Reorder Pointer Flow in Sound Concurrency Bug Prediction

Yuqi Guo*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
guoyq@ios.ac.cn

Shihao Zhu*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
zhush@ios.ac.cn

Yan Cai†
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
yancai@ios.ac.cn

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

Jian Zhang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
zj@ios.ac.cn

ABSTRACT

Due to the non-determinism of thread interleaving, predicting concurrency bugs has long been an extremely difficult task. Recently, several sound bug-detecting approaches were proposed. These approaches are based on local search, i.e., mutating the sequential order of the observed trace and predicting whether the mutated sequential order can trigger a bug. Surprisingly, during this process, they never consider reordering the data flow of the pointers, which can be the key point to detecting many complex bugs. To alleviate this weakness, we propose a new flow-sensitive point-to analysis technique ConPTA to help actively reorder the pointer flow during the sequential order mutation process. Based on ConPTA, we further propose a new sound predictive bug-detecting approach Eagle to predict four types of concurrency bugs. They are null pointer dereference (NPD), uninitialized pointer use (UPU), use after free (UAF), and double free (DF). By actively reordering the pointer flow, Eagle can explore a larger search space of the thread interleaving during the mutation and thus detect more concurrency bugs. Our evaluation of Eagle on 10 real-world multi-threaded programs shows that Eagle significantly outperforms four state-of-the-art bug-detecting approaches UFO, ConVul, ConVulPOE and Period in both effectiveness and efficiency.

CCS CONCEPTS

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

KEYWORDS

Concurrency bug prediction, point-to analysis

ACM Reference Format:
Yuqi Guo, Shihao Zhu, Yan Cai, Liang He, and Jian Zhang. 2024. Reorder Pointer Flow in Sound Concurrency Bug Prediction. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623300

*Co-first authors
†Corresponding author

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623300

1 INTRODUCTION

Writing a bug-free multi-threaded program is highly difficult due to the non-determinism of thread interleaving [45]. We denote these concurrency-related bugs as **concurrency bugs**. Typically, the concurrency bugs include (1) null pointer dereference (NPD), a program tries to access a memory object via a NULL pointer; (2) uninitialized pointer use (UPU), a program tries to use a pointer before it is initialized; (3) use after free (UAF), a program tries to access a memory object after it has been freed; (4) double free (DF), a program tries to free the same memory object twice; and (5) other types of bugs, e.g., data races. By manipulating the thread interleaving, hackers can exploit concurrency bugs to launch serious system attacks [41, 74]. For example, the notorious DirtyCow bug [43] (caused by a data race) in the Linux kernel can be exploited by a malicious attacker to gain the root privilege. According to the NVD database [12], concurrency bugs have been occurring more and more frequently in recent years. Fortunately, the community has been aware of this and has proposed many bug-detecting approaches to handle this threat. However, due to the challenge of exploring all thread interleaving (NP-complete [28, 47]), existing approaches may suffer from low efficiency and/or effectiveness.

Model-checking approaches detect concurrency bugs by exploring all possible program states. When applied to large-scale programs, they may suffer from the *state space explosion* problem, thus being inefficient. Race-based bug detectors [78] infer concurrency bugs based on the race-detecting results. They can be efficient, but usually ineffective, as concurrency bugs are not strongly correlated

with data races: about 90% of reported data races are benign [21, 50] and concurrency bugs can exist without causing data races [11].

Recently, several **witness-based** and **predictive** sound bug-detecting approaches were proposed. The former ones [7, 14, 18, 24] monitor the program execution and report a bug when witnessing a bug pattern. Although these approaches can be helpful in bug fixing, they are unproductive in bug hunting because they cannot *predict* bugs, i.e., finding bugs before they really occur. Cooperating with *fuzzing* [77] and/or the *controlled scheduling* [1, 8, 26, 66, 70] techniques may alleviate this weakness. But these techniques may cause performance issues [13, 71]. Minion [59] combines static analysis and symbolic execution to guide the fuzzing instead of blindly fuzzing. Thus, it achieves high efficiency and effectiveness. However, static analysis and symbolic execution may suffer from the problem of *state space explosion* and Minion might degrade to blindly fuzzing when applied to large-scale programs.

Predictive approaches [11, 23, 33, 75], however, are designed to *predict* concurrency bugs before they really occur. They usually take one observed trace as the input and rely on different models or algorithms to infer not-yet-occurred thread interleaving and then detect concurrency bugs. They extract partial orders from the observed trace and then rely on SMT/SAT solvers or partial order graphs to detect bugs. We denote the former ones [23, 33] the **SMT-based** approaches and the latter ones [11, 75] the **graph-based** approaches. For example, UFO [33] is the state-of-the-art SMT-based UAF bug-detecting approach, and ConVul [11] (and its improved offline version ConVulPOE [75]) is a recently proposed graph-based NPD/UAF/DF bug-detecting approach. Both UFO and ConVul can achieve soundness, i.e., no false positive reports.

In predictive approaches, the key method that supports them to predict the not-yet-occurred thread interleaving is the *sequential order mutation* method. The *sequential order mutation* method is actually a local search algorithm that makes minor mutations to the sequential order of observed traces, i.e., switching the occurrence order of some events, to **predict** new possible thread interleaving. It can help the predictive approaches escape from inefficient fuzzing and/or controlled scheduling processes and thus endow them with an advantage over the witness-based approaches in efficiency. However, due to the reason that such a method only mutates in a small local scope near the sequential order of the observed trace, the thread interleaving coverage of the predictive approaches might be smaller than that of the witness-based approaches.

To alleviate this weakness, existing predictive approaches [10, 11, 33, 48, 54] have proposed several new mutation rules to expand the scope of this local search. Surprisingly, to the best of our knowledge, none of them have ever considered reordering the **data flow of the pointers** (**pointer flow** for short). This can cause them to miss complex bugs. We discuss an example of this in Section 3. To fulfill this research gap, we design a new flow-sensitive point-to analysis technique ConPTA and based on it propose a new sound bug-detecting approach Eagle. Different from many point-to analysis approaches based on pointers and point-to sets, ConPTA analyzes events and LLVM IR and then models the real pointer flow as *pointer flow sequences*. It identifies *potentially conflicting* pairs to help Eagle **actively reorder the pointer flow**. Thus, Eagle achieves a larger thread interleaving coverage and detects more bugs.

We implement Eagle (with ConPTA) to predict four types of concurrency bugs, i.e., NPD, UPU, UAF, and DF. We evaluate it on 10 real-world programs with four recent bug-detecting approaches. We also evaluate it with Eagle$_n$ (Eagle without ConPTA) to evaluate the actual effect of *pointer flow reordering*. The experimental results show Eagle outperforms three *predictive* approaches UFO,ConVul, and ConVulPOE and one *witness-based* approach Period in both effectiveness and efficiency. Under the same time budget, Eagle finds 12 UAF bugs (3 times more than UFO), 107 NPD/UAF/DF bugs (4.28 times more than ConVul, 10.89 times more than ConVulPOE, and many more bugs than Period because Period cannot find any bug), and 202 UPU bugs in addition. Besides, ConPTA (the *pointer flow reordering* technique) helps Eagle detect 45.1% more bugs with only increasing by 0.46% of time consumption.

In summary, we make the following contributions:

- We are the first to provide an analysis of the limitation of the *sequential order mutation* method of the existing predictive bug-detecting approaches in terms of thread interleaving coverage. To alleviate this, we propose to *actively reordering the pointer flow* to help achieve a larger thread interleaving coverage.
- We propose ConPTA, a linear-time flow-sensitive point-to analysis technique, and Eagle, a polynomial-time new sound predictive bug-detecting approach, to effectively and efficiently predict concurrency bugs.
- We implement Eagle (with ConPTA) to predict four types of concurrency bugs. The evaluation of Eagle on 10 real-world programs shows that Eagle outperforms four recent bug-detecting approaches in both effectiveness and efficiency.

The rest of the paper is organized as follows. Section 2 lists the notations used in this paper. Section 3 motivates the work. Section 4 presents the detailed design. Section 5 summarizes the implementation and shows the experimental results. Section 6 introduces the related work. Section 7 concludes the paper.

2 PRELIMINARIES

Following the prior work [11, 33, 35, 37, 54], we assume that multi-threaded programs follow the *sequential consistency* memory model [39], which requires that the result of any execution is the same as if the operations are executed in sequential order. We use an **execution trace** (**trace** for short) to present such sequential order.

Execution trace. An execution trace σ is a full sequence of events, in which each event represents one program operation. We say an event e_1 **occurs before** another event e_2 in the trace σ if e_1 shows up before e_2 in σ. To detect the four concurrency bugs listed in Section 1, we focus on the following five types of events:

(1) **Write/read**, denoted by $wr(t, x, v)/rd(t, x, v)$, indicating thread t writes/reads a value v to/from an object at memory address x.
(2) **Acquire/release**, denoted by $acq(t, x)/rel(t, x)$, indicating thread t acquires/releases a mutex at memory address x.
(3) **Allocate/free**, denoted by $alloc(t, x)/free(t, x)$, indicating thread t allocates/frees an object at memory address x.
(4) **Enter/exit**, denoted by $enter(t, f)/exit(t, f)$, indicating thread t enters/exits a function f.

