RaceBench: A Triggerable and Observable Concurrency Bug Benchmark

Anonymous Author(s)

***Abstract*—Concurrency bugs are one of the most harmful and hard-to-address issues in modern multithreaded software. Such bugs are hard to discover, reproduce, diagnose or fix in practice due to their non-deterministic nature. Although more and more bug discovery solutions are proposed in recent years, it is difficult to evaluate them with existing concurrency bug datasets. It is highly demanded to build a high-quality benchmark of concurrency bugs, to facilitate researchers developing advanced concurrency bug discovery or diagnosis solutions.**

In this paper, we present an automated bug injection solution to automatically inject representative concurrency bugs into real world multithreaded C/C++ programs, and present the first triggerable and observable concurrency bug benchmark RaceBench. We have conducted a large-scale empirical study on concurrency bugs, learned their patterns, and built a program state model to characterize them, which enables us to inject representative bugs. To make the bugs triggerable, we follow the dynamic execution traces of target programs and inject bugs at locations that are reachable from the program entry. To make the bugs observable, these bugs are followed by explicit security assertions, removing the requirement of sophisticated sanitizers to detect the existence of such bugs. We have built a benchmark with this approach consisting of 1500 bugs injected into 15 programs, and evaluated four concurrency bug discovery tools and one general bug discovery tool with it. Results showed that existing concurrency bug discovery solutions are still in the early stage, and our benchmark could shed light on the future direction of improvements.

Keywords*—*concurrency bugs, benchmark, bug addition

I. Introduction

Concurrent programs are prevalent due to the wide use of multi-core systems. However, it is not easy to write thread-safe concurrent programs because programmers have to deal with special constructs (e.g., threads and locks) and avoid synchronization issues (e.g., race conditions and deadlocks). Therefore, a large number of concurrency bugs are introduced[1] in modern multithreaded programs. Some of these bugs have even caused severe consequences, e.g., the DirtyCow [3] bug caused kernel privilege escalation which endangers the operating system and the Therac-25 [27] bug caused radiation overdoses which endangered human lives.

To mitigate the threat of concurrency bugs, many concurrency bug discovery techniques have been proposed, such as static-based race detectors [8], [32], [34], dynamic-based race detectors [31], [26], thread scheduling strategies [9], [37], and thread-aware fuzzing [10], [30], [20]. However, concurrency bugs have a non-deterministic nature, i.e., such bugs can only be triggered when certain inputs are given and the non-deterministic thread interleaving of the program satisfies certain constraints. Therefore, it is hard to reproduce or diagnose the concurrency bugs reported by these tools or compare these tools' performance. In practice, researchers turn to select some target programs and manually verify the reported bugs one by one, or manually prepare a set of ground-truth bugs [19], [14] for the tools to detect. Such manually assisted evaluations in general only cover a limited number of bug types and applications. It is unreliable to fairly compare different concurrency bug discovery techniques in this way, which hinders researchers from locating existing solutions' bottlenecks or finding potential directions for improvements.

To facilitate the research of discovering or diagnosing concurrency bugs, researchers have provided several concurrency bug datasets. The first type of datasets are manually collected from real-world applications [13], [38], [19]. Such datasets consist of *authentic* bugs, but the bug count is relatively small. They in general only include 1 or 2 bugs for each program and have less than 100 bugs in total, therefore are insufficient to precisely evaluate the performance of modern tools. The other type of datasets are automatically generated by certain bug injection solutions [17], [15]. Such datasets in general have good scalability and have a large number of bugs. But they may have bugs that are not triggerable (e.g., due to path constraints) or not observable (e.g., no abnormal behaviors are exhibited), causing problems in evaluating dynamic analysis and testing techniques.

Therefore, a high-quality benchmark consisting of a large number of concurrency bugs is demanded. In order to evaluate different types of concurrency bug discovery techniques, we argue that a high-quality benchmark should have three characteristics.

