid(e_{free'}) \neq tid(w)$ then
17                    $S \leftarrow S \cup \{(e_{free'}, w)\}$
18 return S
```

4.2 Computing Feasible Event Sets

Given a trace σ and a potential pair $e_1, e_2 \in \mathcal{E}_\sigma$, we first compute a feasible event set of an event pair in different threads. Then we construct a trace σ' in which e_2 executes before e_1 to prove the existence of a vulnerability. According to previous definitions, a feasible event set must be prefix-closed, lock-feasible, and observation-feasible.

Let E be the feasible set we want. Two cases need to be considered for computing a potential pair's feasible set. If the two events are not protected by the same lock, the reordered trace σ' should be ended with e_2, e_1 (i.e., e_1 is right behind e_2 and the trace is like $e, \ldots, e_2, e_1$). In this case, we can easily construct a feasible set $E = \mathcal{E}_{\sigma'} \setminus \{e_2, e_1\}$. If the two events are protected by the same lock(s), we cannot put them together at the end of a trace because of the lock semantic, i.e., e_1 must execute after that all common locks of the two events are released by the thread $tid\,(e_2)$. Let l be the last released common locks, $rel\,(l)$ be the first `release` event on l after e_2 in $tid\,(e_2)$, and $acq\,(l)$ be the first `acquire` event on l before e_1 in $tid\,(e_1)$, the reordered trace is like $e, \ldots, e_2, \ldots, rel\,(l), acq\,(l), \ldots, e_1$ and $E = \mathcal{E}_{\sigma'} \backslash \{e_1\}$. Note that E does not contain e_1 and e_2 simultaneously because it may make E not observation-feasible. If we get a feasible trace σ' from E, the trace constructed by adding e_2, e_1 or e_1 to the end of σ' is also feasible. Since there may be several additional events between e_2 and e_1 in σ', we need to check whether the vulnerability will occur in σ' at last to avoid false-positive errors.

In this paper, we first find the candidate feasible event sets of e_1 and e_2 separately and then get their union. We employ the relative causal cones of an event relative to a thread to compute the feasible set for an event [23].

Relative Causal Cones: Given a trace σ, an event pair $e_1, e_2 \in \mathcal{E}_\sigma$, let $t_1 = tid\,(e_1)$ and $t_2 = tid\,(e_2)$, the causal cone $RCone_\sigma(e_1, t_2)$ of e_1 relative to t_2 represents the smallest event set satisfying:

(1) For each $e' \in \mathcal{E}_\sigma$, if $e' <_{\sigma\restriction_{t_1}} e_1$, then $e' \in RCone_\sigma(e_1, t_2)$.
(2) For every `read` event $r \in \mathcal{R}(\mathcal{E}_\sigma) \cap RCone_\sigma\,(e_1, t_2)$, we have $obs_\sigma\,(r) \in RCone_\sigma\,(e_1, t_2)$.
(3) For every lock-acquire event $acq \in \mathcal{L}^A(\mathcal{E}_\sigma) \cap RCone_\sigma\,(e_1, t_2)$, if $tid\,(acq) \neq t_1$ and $tid\,(acq) \neq t_2$, then $match_\sigma\,(acq) \in RCone_\sigma\,(e_1, t_2)$.
(4) For every event pair $e' \in RCone_\sigma\,(e_1, t_2)$ and $e'' \in \mathcal{E}_\sigma$, if $e'' <_{\sigma\restriction_{tid(e')}} e'$, then $e'' \in RCone_\sigma\,(e_1, t_2)$.