	Thread A	*Thread B*		*Thread A*	*Thread B*
I_1	p=h;		e_1	rd(0x100,0x1000)	
			e_2	wr(0x200,0x1000)	
I_2	h=h->next;		e_3	rd(0x1100,0x2000)	
			e_4	wr(0x100,0x2000)	
I_3		q=h;	e_5		rd(0x100,0x2000)
			e_6		wr(0x300,0x2000)
I_4		h=h->next;	e_7		rd(0x2100,0x3000)
			e_8		wr(0x100,0x3000)
I_5		lock(&l)	e_9		acq(0x500)
I_6		q->x++	e_{10}		rd(0x2008, 7)
			e_{11}		wr(0x2008, 8)
I_7		unlock(&l)	e_{12}		rel(0x500)
I_8		free(q)	e_{13}		free(0x2000)
I_9	lock(&l)		e_{14}	acq(0x500)	
I_{10}	p->x++		e_{15}	rd(0x1008, 5)	
			e_{16}	wr(0x1008, 6)	
I_{11}	unlock(&l)		e_{17}	rel(0x500)	
I_{13}	free(p);		e_{18}	free(0x1000)	

Figure 1: The conflicting relation between e_{13} and e_{15} can be missed without actively reordering the pointer flow.

(5) **Branch**, denoted by $br(t)$, indicating thread t jumps *conditionally*. It includes both explicit conditional jumps, e.g., if-else statements, and implicit conditional jumps, e.g., member method calls in object-oriented programming languages [10, 37].

The first three types of events are used to track the memory-accessing behavior of the programs. The fourth type is used to analyze the pointer-passing behavior across function call/return interfaces. The fifth type is required by our sequential order verification algorithm EagleSeqC (discussed in Section 4.4) to achieve soundness. We use a symbol e to denote an event, $e.addr$ to refer to its **address** x (if it exists) and $e.val$ to refer to its **value** v (only defined for write/read events). Please see the above definition of events for the meanings of symbols x and v. We omit the thread ID if there is no ambiguity. An example trace is shown in Figure 1.

Read-from relation. Following prior work [10, 11, 33, 54], we define the **read-from** relation to model the read-after-write dependency, which is the most common type of data dependency in multi-threaded programs. It can be analyzed from the trace. Given a trace σ, we say a read event e_r **reads from** a write event e_w if e_r observes the value written by e_w in σ. We denote this as $RF_\sigma(e_r) = e_w$. If e_r does not read from any write event in σ, we mark it as $RF_\sigma(e_r) = \perp$. In EagleSeqC (discussed in Section 4.4), we maintain all the *read-from* relations before any *branch* event in function *ObsClosure()* of Algorithm 3 to achieve soundness.

3 MOTIVATION

We first discuss the limitation of the *sequential order mutation* method of existing predictive works in terms of thread interleaving coverage. Then, we propose our new mutation rule, i.e., actively reordering the pointer flow.

3.1 Challenge: the Conflicting Relation

Concurrency bugs are usually caused by two events that can access the same object without proper synchronization. These two events are denoted as a **conflicting** pair [10, 11, 33, 48, 54]. Predictive approaches usually begin with identifying the *conflicting* relation from the input observed trace. Based on it, they can adopt the *sequential order mutation* method, i.e., switching the occurrence order of several *conflicting* events, to predict *bug-triggering* sequential order under which the program can trigger concurrency bugs.

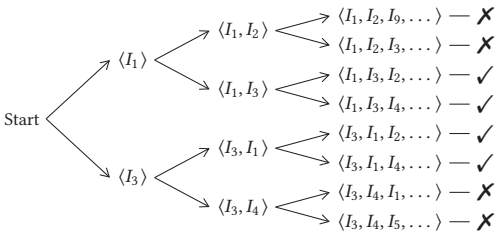


Figure 2: All possible interleavings of the program shown in Figure 1. The check and cross marks indicate whether e_{13} and e_{15} are trace-conflicting under each interleaving.

As the foundation of bug prediction, conflicting relation identification can significantly influence the effectiveness and efficiency of a bug-detecting approach. To be effective and efficient, existing predictive approaches [10, 11, 33] design and propose a simple but effective conflicting relation identification algorithm. We denote their identification result as the **trace-conflicting** relation. Two events e_a and e_b are deemed as **trace-conflicting** with each other if they access the same object **in the observed trace**, i.e., $e_a.addr$ and $e_b.addr$ are inside the memory region of the same object. By focusing on such a relation, existing approaches achieve high effectiveness and efficiency. Note, many existing runtime pointer disambiguation approaches [2, 16] check whether two pointers access the same memory block/range at runtime to decide whether their memory accesses are overlapped. Such approaches can also be considered as belonging to the *trace-conflicting* category.

However, the *trace-conflicting* relation can only capture a subset of the real *conflicting* relation because *conflicting* events do not necessarily access the same object **in every possible thread interleaving**. We take Figure 1 as an example to illustrate this. Threads A and B are two worker threads. Each of them fetches a workload from the global working list h and then works on it. In the normal case shown in Figure 1, they obtain different workloads, i.e., p = 0x1000 and q = 0x2000 as shown in e_1 and e_5. In that case, the trace contains no *trace-conflicting* relation because all the events of the two threads access different workloads separately. As a result, existing predictive bug-detecting approaches will report no bug given this trace. However, e_{13} and e_{15} can actually be *conflicting* with each other and they can form a real UAF bug. For example, under the sequential order $\langle I_1, I_3, I_4, I_2 \rangle$, pointers p and q can be assigned with the same value and thus point to the same workload (Figure 3(b)). In that case, the two events e_{13} and e_{15} can be *conflicting*. If e_{13} further occurs before e_{15}, a UAF bug will occur.

The insight of this false negative is that the *trace-conflicting* relation is a local search result of the *conflicting* relation. Its actual effect is highly dependent on the observed trace. In Figure 2, we list all possible thread interleavings of the program shown in Figure 1 and use check and cross marks to denote whether e_{13} and e_{15} can form a *trace-conflicting* pair under each interleaving. Given the assumption that each interleaving has an equal chance to appear, the *trace-conflicting* relation has a chance up to 50% of missing the real *conflicting* relation and causing the false negative.

To alleviate this weakness, one possible solution seems to provide a predictive approach with multiple observed traces, e.g., by applying fuzzing and/or controlled scheduling techniques. If events e_{13}

and e_{15} can be *trace-conflicting* with each other in some observed traces, the predictive approach will be able to identify this conflicting relation. However, existing fuzzing techniques are mostly oriented by maximizing the control flow coverage [13, 44, 69], which cannot help to explore different thread interleavings. Thread-aware fuzzing techniques explore different thread interleavings by scheduling the threads [13] or changing the global priority of a thread [69] **randomly**. Since the scale of possible thread interleaving is NP-complete [28, 47], these random search techniques can be inefficient. As for controlled scheduling techniques, they have long been known ineffective in exploring the space of possible thread interleaving [71]. Our evaluation also shows that Period, the state-of-the-art witness-based bug-detecting approach which adopts the controlled scheduling technique for thread interleaving exploring, is inefficient in handling large-scale programs as it exceeds the time limit of 300 hours on all programs without producing any report.

Challenge: how to efficiently explore the space of possible thread interleaving to identify the conflicting relation?

3.2 Actively Reorder Pointer Flow

To achieve high efficiency, we still rely on the *trace-conflicting* relation, i.e., comparing the values of addresses for different events to identify *conflicting* relation. However, we intend to not only compare these values on the observed trace but also **predictively** compare them on inferred pointer-flow-reordered traces.

We notice two facts. First, the result of *trace-conflicting* relation identification is fully dependent on the values of $e.addr$. To efficiently **predict** different *conflicting* relations, we focus on predicting traces that have different such values. Second, thread interleaving can influence the pointer flow related *read-from* relations, which can further influence the values of $e.addr$. This is because the values of $e.addr$ are usually pointers and many such values are loaded from the memory before being used. For example, in Section 3.1, we provide an example of different thread interleaving causing the *read-from* relations of e_1 and e_5 (and thus the pointer flow of pointers p and q) to change, which then causes $e_{13}.addr$ and $e_{15}.addr$ to be different. To **predict** traces with different values for $e.addr$, we focus on manipulating thread interleaving to reorder the *read-from* relations, especially the ones related to pointer flow.

Based on the above two facts and inspired by existing predictive bug-detecting approaches [10, 11, 33, 48, 54] which adopt the *sequential order mutation* method to efficiently predict new sequential order, we propose a new mutation rule, namely **actively reordering the pointer flow**, to mutate the sequential order of the observed trace to **predictively** explore traces with different pointer flow related *read-from* relations and thus different values for $e.addr$. Then, we can **predictively** identify more *conflicting* relations.

We first extract the pointer flow information and then try to infer traces with different pointer flow related *read-from* relations to **predictively** identify *conflicting* pairs. In general, the **point-to analysis** technique can be a popular choice for finishing this job. However, existing point-to analysis is limited in terms of accuracy and performance [36]. Inspired by Razzer [36], a recently proposed fuzzing-based data race detector that utilized an improved lightweight *point-to analysis* technique to first overestimate the *race candidates* and then filter out the false positive ones by dynamic fuzzing, we first overestimate the *conflicting* pairs as **potentially conflicting** pairs and then filter out false positive ones by EagleSeqC (Section 4.4). For two events e_a and e_b, we say that they are **potentially conflicting** with each other if either of the two conditions is satisfied:

(1) there exists an event e^* (by the same thread t_a of e_a) that reads a value from the same address written by or to be written by e_b (from a thread $t_b \neq t_a$); and t_a uses the value as $e_a.addr$;
(2) there exists two events e_a^* (by thread t_a of e_a) and e_b^* (by thread $t_b \neq t_a$ of e_b); they read values from the same address and both t_a and t_b use the value read by e_a^* and e_b^* as $e_a.addr$ and $e_b.addr$, respectively.

In the first case, we denote the write event e_b/e_a as a pointer assignment event and assume it may influence the value of $e_a.addr/e_b.addr$ and thus may control which object e_a/e_b will access. If the thread uses the value written by it as $e_a.addr/e_b.addr$, e_a and e_b will access the same object. Note, here we deem that the pointer assignment event accesses the object stored at address $e_b.val$ instead of the one stored at address $e_b.addr$ to keep in consistency with the definition of *conflicting*. In the second case, we assume that the thread may read the values for $e_a.addr$ and $e_b.addr$ from the same address. If that could happen, the two events access the same object.