C1: Representative. In order to thoroughly evaluate different techniques, the benchmark should be representative of real-world concurrency issues. This does not mean that the synthetic bugs have to be representative in all aspects to imitate real ones. Instead, the synthetic bugs are expected to cover common bug types, root causes, and code patterns of real-world concurrency bugs. Further, the synthetic bugs should be located in real-world applications rather than self-crafted small testing programs, making the hosting application representative.

C2: Triggerable. The bugs in the benchmark should be triggerable, i.e., there are certain pairs of (inputs, thread interleavings) able to trigger the bugs. In other words,

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=race+condition

Network and Distributed System Security (NDSS) Symposium 2023
28 February - 4 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24xxx
www.ndss-symposium.org

the bug will be triggered when the specific inputs are given to the program and the specific thread interleaving is followed. Otherwise, the bugs are in fact false positives, not suitable for tool performance evaluation.

C3: Observable. The bugs in the benchmark should have observable outcomes, e.g., crashing or hanging, when they are successfully triggered. Otherwise, such bugs (e.g., ones that yield wrong outputs rather than crash or hang) are not detectable by a wide range of tools (e.g., fuzzing-based solutions without proper sanitizers). Developing concurrency-sensitive sanitizers is a topic orthogonal to concurrency bug discovery. To support the evaluation of more tools without extra effort, we argue that the benchmark should only consist of bugs with observable outcomes.

In this work, we propose a novel bug injection solution RaceBench, able to automatically inject representative, triggerable and observable concurrency bugs into existing real-world multithreaded C/C++ programs and build high-quality benchmarks of concurrency bugs.

First, *to make the bugs representative*, we conducted a large-scale empirical study on existing concurrency bugs, learned their causes and types, and formally describe the common code patterns of the majority of them with a general program state modeling language. Using these code patterns as templates, RaceBench could generate numerous concurrency bugs on demand. Such generated bugs are injected into real-world applications and share the same patterns with existing bugs, and thus are representative.

Second, *to make the bugs triggerable*, we adopt a trace-based approach and only inject concurrency bugs at proper locations in feasible program paths. Specifically, RaceBench first utilizes dynamic testing to get one feasible execution trace, then locates pairs of spots in the trace that can be concurrently executed by two threads, and finally injects concurrency bugs at such pairs of spots. Therefore, if we run the new program with the same inputs and thread execution orders as the trace, and provide a proper thread interleaving related to the bug injection spots, then the injected bugs will be triggered.

Lastly, *to make the bugs observable*, RaceBench further injects security assertions right after the injected triggerable bugs, which will crash and report bug contexts deliberately. The program will crash once the injected bugs are triggered, and thus can be observed by tools without extra effort. In the future, if developers would like to develop and assess concurrency-related sanitizers, we could simply disable these injected assertions.

Based on the proposed approach, we have built a concurrency bug benchmark RaceBench, consisting of 15 real-world programs, each of which is injected with 100 concurrency bugs. The dataset covers a wide range of concurrent applications, and the majority of bug patterns and bug difficulties. Note that, the number of bugs, patterns of bugs, and the difficulties of bugs are all configurable or extendable. The benchmark also provides scripts and inputs that can be used to reproduce and verify bugs.

With the benchmark, we further evaluated four open-source concurrency bug discovery tools, i.e., CONAFL, OPENRACE, TSAN, MAPLE, and one general bug discovery tool AFL++. These tools adopt different techniques and have different performances on bugs of different patterns and complexity. Results showed that, among the four tools targeting concurrency bugs, only TSAN finds more concurrency bugs than the popular thread-unaware fuzzer AFL++. It indicates that there is a lot of room for improvements in concurrency bug discovery techniques with respect to improving reproducibility, reducing runtime overhead, using better thread scheduling strategies, and considering thread interleaving space together with the input space. It is still a big challenge to combine existing techniques to find concurrency bugs effectively.