$RCone_\sigma\,(e_1, t_2)$ is prefix-closed and observation-feasible but may not be lock-feasible (there may be open acquires on the same lock in t_1 and t_2). Algorithm 4 shows the computation of a feasible set via relative casual cones. We first consider the case that two events are not protected by the same lock. A candidate feasible set of e_1 and e_2 is $E = RCone_\sigma\,(e_1, t_2) \cup RCone_\sigma\,(e_2, t_1)$ (line 3). Rules 1, 2, and 4 ensure that E is prefix-closed and observation-feasible. Rule 3 guarantees that there are no two open acquires on the same lock. Note that this rule does not apply for the events in t_1 and t_2, because E should only contain events before both e_1 and e_2. These rules do not ensure that $e_2 \notin RCone_\sigma\,(e_1, t_2)$ and $e_1 \notin RCone_\sigma(e_2,\ t_1)$. Moreover, the union of the two causal cones is not always feasible. Hence, we must perform additional checks (lines 4-5). If e_1 and e_2 are protected by the same lock(s), let l be the last released one of the common locks (line 8), $rel\,(l)$ be the first `release` event on l after e_2 in $tid\,(e_2)$ (line 9), the candidate feasible set is $E = RCone_\sigma\,(e_1, t_2) \cup RCone_\sigma\,(rel(l), t_1) \cup \{rel(l)\}$ (line10). Similarly, we need to ensure that $e_1 \notin E$ and E are feasible (lines 11-12). Now the feasible event set of the motivating example in Figure 1 can be computed as follows: $RCone_\sigma(e_{11}, t_1) = \{e_9, e_{10}\}$, $RCone_\sigma(e_{15}, t_3) = \{e_1, e_2, e_3, e_4, e_5, e_{13}, e_{14}\}$, and the feasible set $E = \{e_i\}_{i=1}^{5} \cup \{e_i\}_{i=9}^{10} \cup \{e_i\}_{i=13}^{15}$.

5 IMPLEMENTATION

Figure 7 presents an overview of ConVulPOE, which consists of two components: trace recorder and vulnerability predictor. Trace recorder generates execution traces of a program, and vulnerability predictor outputs potential vulnerabilities by exploring the traces.

Algorithm 4: GetFeasibleSet

```
Input: A trace $\sigma$, and two conflicting events $e_1, e_2 \in \mathcal{E}_\sigma$
Output: A feasible event set if exists, otherwise $\bot$
1  $t_1, t_2 \leftarrow tid(e_1), tid(e_2)$
2  if $e_1$ and $e_2$ are not protected by the same lock then
3      $E \leftarrow RCone_\sigma(e_1, t_2) \cup RCone_\sigma(e_2, t_1)$
4      if $\{e_1, e_2\} \cap E \neq \emptyset$ or $E$ is not feasible then
5          return $\bot$
6      return $E$
7  else
8      Let $l$ be the last lock that is released of the common
         locks of $e_1$ and $e_2$
9      Let $rel(l)$ be the first lock-release event on $l$ after $e_2$ in $t_2$
10     $E \leftarrow RCone_\sigma(e_1, t_2) \cup RCone_\sigma(rel(l), t_1) \cup \{rel(l)\}$
11     if $e_1 \in E$ or $E$ is not feasible then
12         return $\bot$
13     return $E$
```

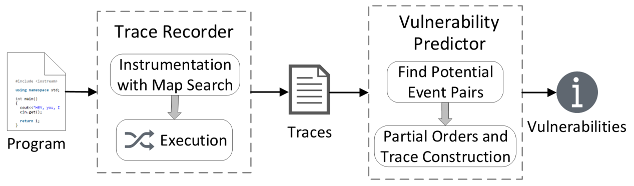

Program
Trace Recorder
Instrumentation with Map Search
Execution
Traces
Vulnerability Predictor
Find Potential Event Pairs
Partial Orders and Trace Construction
Vulnerabilities

Figure 7: An overview of ConVulPOE.

5.1 Instrumentation and Record

To record the execution trace, we need to instrument the original program. We employ Intel Pin [20], which instruments thread operations with POSIX threading primitives, to achieve this goal. We instrument the `malloc` and `free` functions (which is also used in `new` and `delete` operators) to record dynamic memory allocation. The limitation of Pin is that it cannot get a pointer-dereference directly. We adopt a heuristic approach which is used in prior work [4]. A pointer dereference usually consists of two consecutive instructions, including a `read` instruction and a memory access instruction. The read instruction's operand register is used as the base register of the memory access.