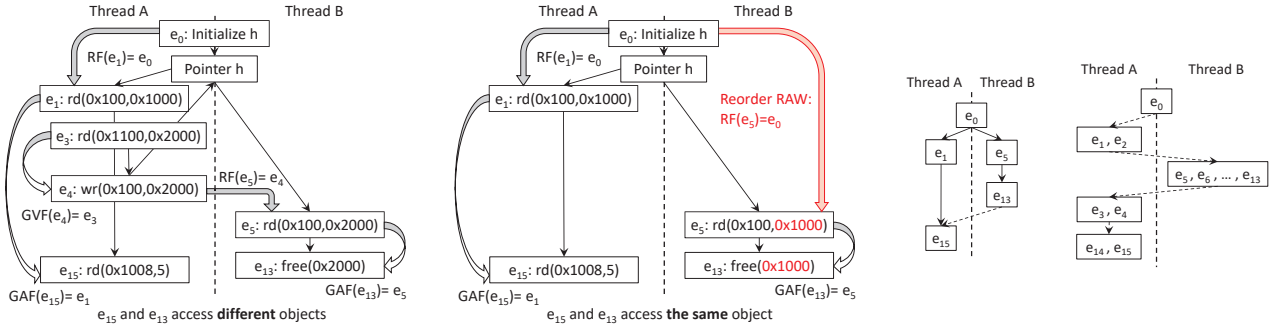
Potentially conflicting relation can provide a better solution to the challenge in Section 3.1 and help detect more bugs. For example, given the trace in Figure 1, the program reads both $e_{15}.addr$ and $e_{13}.addr$, which are values of pointers p and q, from the same address 0x100 (the address of pointer h). Thus, (e_{15}, e_{13}) forms a *potentially conflicting* pair according to the second condition. Based on this, a bug-detecting approach may find the UAF bug.

The general purpose of *potentially conflicting* is similar to that of *may-alias* as both target on identifying *conflicting* relations; but their definitions are significantly different. *May-alias* relation is usually defined on pointers and is identified by calculating point-to sets, whereas *potentially conflicting* relation is defined on events (without an explicit concept of pointers) and we propose *pointer flow sequences*, a model of real pointer flow (Section 4.1), to identify them. We propose a new lightweight flow-sensitive point-to analysis technique ConPTA to identify *potentially conflicting* pairs. It analyzes the context of trace and LLVM IR to identify the *get-from* relations (Section 4.1). Then, it models real pointer flow by inferring *pointer flow sequences* utilizing the *get-from* and *read-from* relations. Finally, it predicts *potentially conflicting* pairs. After that, we propose EagleSeqC to verify these pairs and detect real bugs.

We combine EagleSeqC with ConPTA to propose our new sound predictive bug-detecting approach Eagle to *soundly* predict NPD, UPU, UAF, and DF bugs, i.e., each reported bug can be reproduced by real program execution. By identifying the *potentially conflicting* relation with the help of ConPTA and fully utilizing the *actively reordering the pointer flow* mutation rule, Eagle can achieve a larger thread interleaving coverage and detect more real bugs.

4 OUR APPROACH

Paired relation. Figure 4 provides an overview of Eagle. It first instruments and runs the program to collect an execution trace and an LLVM IR dynamic slice. For each event recorded in the trace,



(a) Original pointer flow (b) Reordered pointer flow (c) Sequential order (d) Witness trace

Figure 3: Sketch of the sequential order mutation process for the UAF bug shown in Figure 1. Thin lines indicate the real pointer flow. Hollow lines indicate how ConPTA utilizes get-from and read-from relations to infer the real pointer flow. Dashed lines indicate the occur-before relations. Given a potentially conflicting pair (e_{15}, e_{13}), red parts indicate how Eagle reorders the pointer flow of the original trace to trigger the bug (Figure 3(b)). To do this, Eagle first infers the bug-triggering sequential order (Figure 3(c), Section 4.3) and then tries to verify it by generating a witness trace (Figure 3(d), Section 4.4).

it records the LLVM IR instruction that triggers this event in the dynamic slice. Besides, it also records all the type-casting and offset-addressing instructions whose results are used by the recorded instructions. There exists a one-to-one **paired** relation between the non-type-casting and non-offset-addressing instructions and the events, e.g., in Figure 5, each I_i is paired with e_i. Besides, LLVM IR instructions have similar operands with *paired* events. A non-type-cast and non-offset-addressing instruction has an address operand and its *paired* event also has an address, and a read/write instruction has a value operand and its *paired* read/write event also has a value. Thus, we further define a one-to-one **paired** relation between LLVM IR variables used as the address/value operands in instructions and the address/value of the *paired* events. In Figure 5, we pair %1 of foo() with $e_2.addr$, and %2 of foo() with $e_2.val$.

After instrumentation and execution, Eagle analyzes both the trace and dynamic LLVM IR slice to predict bugs. In the first two steps, it utilizes ConPTA to analyze the pointer flow of the observed trace (Section 4.1) and identifies *potentially conflicting* pairs (Section 4). Next, it constructs a *bug-triggering* sequential order for each *potentially conflicting* pair and each bug type (Section 4.3). Finally, it utilizes EagleSeqC to verify the *bug-triggering* sequential order and gives sound reports (Section 4.4).

4.1 Pointer Flow Analysis

Get-from Relation. We define two **get-from** relations on a trace σ, utilizing it together with the *read-from* relations to infer and model the real pointer flow of an observed trace, as shown in Figure 3(a). If an LLVM IR variable paired with $e_a.addr/e_a.val$ *originates from* a *load* instruction paired with a read event e_b, we say e_a gets its address/value from e_b and denote it as $GAF_\sigma(e_a)/GVF_\sigma(e_a) = e_b$. We say that an LLVM IR variable v **originates from** an LLVM IR *load* instruction I if it is either:

(1) the same variable or copied from the load result of I,
(2) type-cast or offset-addressed from the load result of I

For example, in Figure 5, the variable paired with $e_{10}.val$ (%4 of foo()) is *directly* the same variable that stores the load result of e_4, so we have %4 originates from I_4 and thus $GAF(e_{10}) = e_4$; the variable paired with $e_4.addr$ (%t of foo()) is *type-cast* from %3, while %3 the variable that stores the load result of I_3, so we have *%t originates from* I_3 and thus $GAF(e_4) = e_3$.

Algorithm 1 computes the *get-from* relation. To calculate where an event e gets $e.addr/e.val$ from in an observed trace, ConPTA calls this algorithm with an LLVM IR variable v as the argument, where v is paired with the given $e.addr/e.val$. The algorithm will finally return a read event e^* from which e gets $e.addr/e.val$; or $\perp$ if such $e.addr/e.val$ is not dynamically read from the memory at runtime (thus e^* does not exist), e.g., the values of static LLVM IR variables (the ones prefixed with an @) can be calculated at link time thus LLVM does not generate load instructions for them.

The insight of Algorithm 1 is as follows. After stripping the type-casting and offset-addressing instructions (Lines 1-4), an LLVM IR variable can be assigned by one of the four types of instructions: (1) a load instruction, (2) a function parameter, (3) a function call instruction, or (4) an initialization instruction of a global static variable. In the following, we take Figure 5 to illustrate Algorithm 1. In the first case (Lines 6-7), Algorithm 1 directly identifies the *get-from* relation from the LLVM def-use chain, e.g., it has $GAF(e_2) = getOrigin(\%1 \text{ of foo}()) = e_1$. In the second case (Lines 8–12), it tracks back to the caller function to get the assignment instruction. Take $e_6.addr$ as an example, it is paired with %0 of $goo()$, which is the first function parameter of $goo()$. Algorithm 1 further finds that e_6 executes between e_5 and e_9, thus e_6 is inside the function $goo()$. Event e_5 implies that function $goo()$ is called by instruction I_5, and I_5 gets its first parameter %2 from I_2. Thus, Algorithm 1 infers that $GAF(e_6) = getOrigin(\%0 \text{ of goo}()) = getOrigin(\%2 \text{ of foo}()) = e_2$. In the third case (Lines 13-17), Algorithm 1 tracks to the callee function. Take e_{10} as an example, $e_{10}.addr$ is paired with variable %5, which is the returned variable of instruction I_5. I_5 calls the function $goo()$, and in $goo()$ the return instruction is I_9 and the returned variable is %4. Aftering calculating out $getOrigin(\%4) = e_8$, Algorithm 1 has $GAF(e_{10}) = getOrigin(\%5 \text{ of foo}()) = getOrigin(\%4 \text{ of goo}()) = e_8$. In other cases (Line 18), the variable v is a static variable,

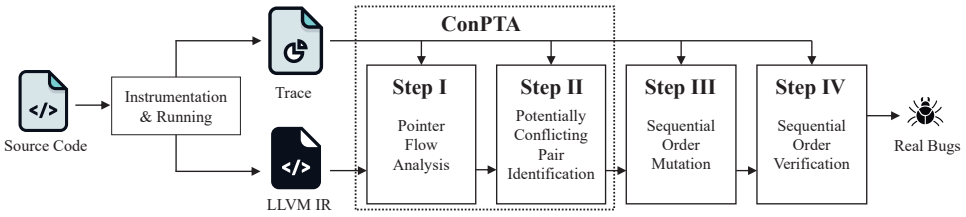


Figure 4: An overview of Eagle.

```
    define i32* @goo(i32* %0, i32 %1) {
I6    store i32 %1, i32* %0                          e6   wr(%0: 0x200, %1: 2)
I7    %3 = load i32**, i32*** @p                     e7   rd(@p: 0x1000, %3: 0x100)
I8    %4 = load i32*, i32** %3                       e8   rd(%3: 0x100, %4: 0x200)
I9    ret i32* %4                                    e9   exit("goo")
    }
    define void @foo() {
I1    %1 = load i32**, i32*** @p                     e1   rd(@p: 0x1000, %1: 0x100)
I2    %2 = load i32*, i32** %1                       e2   rd(%1: 0x100, %2: 0x200)
I3    %3 = load i8*, i8** @q                         e3   rd(@q: 0x1000, %3: 0x300)
      %t = bitcast (i8* %3 to i32*)
I4    %4 = load i32, i32* %t                         e4   rd(%3: 0x300, %4: 2)
I5    %5 = call i32* @goo(i32* %2, i32 %4)           e5   enter("goo")
I10   store i32 4, i32* %5                           e10  wr(%5: 0x200, constant: 4)
I11   ret void                                       e11  exit("foo")
    }
```

Figure 5: An example to illustrate the get-from relations. Blue notations mark the paired LLVM IR variables.

which is always initialized statically. Algorithm 1 sets the result as $\perp$ because in such a case v cannot originate from a load instruction.