5.2 Vulnerability Prediction

The vulnerability predictor is independent of the instrumentation. It finds potential event pairs based on the traces generated by the trace recorder. We construct a partial order incrementally by inserting new events in an order as the same as they executed. In practice, we also consider other thread-related operations and add additional edges in the partial order graph. For example, we insert an edge from every thread-create event to the first event in the created thread. And for every thread-join event and the target thread it waits for, we insert an edge from the last event in the target thread to the thread-join event.

6 EXPERIMENTS

In this section, we conduct experiments to evaluate the performance of ConVulPOE and answer the flowing questions:

- **RQ1:** Can ConVulPOE detect the three kinds of concurrency memory vulnerabilities?
- **RQ2:** What is the capability of ConVulPOE in detecting vulnerabilities in real programs with fuzzing tools?
- **RQ3:** How does the number of threads influence ConVulPOE's detection capability?

6.1 Effectiveness Validation

We run ConVulPOE on a CVE benchmark [3] to verify its ability to detect vulnerabilities in multi-threaded programs. The dataset is used in ConVul [4] and contains ten programs which have various vulnerabilities. These programs reproduce real vulnerabilities in Linux and Firefox. We run each program ten times and present the experimental results in Table 1. The third column presents the times that ConVulPOE finds a vulnerability, and the fourth one shows the number of crashes in the ten runs. The fifth column shows the causes of vulnerabilities in the programs. It shows that some of these vulnerabilities are caused by race conditions, and some are due to the exchange of critical sections (CS). The results show that ConVulPOE finds vulnerabilities in eight benchmarks. ConVulPOE cannot always detect a vulnerability in every run of a program because it predicts vulnerabilities from a completed trace. If a trace has no potential pairs, ConVulPOE detects no vulnerabilities. We can also see that some programs crashed because of the occurrence of vulnerabilities during runtime. In such cases, the predictor component does not work since the trace is not complete. ConVulPOE fails to find vulnerabilities for two benchmarks, including CVE-2011-2183 and CVE-2016-1972. An detailed investigation reveals that CVE-2011-2183 contains an NPD vulnerability and CVE-2016-1972 contains a UAF vulnerability. The vulnerabilities in both benchmarks occur with totally new execution paths, and ConVulPOE fails to infer new feasible traces from an observed one. We present the analyses of the code for both benchmarks in supplemental material due to space limitations.

The results verify that our approach can detect all three kinds of concurrency memory vulnerabilities. According to the evaluation on ConVul [4], other tools such as UFO [13], and ThreadSanitizer [31] fails to detect most vulnerabilities in the benchmarks.

Table 1: Experimental results on CVE benchmarks

CVE ID	Cat.	# of suc.	# of cra.	Causes
CVE-2009-3547	NPD	10	0	exchange of CS
CVE-2011-2183	NPD	0	0	exchange of CS
CVE-2013-1792	NPD	4	0	race condition
CVE-2015-7550	NPD	6	4	exchange of CS
CVE-2016-1972	UAF	0	1	race condition
CVE-2016-1973	UAF	4	1	race condition
CVE-2016-7911	NPD	10	0	race condition
CVE-2016-9806	DF	10	0	race condition
CVE-2017-6346	UAF	9	0	race condition
CVE-2017-15265	UAF	10	0	race condition

6.2 Detection on Real Programs

In this experiment, ConVulPOE works as a bug oracle [21] from the perspective of fuzzing. We evaluate the performance of ConVulPOE with several real multi-threaded programs tested by fuzzing tools in prior works [5]. Table 2 shows a summary of the tested programs, including four parallel compression/decompression utilities (lbzip2, pbzip2, pigz, and xz), two libraries (libwebp and libvpx) used in popular browsers like Chrome and Firefox, and two video encoders (x264 and x265) for H.264/AVC and HEVC/H.265, respectively. All these programs use the native pthread library. The binary sizes of these programs range from 507K to 9.27M.