Pointer Flow Sequence. With the knowledge of *get-from* and *read-from* relations, ConPTA can then infer the real pointer flow of an observed trace. We use an event sequence $PF(e) = \langle e_1, e_2, \ldots, e_n, e\rangle$, namely the **pointer flow sequence** of event e, to model the real pointer flow. It can be viewed as an adapted version of the *thin slice* for $e.addr$ except that $PF(e)$ is defined on events while the original *thin slice* [65] is defined on statements. $PF(e)$ depicts a chain of write/read events that compute (the first event in $PF(e)$) and copy (all intermediate events except for the first and last) a pointer value to $e.addr$. It should satisfy all the following rules:

(1) **Get-address-from rule**: $GAF(e) = e_n$;
(2) **Read-from rule**: for any event e_k that $n-k$ is an even number, $e_{k-1} = RF(e_k)$ if $RF(e_k) \neq \perp$;
(3) **Get-value-from rule**: for any event e_k that $n-k$ is an odd number, $e_{k-1} = GVF(e_k)$ if $GVF(e_k) \neq \perp$;
(4) **Longest rule**: $PF(e)$ should be the longest sequence that can satisfy the above three rules.

Given an event e, $PF(e)$ can be inferred backward by applying the first three rules recursively (see Lines 7–13 in Algorithm 2). We only stop backward inferring when the sequence cannot be made longer (the *longest rule*) and only mark the final result as $PF(e)$.

With the above design, ConPTA can extract the pointer flow information. It next identifies *potentially conflicting* pairs.

4.2 Potentially Conflicting Pair Identification

Pointer Flow Intersection. With the knowledge of $PF(e)$, the chain through which e obtains the value for $e.addr$, ConPTA can identify *potentially conflicting* relations by detecting where $PF(e_a)$ and $PF(e_b)$ have an **intersection**. We say $PF(e_a)$ and $PF(e_b)$ have an **intersection** if they satisfy one of the following conditions:

(1) e_b is a write event, $\exists\ e_a^* \neq e_a \in PF(e_a)$, where e_a^* is a read event and $e_a^*.addr = e_b.addr$;
(2) $\exists\ e_a^* \neq e_a \in PF(e_a), e_b^* \neq e_b \in PF(e_b)$, where one is a write event and another is a read event and $e_a^*.addr = e_b^*.addr$;

Algorithm 1: getOrigin(v)

```
1  I_d ← the instruction that assigns v with a value (where v first occurs)
2  while I_d is a type-casting or offset-addressing instruction do
3      v ← the variable before type-casting or offset-addressing
4      I_d ← the instruction that assigns v with a value (where v first occurs)
5  switch I_d do
6      case load instruction do
7          return the paired read event of this load instruction
8      case function parameter do
9          x ← index of I_d in the parameter list
10         I_c ← LLVM IR instruction that creates the function call
11         v* ← the LLVM IR variable of x-th parameter in I_c
12         return getOrigin(v*)
13     case function call instruction do
14         F ← function called in I_d
15         I_r ← return instruction of F
16         v* ← the variable of the returned value in I_r
17         return getOrigin(v*)
18     otherwise do return ⊥
```

The first condition examines whether $e_a.addr$ can be read from the same address written to by e_b (thus satisfying the first rule of *potentially conflicting*). The second examines whether the program reads $e_a.addr$ and $e_b.addr$ from the same address (thus satisfying the second rule of *potentially conflicting*).If $PF(e_a)$ and $PF(e_b)$ have an *intersection*, (e_a, e_b) can form a *potentially conflicting* pair.

Algorithm 2 shows the pseudo-code that examines whether (e_a, e_b) can form a *potentially conflicting* pair. It first enumerates $PF(e_a)$ and $PF(e_b)$ to check whether the first condition can be satisfied in Lines 3 and 4 respectively. Due to symmetry property, Algorithm 2 also checks (e_b, e_a). For each event e^* in $PF(e)$ except e, the function *TrackBackSingle* checks whether e^* and e' can satisfy the first or second intersection condition (Line 19). Meanwhile, it also fills two sets W and R on the fly (Lines 16 and 18). These two sets are used to store the values of $e^*.addr$ for write and read events $e^* \in PF(e) \setminus \{e\}$ respectively. They will be used at Line 5 to check whether the third intersection condition can be satisfied.

We further provide a running example with Figure 3(a). While examining the pair $(e_a = e_{15}, e_b = e_{13})$, Algorithm 2 finds that $W_a = \{e_0.addr\} = \{0x100\}$, $R_a = \{e_1.addr\} = \{0x100\}$, $W_b = \{e_4.addr\} = \{0x100\}$, and $R_b = \{e_5.addr, e_3.addr\} = \{0x100, 0x1100\}$. The condition $(W_a \cap R_b) \cup (W_b \cap R_a) \neq \varnothing$ at Line 5 is true and ConPTA concludes that (e_{15}, e_{13}) is a *potentially conflicting* pair.

Algorithm 2: isConflicting

```
1  Function isConflicting(e_a, e_b):
2      W_a ← ∅, W_b ← ∅, R_a ← ∅, R_b ← ∅
3      if TrackBackSingle(e_a, e_b, W_a, R_a) then return True
4      if TrackBackSingle(e_b, e_a, W_b, R_b) then return True
5      if (W_a ∩ R_b) ∪ (W_b ∩ R_a) ≠ ∅ then return True
6      return False
7  Function PF(e):
8      s ← [e], r ← GAF(e)
9      while r ≠ ⊥ do
10         s ← [r] ∘ s, w ← RF(r)
11         if w ≠ ⊥ then s ← [w] ∘ s, r ← GVF(w)
12         else r ← ⊥
13     return s
14 Function TrackBackSingle(e, e', W, R):
15     foreach e* ∈ PF(e)\{e} do
16         if e* is a write event then W.insert(e*.addr)
17         if e* is a read event then
18             R.insert(e*.addr)
19             if e*.addr = e'.addr ∧ e' is a write event then return True
20     return False
```

4.3 Sequential Order Mutation

During the *sequential order mutation* process, Eagle focuses on **bug-triggering** sequential order, following which the program will really trigger a bug. Given a *potentially conflicting* pair (e_a, e_b), it first actively reorders the pointer flow trying to construct sequential order where $e_a.addr$ and $e_b.addr$ have the same value and thus being *conflicting* (except UPU bugs whose semantics requires the opposite). Then, Eagle further reassigns the occurrence order of some critical events according to bug semantics to generate the *bug-triggering* sequential order. We present such *bug-triggering* sequential order in the form of a list of sequences $s = [\rho_1, \rho_2, \dots]$. A sequence $\rho = \langle e_1, e_2 \dots, e_n \rangle$ wrapped by a pair of angle brackets indicates a reordered *pointer flow sequence*; and a sequence $\rho = (e_a, e_b)$ consisted of two events and wrapped by a pair of round brackets indicates occur-before relation e_a occurring before e_b required by the bug semantics. Note, these notations are only used for presentation; the sequences are treated the same by the EagleSeqC algorithm. Some bug semantics may have special requirements for event types. Eagle skips the pair if special requirements cannot be satisfied. In the following, we present the *bug-triggering* sequential order constructing algorithms for each bug type.

NPD bugs. The NPD bug semantics requires one of the two *potentially conflicting* events (denoted as e_a) to be a write event and $e_a.val = NULL$, and the other event (denoted as e_b) uses this NULL as $e_b.addr$. Eagle chooses the **last** read event $e^* \in PF(e_b) \setminus \{e_b\}$ that $e_a.addr = e^*.addr$ and then reorders e^* to *read from* e_a, letting the NULL value flows from e_a to $e_b.addr$. The reordered pointer flow sequence is $PF^*(e_b) = \langle e_a$, suffix of $PF(e_b)$ starting with $e^* \rangle$ and the *bug-triggering* sequential order is $s = [PF^*(e_b)]$.

UPU bugs. Given a read/write/acquire/release/free event e_a, the UPU bug semantics requires e_a to occur before all other write events (in the same trace) that may generate values for $e_a.addr$ (thus $e_a.addr$ is uninitialized). Eagle assigns all write events (denoted as $e_{b1}, e_{b2}, \dots$) that there exists a read event $e^* \in PF(e_a) \setminus \{e_a\}$ where $e^*.addr = e_{b*}.addr$ to occur after e_a and gets the reordered pointer flow sequence as $PF^*(e_a) = \langle$ suffix of $PF(e_a)$ starting with $e^* \rangle$. The *bug-triggering* sequential order as $s = [PF^*(e_a), (e_a, e_{b1}), (e_a, e_{b2}), \dots]$.