We adopt AFL to fuzz these programs for 24 hours with initial seeds provided by Muzz [5] to get the inputs for our evaluation. We do not use Muzz for fuzzing since the source code is unavailable. Before our evaluation, we use afl-cmin to minimize the input files generated by AFL. AFL finds no vulnerability in the 24 hours fuzzing. The fifth column of Table 2 shows the number of inputs for these programs in our evaluation. All inputs are generated by AFL except the inputs for x264 and x265. We use initial seeds as the two programs' inputs because there is only one file left with a broken format for each program after minimizing. The lengths of recorded traces range from hundreds to 100 millions.

We compare with two state-of-the-art detection tools, including ConVul [4] and UFO [13]. ConVul is implemented as a pin tool based on Intel Pin. Different from ConVulPOE, ConVul detects vulnerabilities during runtime. UFO works similarly to our tool. It instruments the program, records the trace, and employs a trace analyzer for vulnerability detection. UFO's instrumentation is based on the LLVM pass, which inserts additional code during the compile stage. Although LLVM-based instrumentation is faster than Pin, it is more complicated. We use Pin for instrumentation because it is suitable for black-box, binary-only instrumentation. Both ConVulPOE and ConVul can detect UAF, NPD, and DF while UFO can only detect UAF. All experiments are conducted on a server with an Intel Xeon E5-2650 v4 CPU and 128GB RAM. The operating system is Ubuntu 16.04, installed with GCC 9.3.0, AFL 2.57b.

We run these programs five times and report the average. When we run ConVul and UFO, they cannot complete all the tests for some programs. When ConVul analyzes pigz and cwebp, it seems to trigger a deadlock, and the program is stuck. UFO sometimes throws an unhandled exception, which makes UFO abort. After finding a UAF, UFO is expected to write the relevant information to a file. But in our evaluation, it did not work properly and threw an exception in some cases, leading to the stop of the program and the test. This is the major reason that the tests for lbzip2 and cwebp are not completed. Considering that UFO and ConVul did not complete some tests of inputs, and the numbers of inputs used for different programs are different, we use the metric vulnerabilities-per-input for fair comparisons. Table 3 presents the average number of vulnerabilities detected in a single execution of these programs. They all find the UAF vulnerability in lbzip2. Both ConVulPOE and ConVul find it in every execution. Currently, UFO can only find UAFs. It finds 0.96 UAF in one execution on average for lbzip2. ConVulPOE find more vulnerabilities on cwebp, x264 and x265 than ConVul and UFO, indicating that it can find vulnerabilities that cannot be found by other tools. However, ConVulPOE also misses some UAFs detected by UFO and some DFs detected by ConVul in pigz. Both ConVulPOE and UFO miss some UAFs detected by ConVul in vpxdec. These results indicate that ConVulPOE is an important complement to existing tools.

Table 4 presents the time used by the three detectors. For ConVulPOE and UFO, the time includes the execution time of an instrumented program and the detection time based on the trace. UFO consumes the least time on pbzip2-c, pbzip2-d, xz, and vpxdec because it finds no potential UAFs and needs no further computations. Although UFO finds more vulnerabilities than ConVulPOE on pigz, it costs more time (98544s vs. 287s). In general, ConVulPOE can find more vulnerabilities in less time. For programs such as lbzip2, cwebp, x264, and x265 with many potential UAFs, UFO needs to use Z3 for verification, which consumes much time to solve constraints. There usually are hundreds of potential UAFs in a single test, and the timeout for Z3 to solve each potential UAF's constraints is 2 minutes by default. In most cases, Z3 reaches the timeout, indicating that it usually takes more than one day to check all the potential UAFs. A less timeout will make UFO faster. However, it results in fewer vulnerabilities found by UFO. That is why UFO cannot complete the whole test in a week for these programs. Compared to ConVul, ConVulPOE consumes less time on pbzip2-c, pbzip2-d and vpxdec. For pigz and cwebp, ConVulPOE finds more vulnerabilities in less time. We also present the average time used by the predictor of ConVulPOE in parentheses. The results show that the predictor costs much less time than the recorder to record an execution trace. Detailed investigations reveal that the instrumentation of Intel Pin is time-consuming. For the programs that ConVulPOE finds vulnerabilities, the time cost by the predictor accounts from 20% to 63% of the total time.