UAF bugs. The UAF bug semantics requires one free event (denoted as e_a) to occur first and another event (denoted as e_b) to occur later and access the same object with e_a. Given a potentially conflicting pair (e_a, e_b) where e_a is a free event and the other accesses an object, Eagle finds the **last** two write events $e_a^* \in PF(e_a) \setminus \{e_a\}$ and $e_b^* \in PF(e_b) \setminus \{e_b\}$ that $e_a^*.addr = e_b^*.addr$. Then it chooses e^* within e_a^* and e_b^* that occurs earlier in the observed trace and reorders the pointer flow sequences as $PF(e_a)^* = \langle e^*$, suffix of $PF(e_a)$ after $e_a^* \rangle$ and $PF(e_b)^* = \langle e^*$, suffix of $PF(e_b)$ after $e_b^* \rangle$, letting the value flows from e^* to both e_a and e_b. Finally, it assigns e_a to occur before e_b to trigger the UAF bug. The *bug-triggering* sequential order is $s = [PF(e_a)^*, PF(e_b)^*, (e_a, e_b)]$.

DF bugs. The DF bug semantics requires both the two *potentially conflicting* events to be free events. DF bugs can be viewed as a special type of UAF bugs: we treat the second free event as an *use* event and predict DF bugs in the same way as predicting UAF bugs.

A running example for UAF. In Figure 3(c), given the pair (e_{15}, e_{13}), $PF(e_a) = \langle e_0, e_1, e_{15} \rangle$ and $PF(e_b) = \langle e_3, e_4, e_5, e_{13} \rangle$, Eagle finds that $e_a^* = e_0$ and $e_b^* = e_4$. Then, it chooses $e^* = e_0$ and reorders the pointer flow sequences as $PF^*(e_{15}) = \langle e_0, e_1, e_{15} \rangle$ and $PF^*(e_{13}) = \langle e_0, e_5, e_{13} \rangle$. Besides, it assigns e_{13} to occur before e_{15}. Thus, the *bug-triggering* sequential order is $[PF^*(e_{15}), PF^*(e_{13}), (e_{13}, e_{15})]$.

4.4 Sequential Order Verification

A *bug-triggering* sequential order is a sufficient condition for triggering that bugs. However, such a sequential order may be *infeasible* because (1) the reordered *pointer flow sequence* may be unrealizable, and (2) the *occur-before* relations required by bug semantic may have conflicts with existing *read-from* relations and/or the lock semantics. To address this problem, Eagle relies on a sequential order verifier to check the feasibility. We then propose a new sound sequential order verification algorithm EagleSeqC. EagleSeqC is developed based on SeqCheck and shares a similar framework with it. We adapt SeqCheck in sub-algorithms to meet our requirements. To make it easier to understand, we first introduce the original SeqCheck algorithm and then illustrate EagleSeqC.

SeqCheck. The core algorithms of SeqCheck are shown in Algorithm 3 (please ignore the blue lines which are modifications made by EagleSeqC). These algorithms can verify whether a given sequence ρ is **feasible**, i.e., can be realized by real program execution where the occurrence of events follows this sequence.

SeqCheck implements this verification by reasoning on a directed graph G. It models each event e as a node $N(e)$ and each relation that e_a occurs before e_b as an edge $\langle N(e_a), N(e_b) \rangle$ (denoted as as $e_a \prec_G e_b$). Algorithm 3 starts by inserting all edges in ρ into the graph G (Lines 7-8). Then, it tries to serialize the graph into a sequence (Line 9, denoted **witness trace**) using a greedy algorithm (which may insert edges into graph G, see its paper [10] for details). During this process, for each inserted edge, it calculates a closure of it to (1) preserve the read-from relations (Line 18) and (2) prevent lock areas from overlapping (Line 19). If a cycle is formed (Line 16), it detects a conflict between edges in ρ and *read-from* relations and/or lock semantics. Thus, it reports the given sequence ρ infeasible. Otherwise, if the serialization is finished without forming any cycle, Algorithm 3 reports ρ feasible (Line 10). This algorithm is sound [10]: if it reports a sequence ρ feasible, then ρ is truly feasible, i.e., can be realized by real program execution.

Algorithm 3: EagleSeqC

```
1  Function EagleSeqC(σ, ρ, s):
2      initialize graph G
3      RF ← ∅
4      foreach sequence ρ ∈ s do
5          foreach read-from relation ⟨w, r⟩ ∈ ρ do RF ← RF ∪ {⟨w, r⟩}
6      foreach sequence ρ ∈ s do
7          for i = 0 to len(ρ) − 2 do
8              InsertAndClose(G, ρ[i], ρ[i + 1], RF)
9      greedily serialize graph G to a sequence (may insert new edges)
10     report FEASIBLE
11 Function InsertAndClose(G, e1, e2, RF):
12     q ← empty queue
13     q.push(⟨e1, e2⟩)
14     while q not empty do
15         ⟨ea, eb⟩ ← q.pop()
16         if eb ≺G ea then report INFEASIBLE
17         G.insert(⟨ea, eb⟩)
18         q.push(ObsClosure(G, ea, eb))
19         q.push(LockClosure(G, ea, eb))
20         q.push(PointerFlowClosure (G, ea, eb, RF))
21 Function ObsClosure(G, ea, eb):
22     C ← ∅
23     foreach read event r ∈ G | ∃ a branch event br, r ≺G br do
24         w ← RF(r)
25         if ∃ w* ∈ G | w.addr = w*.addr ∧ w* ≺G ea ∧ eb ≺G r then
26             C ← C ∪ {⟨w*, w⟩}
27         if ∃ w* ∈ G | w.addr = w*.addr ∧ w ≺G ea ∧ eb ≺G w* then
28             C ← C ∪ {⟨r, w*⟩}
29     return C
30 Function LockClosure(G, ea, eb):
31     C ← ∅
32     foreach acquire event e ∈ G do
33         if ∃ a release event e* ∈ G | e.addr = e*.addr ∧ e ≺G ea ∧ eb
             ≺G e* ∧ e* is not the realease event of e then
34             rel ← the release event of e
35             acq ← the acquire event of e*
36             C ← C ∪ {⟨ rel, acq ⟩}
37     return C
38 Function PointerFlowClosure(G, ea, eb, RF):
39     C ← ∅
40     foreach ⟨w, r⟩ ∈ RF do
41         foreach write event w* ≠ w that w*.addr = w.addr or
             w*.addr = r.addr do
42             if eb ≺G r and w* ≺G ea then C ← C ∪ {⟨w*, w⟩}
43             if w ≺G ea and eb ≺G w* then C ← C ∪ {⟨r, w*⟩}
44     return C
```

EagleSeqC. EagleSeqC modifies SeqCheck in two aspects to verify the feasibility of the *bug-triggering* sequential order inferred in Section 4.3. First, it extends Algorithm 3 at Line 6 by enumerating the whole list to handle the sequential order consisting of multiple sequences. Second, it proposes a new closure function *PointerFlowClosure()* and adds it into the closure-calculating function *InsertAndClose* at Line 20. This new function explicitly inserts edges to maintain the reordered *read-from* relations specified by the reordered pointer flow sequences, thus maintaining the reordered pointer flow (*GAF* and *GVF* relations are self-maintained by the programming logic). Algorithm 3 first extracts all such *read-from* relations $\langle w, r\rangle$ where r reads from w as RF (Lines 4-5). Then, during the serialization, if the function *PointerFlowClosure()* detects a potential violation, i.e., a write event w^* that $w^*.addr = w.addr$ and has occurred either before r or after w, it may be possible for w^* to occur between w and r thus interrupt the *read-from* relation and violate the reordered pointer flow sequence. Algorithm 3 inserts new edges (Lines 25-28) to force w^* to occur either before or after both w and r to prevent this from happening.

These two modifications help EagleSeqC to insert all edges necessary to maintain the *bug-triggering* sequential order s. EagleSeqC can achieve soundness, i.e., not reporting false positive bugs. In the following, we define and prove the soundness of EagleSeqC.

Theorem 1 (**Soundness of EagleSeqC**). *Given a trace σ and a* bug-triggering *sequential order s, if* EagleSeqC(σ, s) *reports FEASIBLE, then s is truly feasible as there exists real program execution:*

(1) the occurrence order of events follow the sequential order s;
(2) the pointer flow is the same as that suggested by s.

Proof Sketch. The original SeqCheck algorithm (without our modifications colored in blue) has been proved to satisfy the first condition [10]. Since our new closure algorithm *PointerFlowClosure()* only inserts new edges into the graph G without removing any edge from it, the first condition can still hold for EagleSeqC.

For the second condition, real program execution can only follow the pointer flow sequence in s when all the *read-from* and *get-from* relations suggested by it are maintained (Section 4.1). We only need to prove the *read-from* relations are not violated because the *get-from* relations are self-sustained by the programming logic. We show it by induction on the edge insertion operations:

(1) First, given a newly initialized empty graph G, the *read-from* relations are obviously not violated.
(2) Second, when EagleSeqC inserts a new edge $\langle e_a, e_b\rangle$ to G inside the function *InsertAndClose*, it will not cause a new violation. Given a read-from relation $\langle w, r\rangle \in RF$ and a write event w^* that $w^*.addr$ is equal to $w.addr$ or $r.addr$:
   (a) if $\langle e_a, e_b\rangle$ may cause w^* to occur between w and r, *PointerFlowClosure()* inserts additional edges to cause w^* to occur either before w or after r (Lines 25-28). Thus it prevents w^* from interrupting the *read-from* relation $RF(r) = w$;
   (b) otherwise, w^* is still unordered with both w and r after edge inserting and thus cannot interrupt $RF(r) = w$.
(3) Thus, the *read-from* relations in RF are always maintained.

Based on the above two paragraphs, Theorem 1 holds. □

4.5 Time Complexity Analysis

Given n as the trace size, i.e., the number of events in the trace, Step I has an overall time complexity of $O(n^2)$. This is because Algorithm 1 has a time complexity of $O(n)$ and it runs for each address/value at most once. Considering that Eagle examines at most n^2 pairs in Step II, the average time complexity of Step I for predicting one *potentially conflicting* pair is $O(1)$. Step II examines an event pair in $O(n)$: the enumeration in Algorithm 2 (Lines 15-19) has a time complexity of $O(n)$ because each event is examined at most once, and Line 5 has a time complexity of $O(n)$ because the sets W_a/R_a and W_b/R_b have at most n elements. Step III can finish in $O(n)$ within one scan of the observed trace. Step IV has a time complexity of $O(n^2 \log n)$ because the extensions made by EagleSeqC do not increase the time complexity of *InsertAndClosure()* and thus EagleSeqC has the same time complexity as SeqCheck [10]. With the above knowledge, ConPTA (Steps I and II) has a time complexity of $O(n)$ and Eagle $O(n^2 \log n)$.

Table 1: Prediction results of Eagle, UFO, ConVul, ConVulPOE, Period, and Eagle$_n$ on the 10 real-world programs.

Program	Version	LOC	Thread	Eagle	UFO	ConVul	ConVulPOE	Period	Eagle$_n$
pbzip2	v0.9.4	45.9K	5	**5 NPD**, **5 UPU**, **8 UAF**, **1 DF**	4 UAF	1 NPD, 1 UAF, 0 DF	3 NPD, 1 UAF, 0 DF	0	**5 NPD**, **5 UPU**, 4 UAF, 0 DF
pixz	v1.0.7	89.8K	6	0 NPD, **1 UPU**, 0 UAF, **1 DF**	0 UAF	0 NPD, 0 UAF, **1 DF**	0 NPD, 0 UAF, 0 DF	0	0 NPD, **1 UPU**, 0 UAF, 0 DF
pigz	v2.7	107.1K	6	**8 NPD**, **3 UPU**, **4 UAF**, 0 DF	0 UAF	7 NPD, 0 UAF, 0 DF	4 NPD, 1 UAF, 0 DF	0	**8 NPD**, **3 UPU**, 3 UAF, 0 DF
lbzip2	v2.5	300.5K	7	0 NPD, **9 UPU**, 0 UAF, 0 DF	0 UAF	0 NPD, 0 UAF, 0 DF	0 NPD, 0 UAF, 0 DF	0	0 NPD, 2 UPU, 0 UAF, 0 DF
httrack	v3.43.9	528.4K	2	0 NPD, **3 UPU**, 0 UAF, **1 DF**	0 UAF	#Err	0 NPD, 0 UAF, 0 DF	0	0 NPD, **3 UPU**, 0 UAF, **1 DF**
libwebp	v1.2.2	822.2K	3	**4 NPD**, 0 UPU, 0 UAF, **2 DF**	0 UAF	1 NPD, 0 UAF, 0 DF	0 NPD, 0 UAF, 0 DF	0	0 NPD, 0 UPU, 0 UAF, 0 DF
libvpx	v1.11.0	2.1M	4	**4 NPD**, **31 UPU**, 0 UAF, 0 DF	#Err	**4 NPD**, 0 UAF, 0 DF	0 NPD, 0 UAF, 0 DF	0	1 NPD, **31 UPU**, 0 UAF, 0 DF
x264	HEAD: 19856c	632.1K	7	**34 NPD**, **59 UPU**, 0 UAF, **1 DF**	0 UAF	10 NPD, 0 UAF, 0 DF	0 NPD, 0 UAF, 0 DF	0	2 NPD, 47 UPU, 0 UAF, **1 DF**
x265	v3.4	3.0M	10	0 NPD, **23 UPU**, 0 UAF, **1 DF**	0 UAF	0 NPD, 0 UAF, 0 DF	0 NPD, 0 UAF, 0 DF	0	0 NPD, 0 UPU, 0 UAF, **1 DF**
MySQL	v5.7.36	10.9M	27	**17 NPD**, **68 UPU**, 0 UAF, **16 DF**	0 UAF	0 NPD, 0 UAF, 0 DF	0 NPD, 0 UAF, 0 DF	0	16 NPD, 63 UPU, 0 UAF, **16 DF**
Total	N/A	N/A	N/A	**72 NPD**, **202 UPU**, **12 UAF**, **23 DF**	4 UAF	23 NPD, 1 UAF. 1 DF	7 NPD, 2 UAF, 0 DF	0	32 NPD, 155 UPU, 7 UAF, 19 DF

5 EVALUATION

We evaluate the following approaches in our experiments:

- **Eagle**, our approach;
- **UFO** [33], state-of-the-art SMT-based predictive approach;
- **ConVul** [11], a recent graph-based predictive approach;
- **ConVulPOE** [75], an improved offline version of ConVul;
- **Period** [71], a state-of-the-art witness-based approach;
- **Eagle$_n$**, Eagle without pointer flow reordering (ConPTA).

We target to answer the following research questions:

- **RQ1**: can Eagle detect more real concurrency bugs (NPD, UPU, UAF, and DF) than existing bug-detecting approaches UFO, ConVul, ConVulPOE, and Period?
- **RQ2**: what is the efficiency of Eagle in bug-detecting?
- **RQ3**: how can ConPTA benefit Eagle?

We implement the instrumentation tool of Eagle upon LLVM 12.0 as an LLVM Pass and implement Eagle (and ConPTA) in C++. We use the instrumentation tool to instrument programs and then run the instrumented programs to obtain traces and LLVM IR dynamic slices. We utilize the command line interfaces to drive the programs. The inputs of MySQL are generated by its test suites. Following ConVulPOE [75], the inputs of other programs are randomly grabbed from the Internet. We prepare multiple inputs for each program and run each program on each input once in every round. Each input generates one trace. We use the artifacts of ConVul and ConVulPOE received from their authors and those of UFO and Period published [34, 72] to conduct our experiments.

We conduct a pre-experiment on the CVE benchmark [11, 49] to ensure the soundness and correctness of the artifacts before evaluating them on 10 real-world programs. The CVE benchmark is proposed by ConVul [11]. It consists of 10 extracted programs. Each program replicates a known CVE vulnerability. On this benchmark, all artifacts can correctly predict all concurrency bugs without producing any false positive report except that ConVulPOE misses two bugs (consistent with its paper [75]) and Period misses two bugs due to segmentation faults. We try to fix this segmentation fault issue but fail. Since Period does not report segmentation faults on the 10 real-world programs (discussed below), this issue may not weaken its performance in later experiments.

We further conduct our experiments on 10 real-world multi-threaded programs, including four parallel compression utilities (pbzip2, pixz, pigz, lbzip2), one offline browser utility (httrack), one image processing library (libwebp), three video encoders/decoders (libvpx, x264, x265), and one database (MySQL). These programs have been studied by the prior work [13, 33] and are proven to be vulnerable in concurrency security. We conduct the experiments on a Linux server with an Intel Xeon Platinum 8260 CPU and 512GB RAM, and set a time limit of 300 hours (for each program) to handle the endless running issue. All experiments are repeated 5 times.

5.1 RQ1: Effectiveness

Table 1 shows the program information on the left and the number of **unique** bugs (bugs with different code locations) reported by each artifact on the right. Period exceeds the hard time limit of 300 hours and produces no output on all ten programs, thus we mark all its results as zeros. UFO and ConVul abort without producing any debug information on *libvpx* and *httrack*, thus we mark their results on these two programs as *#Err*. Overall, Eagle can predict more bugs than UFO, ConVul, and Period. Eagle can find bugs on all 10 programs, while UFO can only find bugs on 1 program, ConVul on 6 programs, ConVulPOE on 2 programs, and Period on 0 program. In addition, Eagle can predict UPU bugs. It finds 202 UPU bugs on these 10 programs. The above facts show that Eagle has an advantage over existing approaches on effectiveness.

Answer to RQ1: Eagle can predict more real concurrency bugs: on 10 real-world programs, it finds 3 times more UAF bugs than UFO, 4.28 and 10.89 times more NPD/UAF/DF bugs than ConVul and ConVulPOE, and many more bugs than Period apart from 202 UPU bugs.

5.2 RQ2: Efficiency

Table 2 shows the time consumption of Eagle, UFO, ConVul, and Period for bug detection. We list the **average** number of read/write events (N_{rw}), acquire/release events (N_l), allocate/free events (N_m), enter/exit events (N_f) and branch events (N_{br}) **in one trace**, and the number of traces (#) tested **in one round** for reference. We calculate the speedup that Eagle achieves over UFO and ConVul in Column *SpeedUp*. Compared with UFO, Eagle achieves a speedup of 5.16x on average, and 7.30×10^5x at most. It runs only a bit slower than UFO on three programs *lbzip2*, *httrack* and *MySQL*, but finds 11 times more UAF bugs (Table 1). As for ConVul and ConVulPOE, they run much faster than Eagle as their time consumption is about two and three orders of magnitude lower than Eagle. This is because they target at fast speed (but may miss bugs). To fairly compare Eagle with them, we give them the time budget of Eagle and combine them with AFL++ [77]: we utilize AFL++ to generate

Table 2: The time consumption of Eagle, UFO, ConVul, ConVulPOE, and ConPTA on the 10 real-world programs.