Table 5 presents the maximum, minimum, and average distances of vulnerability-potential event pairs in each program's executions. The distance of an event pair is defined as the absolute difference between the indices of two potential events. The results show that most vulnerability-potential event pairs have very large distances in the traces. This finding indicates that even long-distant event pairs can cause a vulnerability, verifying the difficulty of detecting concurrency vulnerabilities. Compared with ConVul which limits the maximum sync-distance of potential event pairs, ConVulPOE detects vulnerabilities without limitations on potential event distances. It is also the reason why ConVulPOE finds more vulnerabilities than ConVul, as shown in Table 3.

In summary, ConVulPOE detects more vulnerabilities than other detectors with a competitive time. Although ConVul and UFO are faster than ConVulPOE in some cases, they are also extremely slow for some programs. However, ConVulPOE can finish the detection in no more than two days. Therefore, ConVulPOE is an important complement to existing detectors. The combination of AFL with concurrency vulnerabilities detectors also works well. With the help of these detectors, AFL can find more vulnerabilities.

6.3 Impacts of Number of Threads

To answer RQ3, we evaluate ConVulPOE on programs running with varying numbers of threads. In this experiment, we chose lbzip2, pigz, x264, and x265 for evaluation. The number of threads

Table 2: A Summary of tested Programs. The FILE in command will be replaced by actual input file.

ID	Program	Command	Binary size	# of inputs	Ave. length	Max. length
lbzip2	lbzip2 2.5	lbzip2 -k -t -9 -z -f -n4 FILE	507K	841	1,208,862	3,351,406
pbzip2-c	pbzip2 1.1.13	pbzip2 -f -k -p4 -S16 -z FILE	509K	27	282	282
pbzip2-d	pbzip2 1.1.13	pbzip2 -f -k -p4 -S16 -d FILE	509K	121	357	713
pigz	pigz 2.5	pigz -p 4 -c -b 32 FILE	592K	106	3106	6819
xz	xz 5.2.5	xz -9 -k -T 4 -f FILE	1.1M	39	2,251,121	23,282,362
cwebp	libwebp 1.2.0	cwebp -mt FILE -o out.webp	4.5M	1182	3,617,616	115,915,206
vpxdec	libvpx 1.9.0	vpxdec -t 4 -o out.y4m FILE	5.49M	1609	36,490	266,384
x264	x264 HEAD: 59c0609	x264 -threads=4 -o out.264 FILE	9.27M	17	495,141	845,987
x265	x265 3.4	x265 -input FILE -pools 4 -F 2 -o out.265	688K	10	11,116,821	11,116,880

Table 3: Average number of vulnerabilities detected per input. The symbol (-) means that the tests are not completed.

ID	ConVulPOE	ConVul	UFO
lbzip2	1 UAF	1 UAF	0.96 UAF (-)
pbzip2-c	0	0	0
pbzip2-d	0	0	0
pigz	0.69 UAF, 10.42 NPDs	0.29 DF (-)	15.29 UAFs
xz	0	0	0
cwebp	1.35 UAFs	0 (-)	0.58 UAF (-)
vpxdec	0	0.01 UAF	0 (-)
x264	5.35 NPDs	0.18 NPD	0 (-)
x265	1.5 NPDs	0.1 UAF	0 (-)

Table 4: Time consumed by different detectors. The symbol - means that the tests either are not completed due to fatal bugs or cannot complete in a week.