Program	N_{rw}	N_l	N_m	N_f	N_{br}	#	Eagle	UFO	SpeedUp	ConVul	SpeedUp	ConVulPOE	SpeedUp	Period	ConPTA	Prop
pbzip2	1.57K	84	29	212	258	10	0.04s	4m52s	7.30×10^{5}	3.06s	7.65×10^{1}	10.06s	2.52×10^{2}	>300h	0.01s	25.0%
pixz	31.48K	1.14K	2.17K	3.16K	4.72K	90	0.39s	33.19s	8.51×10^{1}	57.20s	1.47×10^{2}	2m05s	3.21×10^{2}	>300h	0.02s	5.13%
pigz	25.27K	2.70K	1.36K	10.44K	33.41K	50	5.00s	20h59m21s	1.1×10^{4}	27.83s	5.57×10^{0}	5.84s	1.17×10^{0}	>300h	0.01s	0.20%
lbzip2	126.65M	462	603	437.64K	16.73M	55	12m55s	3m20s	2.58×10^{-1}	12.20s	1.57×10^{-2}	4m11s	3.24×10^{-1}	>300h	3m13s	24.90%
httrack	19.76M	1.60K	66.10K	3.91M	26.82M	5	58.43s	0.39s	6.67×10^{-3}	#Err	#Err	36.68s	6.68×10^{-1}	>300h	8.57s	14.67%
libwebp	115.74M	564	1.69K	3.16M	5.07M	55	6m52s	141h41m59s	1.24×10^{3}	2m00s	2.91×10^{-1}	1m17s	1.87×10^{-1}	>300h	1m21s	19.63%
libvpx	98.56M	8.37K	11.12K	5.40M	38.88M	90	1h06m40s	#Err	#Err	25m26s	3.83×10^{-1}	2m40s	4.00×10^{-2}	>300h	1m16s	1.90%
x264	152.13M	7.13K	49.23K	4.86M	38.92M	70	1h03m30s	>300h	$>2.83\times10^{2}$	8m01s	1.26×10^{-1}	4m47s	7.01×10^{-2}	>300h	7m34s	11.93%
x265	140.33M	4.47K	11.52K	18.22M	68.63M	20	2h09m37s	>300h	$>1.39\times10^{2}$	1h34m45s	7.31×10^{-1}	6m52s	5.30×10^{-2}	>300h	3m03s	2.35%
MySQL	1.61B	9.21M	859.56K	880.33M	1.15B	31	169h40m11s	136h38m23s	8.05×10^{-1}	54m13s	5.33×10^{-3}	4m36s	4.52×10^{-4}	>300h	31m57s	0.31%
Total	2.26B	9.24M	1.00M	916.33M	1.38B	476	174h20m56s	>899h48m28s	>5.16	3h06m05s	1.78×10^{-2}	27m22s	2.62×10^{-3}	>3000h	48m33s	0.46%

Table 3: The results of ConVul and ConVulPOE given Eagle's time budget.

Program	Eagle	ConVul	Round	Diff	ConVulPOE	Round	Diff
pbzip2	0.04s	3.07s	1	-	10.16s	1	-
pixz	0.39s	57.03s	1	-	2m06s	1	-
pigz	5.00s	27.55s	1	-	5.65s	1	-
lbzip2	12m55s	13m06s	65	-	16m38s	4	-
httrack	58.43s	#Err	#Err	#Err	1m10s	2	-
libwebp	6m52s	8m04s	4	-	7m48s	6	-
libvpx	1h06m40s	1h14m52s	3	-	1h09m00s	26	-
x264	1h03m30s	1h04m29s	8	-	1h03m34s	13	-
x265	2h09m37s	3h10m40s	2	-	2h16m29s	20	-
MySQL	169h40m11s	169h42m02s	188	-	169h45m55s	2324	-
Total	174h20m56s	175h34m40s	≈30	-	174h42m56s	≈240	-

different observed traces and repeatedly run them on different traces until running out of time budget. The experimental results are presented in Table 3: ConVul and ConVulPOE cannot make any progress by running on more observed traces. Thus, they can be less efficient than Eagle in terms of not being able to take advantage of a larger time budget to detect more bugs. As for Period, however, it exceeds the hard time limit on all 10 programs without reporting any bug. Thus, it is significantly less efficient than Eagle.

Answer to RQ2: Eagle is significantly more efficient than UFO and Period because it predicts more bugs within less time. It runs about two or three orders of magnitude slower than ConVul or ConVulPOE but predicts 11.36 or 33.34 times more bugs in return, which cannot be achieved by them even if given more time budget.

5.3 RQ3: Evaluation of ConPTA

ConPTA benefits Eagle in two aspects. First, it benefits Eagle to detect more concurrency bugs with little additional time consumption. In Table 1, Eagle$_n$ finds 32 NPD, 155 UPU, 7 UAF, and 19 DF bugs, while Eagle finds 72 NPD, 202 UPU, 12 UAF, and 23 DF bugs. That is, ConPTA helps Eagle find 125% more NPD, 30.3% more UPU, 71.4% more UAF and 21.1% more DF bugs. As for time consumption, ConPTA spends 49 minutes (0.46% of Eagle's total time) on pointer flow analysis (see the two rightmost columns in Table 2). Thus, ConPTA is efficient in helping Eagle reorder the pointer flow to detect more concurrency bugs.

Second, while identifying more valid *bug candidates* to help Eagle detect more bugs, the ratio of candidates that are not verified as real bugs (denoted as the false positive rate of *bug candidates*) for ConPTA is still at the same level with or even significantly lower than existing approaches. In Table 4, we list the number of *bug candidates* identified by each approach on each of the ten tested programs. Note, the two online approaches ConVul and Period do not explicitly identify bug candidates, thus we mark their results as *N/A*. Overall, ConPTA identifies 43,424 *bug candidates* where 309 are verified as real bugs; the false positive rate is 99.29%. In comparison, UFO identifies 1,814,393 *bug candidates* (including the ones caused by memory reuse; UFO filters them in later SMT solving process) and 4 of them are verified as real bugs, the false positive rate is higher than 99.99%; ConVulPOE identifies 1,554 *bug candidates* (excluding the memory reuse situation) and 9 of them are verified as real bugs, thus the false positive rate is 99.42%; Eagle$_n$ identifies 43,358 *bug candidates* and 213 of them are verified as real bugs, thus the false positive rate is 99.51%. The false positive rate of *bug candidates* for ConPTA is significantly lower than UFO, and it is at the same level with ConVulPOE and Eagle$_n$, but ConPTA benefits Eagle to outperform them by identifying more valid *bug candidates* and thus help Eagle detect significantly more bugs.

Table 4: The number of bug candidates.

Program	Eagle	UFO	ConVul	ConVulPOE	Period	Eagle$_n$
pbzip2	244	202	N/A	66	N/A	269
pixz	6,567	1,905	N/A	0	N/A	2,924
pigz	2,496	9,289	N/A	1,363	N/A	4,459
lbzip2	2,562	186	N/A	66	N/A	2,743
httrack	2,015	0	N/A	50	N/A	1,769
libwebp	4,637	19,859	N/A	9	N/A	2,815
libvpx	1,700	#Err	N/A	0	N/A	1,687
x264	7,044	1,180,993	N/A	0	N/A	6,178
x265	2,763	588,052	N/A	0	N/A	5,395
MySQL	13,396	13,906	N/A	0	N/A	15,119
Total	43,424	1,814,393	N/A	1,554	N/A	43,358

Answer to RQ3: the point-to analysis technique ConPTA can help Eagle detect significantly more concurrency bugs both at a small additional time cost and without increasing the false positive rate of bug candidates.

5.4 Case Study

On real-world programs, Eagle finds several new bugs. We randomly inspect 5 on pbzip2 and 5 on MySQL. These bugs have been confirmed by CNVD [52] and/or developers and fixed, where one pbzip2 bug is a variant of the other four and five MySQL bugs are related to the same removed code file. We present bug IDs in Table 5. We analyze a new NPD bug found on MySQL to show how pointer flow reordering helps Eagle to detect the bug.

We illustrate the root cause of this bug in Figure 6. This bug involves two functions *ut_list_remove()* and *ut_list_prepend()*. Due to

Table 5: Status of the bugs found by EAGLE.

Program	Bug ID	Type	Status	EAGLE	UFO	CONVUL	CONVULPOE
pbzip2	2022-88832	UAF	Confirmed, Fixed	✓	✗	✗	✗
pbzip2	2022-88833	NPD	Confirmed, Fixed	✓	-	✗	✗
pbzip2	2022-88834	UAF	Confirmed, Fixed	✓	✗	✗	✗
pbzip2	2022-88835	NPD	Confirmed, Fixed	✓	-	✗	✗
MySQL	#106471	NPD	Code Removed	✓	-	✗	✗

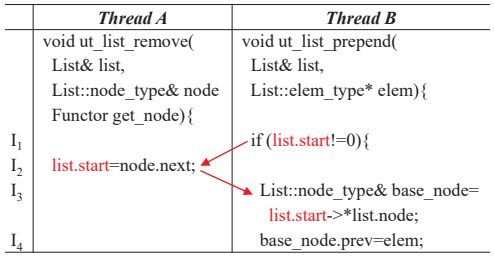


Figure 6: An NPD bug found by EAGLE in MySQL.

improper synchronization, the two functions may update the workload list *list* concurrently, which may cause an NPD bug. Thread A assigns the pointer *list.start* in I_2, where *node.next* can be NULL in some cases. Thread B accesses a workload via pointer *list.start* in I_3 after confirming that *list.start* is not NULL in I_1. Thread A can assign *list.start* to NULL after thread B finishes its check in I_1 but before it accesses the workload in I_3. In that case, reference *base_node* will be assigned as NULL and the program will trigger an NPD bug at I_4. Thanks to our pointer flow reordering technique, EAGLE can actively reorder the pointer flow to explore the thread interleaving where I_3 read from I_2, and thus it can find this bug.

6 RELATED WORK

Data Race Detection. Among concurrency bugs, data race is a research hotspot. Many data race detectors [4–6, 19, 20, 22, 25, 55, 61–63, 67] are based on the *happens-before* [40] relation, and some others [17, 51, 60, 68, 76] are based on the *lockset discipline* [60]. The famous Google ThreadSanitizer [30] further combines the two techniques to achieve efficient race detection. However, these approaches are usually unsound. SHB [46], CP [64], WCP [38], DC [58], SyncP [48], M2 [54] are six recently proposed sound race-detecting approaches, with M2 being the latest and state-of-the-art one. SeqCheck [10] further extends M2 to soundly predict deadlocks and atomicity violations. They inspire the design of EAGLE.