ID	ConVulPOE (s)		ConVul (s)	UFO (s)
lbzip2	18,753	(3766.4)	8169	-
pbzip2-c	238	(< 1)	572	214
pbzip2-d	646	(< 1)	3367	496
pigz	287	(101.4)	-	98,544
xz	567	(42.8)	211	143
cwebp	108,593	(25,266.4)	-	-
vpxdec	24,558	(17)	126,169	1069
x264	534	(239.6)	146	-
x265	994	(624.8)	140	-

Table 5: Distances of vulnerability-potential event pairs.

ID	Max	Min	Average
lbzip2	3,351,132	2880	1,208,588.64
pigz	821	6	207.03
cwebp	34,106,101	1	1,027,149.48
x264	15,679	5	778.90
x265	4,191,081	682,628	1,924,895.62

is set to be 2, 4, 8, and 16. Other settings are the same as previous. We run each program five times and report the average results.

Figure 8 shows the average number of vulnerabilities that ConVulPOE finds in a program's single run versus the number of threads. The average number of UAFs found in pigz per run increases from 0.38 to 0.70, 0.68, and 0.69, respectively. However, the average number of NPDs found in x264 and pigz per run decreases as the number of threads increases. The number of threads has no impact on the number of vulnerabilities found in lbzip2 and x265.

The results demonstrate that the number of threads has different influences on ConVulPOE's capability. On the one hand, the probability of finding more potential pairs increases as the number of threads increases. On the other hand, since Algorithm 1 always makes unordered conflicting event pairs keep their original execution order, it increases the probability of forming a cycle. If a cycle is formed due to wrong orders, we will miss a vulnerability. Thus, ConVulPOE is more likely to miss some vulnerabilities as the number of threads increases. In summary, the number of threads is a double-edged sword for ConVulPOE.

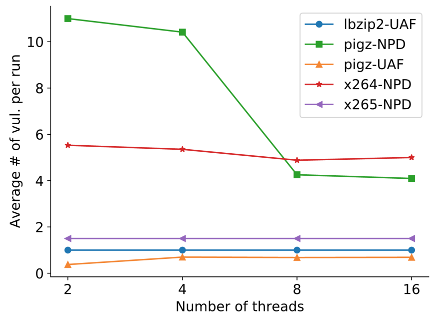


Figure 8: Average number of vulnerabilities found by ConVulPOE per input versus the number of threads.

7 DISCUSSION

ConVulPOE works on the traces generated in the executions of a program. To detect a vulnerability from a trace, we must generate a new trace exposing the vulnerability. We use partial orders to represent the necessary execution orders of events to reduce the search space. The trace may be very large in some large programs, consuming a long time to generate a partial order. In such a case, we can divide the trace into multiple parts and detect each part separately. Fuzzing tools like AFL also suggest using small input files and simple target binary files [39]. It suggests detecting on modules of a large program rather than the whole program.

There is also room for future research. The first is trace reducing. Loop structures are widely used in programs, generating many events if they contain some operations in which we are interested. It is not extendable to record all the events. The second one is to improve the completeness. Algorithm 1 needs to order the unordered conflicting pairs and avoids forming a cycle. Every ordering inserts a new edge to the partial order graph. Thus, a cycle may be formed by different orderings. For example, as described in Section 3.3, there may be unordered conflicting event pairs (see line 5 of Algorithm 1). If these conflicts are not resolved properly, there may exist edges such as $e \rightarrow e'$, $e' \rightarrow e''$, and $e'' \rightarrow e$. Our algorithm cannot construct a feasible trace from the partial order graph in these cases and thus misses some vulnerabilities. That is, our algorithm is still incomplete. It is worth noting that there are no guaranteed methods (including ours) to find a vulnerability in polynomial time if there is one. How to improve the completeness of the algorithm is worth further studying.

8 RELATED WORK

Partial-order-based predictive analysis. In previous works, partial-order-based prediction is used to detect data races in multi-threaded programs. A data race occurs when two threads access the same data simultaneously, and at least one modifies it. Most race detectors [1, 6, 12, 24, 27, 38] employ Lamport's happen-before (HB) partial order [18]. It ensures that the partial order can always generate a feasible trace. However, it is highly incomplete. Causally-precedes (CP) partial order [32] and weakly-causally-precedes (WCP) partial order [16] are two weaker partial orders based on HB. They can generate more different traces than HB. However, they are still highly incomplete because they are closed under the composition with HB. Doesn't-Commute (DC) [26] is weaker than WCP and can explore more possible execution trace and find more races. However, the constraints of DC are not sufficient to generate a feasible trace. M2 [23] proposes the trace-closed partial order, which is weaker and finds more races than DC.

Data race detection uses the partial order to determine the concurrency of two events and get a valid trace exposing the data race. Our algorithm is different from existing approaches designed for data race detection [16, 23, 26]. The key to predicting a vulnerability or a data race from an observed trace is to rearrange the trace to get a new one exposing it. The new trace must meet several constraints to be feasible. Such an approach is also widely used in prior data race detection work. Our algorithm follows the same procedure but expands it. First, we consider two additional event types, including `free` and `dereference`, to detect memory vulnerabilities. Since both events are `read` events, the new setting is still compatible with the old ones. Second, memory vulnerabilities are more complicated than data races, and their witness traces are quite different. A trace exposing a data race only requires that the two events causing a data race are rearranged consecutively (e.g., usually at the end of the trace). However, exposing a memory vulnerability is more complicated because not all data races can cause memory vulnerabilities. For example, if two events in different threads are protected by the same lock, they cannot cause a data race. However, the execution orders of the critical sections containing the two events may change in some situations and thus cause a memory vulnerability.

Concurrency memory vulnerability detection. Programs written in unsafe languages such as C/C++ are prone to memory errors. UAF, NPD, and DF are serious threats to software security. However, it is challenging to find them in multi-threaded programs. UFO [13] is a concurrent UAF detector based on MaxModel [14], which is used to capture a maximal set of feasible traces. To make the trace feasible, it employs a constraint solver to solve the constraints. ConVul [4] detects vulnerabilities by judging whether two events are exchangeable based on sync-edge and sync-distance, which is defined by HB. ExceptioNULL [11] detects NPD in concurrent Java programs, which is also based on constraint solving. ExceptioNULL adopts a scheme that focuses on a small trace segment to make the constraints simpler. However, it reduces the number of feasible traces and may miss some vulnerabilities.

Different from these methods, our approach constructs new traces exposing vulnerabilities via partial orders. It benefits the detection from the following two aspects. First, some events must execute before others to meet these constraints. It is straightforward to use partial orders to represent such relations between two events. Second, we can add new partial order relations via the rules described in Section 3.2. Although a partial order graph meets all the constraints, we may fail to get a feasible trace from it directly because there may exist unordered conflicting events. To solve this difficulty, we use a heuristic algorithm that inserts additional edges to the partial order graph to get a feasible trace.

9 CONCLUSION

This paper proposes a new approach for concurrency vulnerability detection based on partial orders. We propose algorithms to extract vulnerability-potential event pairs from an execution trace of a multi-threaded program. A novel algorithm is developed to generate an event pair's feasible set, which considers the cases that two events are protected by the same lock. Then, we construct partial orders of feasible event sets and generate a feasible trace exposing vulnerabilities. Exploiting partial orders to generate traces reduces the search space of possible executions and improves computational efficiency. Compared to other detectors, our method is more theoretically sound and efficient. We implement a prototype of our method and conduct experiments to evaluate its performance. Experimental results show that ConVulPOE shows superiority over state-of-the-art tools in both effectiveness and efficiency.

ACKNOWLEDGMENTS

The research presented in this paper is supported in part by the National Natural Science Foundation of China (No. 61602370, 62032010, 61772411, U1736205), the Natural Science Foundation of Shaanxi Province (No. 2021JM-018), the Key Research Program of Frontier Sciences, CAS (No. ZDBS-LY-7006), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (No. 2017151), and the HK RGC Project (No. 152239/18E).