Fuzzing. Fuzzing tools like AFL [77] are helpful in concurrency bug detection. Witness-based concurrency bug detectors can utilize fuzzing techniques to achieve higher effectiveness [13, 73]. Furthermore, they can adopt *static analysis*, *symbolic execution* and/or *taint propagation* to improve code coverage and/or bypass invalid cases [15, 29, 57, 59]. Besides, *stress testing* and *controlled concurrency testing* can be applied to increase thread parallelism [1, 8, 26, 56, 66, 70, 71]. Predictive approaches including EAGLE can also adopt these fuzzing-like techniques for further improvements, e.g., PredFuzz [31] proposes to use fuzzing techniques to improve the path coverage and concurrency coverage for SeqCheck.

Static Analysis. Several bug detectors, e.g., DCUAF [3], rely on static analysis to detect concurrency bugs. By integrating the SMT (Satisfiability modulo theory) solving technique, they can further gain better effectiveness. For example, Canary [9] integrates the SMT-solving technique in its value flow analysis, and ZORD [32] utilizes such a technique in its multi-threaded program verification process. Recently, ToccRace [79] proposes to use static analysis to find the fix-point events for tolerating control flow changes when predicting concurrency bugs. EAGLE may also adopt these static analysis techniques to improve the effectiveness and efficiency of its pointer flow analysis.

Point-to Analysis. Recently, K-miner [27] and RAZZER [36] novelly propose to use the point-to analysis technique in the bug detection area. K-miner proposes to use an inter-procedural and context-sensitive point-to analysis technique to detect memory corruption vulnerabilities in the commodity operating systems, and RAZZER further extends this technique to identify the *race candidates* and then detect the data races in the Linux kernel. They inspire EAGLE to propose CONPTA to predict *potentially conflicting* pairs.

Active Delay Injection. The active delay injection techniques inject delays at potentially buggy points aiming to drive the threads toward making conflicting accesses. Witness-based approaches [21, 42, 53] can utilize it to enlarge the thread interleaving coverage, but predictive approaches seem to be incompatible with it.

7 CONCLUSION

We propose a new idea on actively reordering pointer flow to explore more possible thread interleaving. Based on it, we design and implement a new sound predictive bug-detecting approach EAGLE. We evaluate EAGLE on 10 real-world programs. The evaluation results show that EAGLE outperforms four state-of-the-art bug-detecting approaches UFO, CONVUL, CONVULPOE, and PERIOD in both effectiveness and efficiency. As a modularly designed technique, EAGLE can be extended to detect other types of bugs by re-designing its sequential order mutation and verification parts, and CONPTA can be embedded into other existing bug-detecting techniques for enhancing effectiveness and/or efficiency. In the future, we aim to develop EAGLE (and CONPTA) as a generic concurrency bug detecting platform.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their insights on improving this work. This work is supported in part by the National Key Research and Development Program of China (No. 2022YFB3104004), National Natural Science Foundation of China (NSFC) (Grant No. 61932012, 62132020, 62132020, 62232016, 62372437), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. Y2021041 and 2019111).

REFERENCES

[1] Mahmoud Abdelrasoul. 2017. Promoting secondary orders of event pairs in randomized scheduling using a randomized stride. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 741–752. https://doi.org/10.1109/ASE.2017.8115685

[2] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. 2015. Runtime Pointer Disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 589–606. https://doi.org/10.1145/2814270.2814285

[3] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective Static Analysis of Concurrency Use-after-Free Bugs in Linux Device Drivers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 255–268.

[4] Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. 2017. Lightweight data race detection for production runs. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC'17)*. Association for Computing Machinery, ustin, TX, USA, 11–21.

[5] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: Efficient, Software-Only Region Conflict Exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 241–259. https://doi.org/10.1145/2814270.2814292

[6] 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.

[7] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 213–223.

[8] Yan Cai and Zijiang Yang. 2016. Radius Aware Probabilistic Testing of Deadlocks with Guarantees. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 356–367.

[9] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1126–1140.

[10] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and Efficient Concurrency Bug Prediction. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 255–267.

[11] 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.

[12] US-CERT Security Operations Center. Mar. 2023 (last accessed). National Vulnerability Database (NVD). https://nvd.nist.gov.

[13] 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.

[14] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. *Detecting Memory Errors at Runtime with Source-Level Instrumentation*. Association for Computing Machinery, New York, NY, USA, 341–351.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278.

[16] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. 2022. The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 153 (oct 2022), 25 pages. https://doi.org/10.1145/3563316

[17] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. 2002. Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 258–269. https://doi.org/10.1145/512529.512560

[18] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. 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, 181–195. https://doi.org/10.1145/3192366.3192388

[19] Laura Effinger-Dean, Brandon lucia, luis Ceze, Dan Grossman, and Hans-J Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *Acm International Conference on Object Oriented Programming Systems Languages & Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*.

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

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

[22] 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)*. 151–162.

[23] 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.

[24] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development (SecDev)*. 23–30. https://doi.org/10.1109/SecDev45635.2020.00019

[25] 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)*. ACM, New York, NY, USA, 121–133.

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

[27] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux. https://doi.org/10.14722/ndss.2018.23331

[28] Phillip B. Gibbons and Ephraim Korach. 1997. Testing Shared Memories. *SIAM J. Comput.* 26, 4 (aug 1997), 1208–1244. https://doi.org/10.1137/S0097539794279614

[29] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing: SAGE Has Had a Remarkable Impact at Microsoft. *Queue* 10, 1 (Jan. 2012), 20–27.

[30] Google. Mar. 2023 (last accessed). ThreadSanitizer. https://clang.llvm.org/docs/ThreadSanitizer.html/.

[31] Yuqi Guo, Zheheng Liang, Shihao Zhu, Jinqiu Wang, Zijiang Yang, Wuqiang Shen, Jinbo Zhang, and Yan Cai. 2023. Sound Predictive Fuzzing for Multi-threaded Programs. In *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 810–819. https://doi.org/10.1109/COMPSAC57700.2023.00110

[32] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo Ordering Consistency Theory for Multi-Threaded Program Verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1264–1279.

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

[34] Jeff Huang. Mar. 2023 (last accessed). UFO. https://github.com/parasol-aser/UFO.

[35] 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.

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

[37] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages.

[38] 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.

[39] 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

[40] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[41] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. 2021. ExpRace: Exploiting Kernel Races through Raising Interrupts. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2363–2380.

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

[43] Linux developers. Mar. 2023 (last accessed). Dirty Cow (CVE-2016-5195) . https://dirtycow.ninja/.

[44] 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

[45] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339.

[46] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 145 (oct 2018), 29 pages.

[47] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727.

[48] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (Jan. 2021), 29 pages.

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

[50] 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* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 22–31.

[51] Hiroyasu Nishiyama. 2004. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium (VM '04)*. New York, NY, USA, 127–138.

[52] National Computer Network Emergency Response Technical Team/ Coordination Center of China. Mar. 2023 (last accessed). CNVD. https://www.cnvd.org.cn/.

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

[54] Andreas Pavlogiannis. 2020. Fast, sound, and effectively complete dynamic race prediction. *Proc. ACM Program. Lang.* 4, POPL (2020), 17:1–17:29. https://doi.org/10.1145/3371085

[55] Eli Pozniansky and Assaf Schuster. 2007. Multirace: Efficient On-the-fly Data Race Detection In Multithreaded C++ Programs. *ACM Trans. Comput. Syst.* 19, 3 (Nov. 2007), 327–340.

[56] Kun Qiu, Zheng Zheng, Kishor S. Trivedi, and Beibei Yin. 2020. Stress Testing With Influencing Factors to Accelerate Data Race Software Failures. *IEEE Transactions on Reliability* 69, 1 (2020), 3–21. https://doi.org/10.1109/TR.2019.2895052

[57] 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)*.

[58] 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.

[59] Malavika Samak, Omer Tripp, and Murali Krishna Ramanathan. 2016. Directed Synthesis of Failing Concurrent Executions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 430–446. https://doi.org/10.1145/2983990.2984040

[60] 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.

[61] Koushik Sen. 2008. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 11–21. https://doi.org/10.1145/1375581.1375584

[62] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, NY, USA) *(WBIA '09)*. 62–71.

[63] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic Race Detection with LLVM Compiler. In *Runtime Verification (RV 2011)*. 110–114.

[64] 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.

[65] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. 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, 112–122. https://doi.org/10.1145/1250734.1250748

[66] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2016. Concurrency Testing Using Controlled Schedulers: An Empirical Study. *ACM Trans. Parallel Comput.* 2, 4, Article 23 (feb 2016), 37 pages.

[67] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 369–384.

[68] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 70–82. https://doi.org/10.1145/504282.504288

[69] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-Guided Fuzzer for Discovering Use-after-Free Vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 999–1010. https://doi.org/10.1145/3377811.3380386

[70] Zan Wang, Dongdi Zhang, Shuang Liu, Jun Sun, and Yingquan Zhao. 2019. Adaptive Randomized Scheduling for Concurrency Bug Detection. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 124–133. https://doi.org/10.1109/ICECCS.2019.00021

[71] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled Concurrency Testing via Periodical Scheduling. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 474–486.

[72] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. Mar. 2023 (last accessed). PERIOD. https://github.com/wcventure/PERIOD.

[73] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo 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

[74] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. 2012. Concurrency Attacks. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism* (Berkeley, CA) *(HotPar'12)*. USENIX Association, USA, 15.

[75] 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 Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 280–291.

[76] 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.

[77] Michal Zalewski. Mar. 2023 (last accessed). American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[78] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng 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

[79] Shihao Zhu, Yuqi Guo, Long Zhang, and Yan Cai. 2023. Tolerate Control-Flow Changes for Sound Data Race Prediction. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1342–1354. https://doi.org/10.1109/ICSE48619.2023.00118