REFERENCES

[1] 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* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[2] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: Early Detection of Dangling Pointers in Use-after-Free and Double-Free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 133–143. https://doi.org/10.1145/2338965.2336769

[3] Yan Cai. 2019. CVE Benchmark. Retrieved February 20, 2021 from https://github.com/mryancai/ConVul

[4] 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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 706–717. https://doi.org/10.1145/3338906.3338927

[5] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[6] Mark Christiaens and Koenraad De Bosschere. 2001. TRaDe: Data Race Detection for Java. In *Proceedings of the International Conference on Computational Science-Part II (ICCS '01)*. Springer-Verlag, Berlin, Heidelberg, 761–770. https://doi.org/10.1007/3-540-45718-6_81

[7] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[8] Common Weakness Enumeration. 2006. CWE-415: Double Free. Retrieved February 20, 2021 from https://cwe.mitre.org/data/definitions/415.html

[9] Common Weakness Enumeration. 2006. CWE-416: Use After Free. Retrieved February 20, 2021 from https://cwe.mitre.org/data/definitions/416.html

[10] Common Weakness Enumeration. 2010. CWE-476: NULL Pointer Dereference. Retrieved February 20, 2021 from https://cwe.mitre.org/data/definitions/476.html

[11] 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* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 47, 11 pages. https://doi.org/10.1145/2393596.2393651

[12] 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* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[13] J. Huang. 2018. UFO: Predictive Concurrency Use-After-Free Detection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 609–619. https://doi.org/10.1145/3180155.3180225

[14] 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

[15] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. 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

[16] 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* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

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

[18] 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

[19] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 529–541. https://doi.org/10.1145/3274694.3274718

[20] 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* (Chicago, IL, USA) *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[21] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2946563

[22] 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* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 446–455. https://doi.org/10.1145/1250734.1250785

[23] Andreas Pavlogiannis. 2019. Fast, Sound, and Effectively Complete Dynamic Race Prediction. *Proc. ACM Program. Lang.* 4, POPL, Article 17 (Dec. 2019), 29 pages. https://doi.org/10.1145/3371085

[24] 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* (San Diego, California, USA) *(PPoPP '03)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/781498.781529

[25] John Regehr. 2010. A Guide to Undefined Behavior in C and C++. Retrieved February 20, 2021 from https://blog.regehr.org/archives/213

[26] Jake Roemer, Kaan Genç, and Michael D. Bond. 2018. High-Coverage, Unbounded Sound Predictive Race Detection. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 374–389. https://doi.org/10.1145/3192366.3192385

[27] D. Schonberg. 1989. On-the-Fly Detection of Access Anomalies. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI '89)*. Association for Computing Machinery, New York, NY, USA, 285–297. https://doi.org/10.1145/73141.74844

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

[29] SecuriTeam. 2016. OpenSSL NULL Pointer Dereference Vulnerabilities. Retrieved February 20, 2021 from https://securiteam.com/securitynews/5FP3B00HQE/

[30] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC 2012*. https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

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

[32] 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* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[33] E. Stepanov and K. Serebryany. 2015. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 46–55. https://doi.org/10.1109/CGO.2015.7054186

[34] L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal War in Memory. (2013), 48–62. https://doi.org/10.1109/SP.2013.13

[35] The Clang Team. 2007. UndefinedBehaviorSanitizer. Retrieved February 20, 2021 from https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[36] Nischai Vinesh and M. Sethumadhavan. 2020. ConFuzz—A Concurrency Fuzzer. In *First International Conference on Sustainable Technologies for Computational Intelligence*, Ashish Kumar Luhach, Janos Arpad Kosa, Ramesh Chandra Poonia, Xiao-Zhi Gao, and Dharm Singh (Eds.). Springer Singapore, Singapore, 667–691. https://doi.org/10.1007/978-981-15-0029-9_53

[37] M. Xu, S. Kashyap, H. Zhao, and T. Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[38] 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

[39] Michał Zalewski. 2014. Technical "whitepaper" for afl-fuzz. https://lcamtuf.coredump.cx/afl/technical_details.txt