In this paper, we make the following contributions:

- We present an automated bug injection solution RaceBench, which is able to automatically inject representative, triggerable and observable concurrency bugs.
- We conducted a large-scale empirical study on concurrency bugs, learned their patterns, and built a program state model to characterize the majority of them.
- We constructed a benchmark consisting of a large number of concurrency bugs with the proposed bug injection solution.
- We evaluated several bug discovery solutions with the benchmark and showed their limitations, while shedding lights on the future direction of improvements.
- We open source RaceBench[2], to facilitate the community on the research of concurrency bugs.

II. EMPIRICAL STUDY

To make the injected bugs *representative*, we need to find out the types of concurrency bugs that commonly appear in practice, root causes of the concurrency bugs, and the corresponding patterns. Therefore, we conducted a large scale empirical study on historic concurrency bugs in real-world software.

A. Methodology

We collected real bugs from the CVE list [1] and existing datasets for study. First, we collected all 280 bugs in the CVE list dating from 2020 to 2022 matching the keywords: race, concurrent, lock, atomic, sync, thread, or wait. Then we manually checked their description to verify whether they are indeed concurrency bugs, and removed 116 non-concurrent bugs and 72 bugs without technical details, leaving 92 bugs. Second, we also collected bugs from two existing datasets RADbench [13] and the one [36] labeled by Yu and Narayanasamy. Apart from bugs that are not confirmed by developers, we obtained another 30 bugs from these two datasets. In total, 122 concurrency bugs are included in this paper for further concurrency bugs analysis. These bugs mainly reside in open source projects of various kinds, including large-size projects such as Firefox, Android, Linux kernel, and MySQL, as well as some small-size or medium-size projects.

We then analyzed these 122 bugs from two perspectives: bug causes and bug types. For bug causes, we investigate why programmers would introduce concurrency bugs, based on the discussions on the software issue tracking platforms (e.g.,

[2] https://github.com/rb130/RaceBench

GitHub, Bugzilla) and the commit messages of the bug fixes. For bug types, we study taxonomy from previous works [23], [19], [12], [5] and classify bugs according to their technical mechanisms. Then we characterize the common code patterns of such bugs and formally describe them with a general program state modeling language.

B. Concurrency Bug Categorization

By analyzing the collected bugs, we group the bugs into different root causes and types.

1) Bug Root Causes: There are three major root causes of concurrency bugs, as follows.

R1: Wrong Program State Assumption. Developers may assume the program executes sequentially, and the program state (e.g., the value of one variable) does not change from the last write operation till the current site. This assumption holds in most cases, but not all cases. For instance, in a multi-threaded program, the value of a variable in one thread could be modified by code in another thread. In such cases, the program state may change in a way unexpected by developers, which further causes bugs.

R2: API Misuse. When a programmer does not fully understand the behavior or side effects of an API, he may use it in a wrong way or in a wrong scenario. For example, the API `pthread_cond_wait` can be used to block the current thread until it receives a notification from a condition variable; however, the programmer may not know that spurious wakeups can occur, which will continue the execution with an unmet condition. Besides, when a thread-unsafe library is used by a multi-threaded program, the programmer may forget to wrap the API with locks at some point.

R3: Lock Dependency Issues. Most deadlock bugs are caused by the failure of lock dependency management. When an action of a thread involves multiple resources and their corresponding locks, the programmer needs to be very careful about the lock relations. Acquiring locks in an incorrect order can result in two or more threads circularly waiting for each other. As an example to show the significance of this cause, the Linux kernel uses a runtime validator `lockdep` to detect lock dependency issues[3].

R4: Other. There are a small number of concurrency bugs caused by other reasons, such as algorithmic errors and design issues.

2) Bug Types: There are three major types of concurrency bugs in terms of the the affections to the application:

P1: Atomicity Violation. In concurrent programs, a thread often needs to perform some operations atomically. While executing these operations, the results should never be interfered by other threads. Atomicity violation happens when a group of operations that should be atomic are not enforced by thread synchronization primitives (such as atomic memory operations and locks). More details will be presented in §II-C2.

P2: Order Violation. Some operations in a thread have to execute after certain operations in another thread. For example, a variable can only be accessed after initialization in another thread, and a heap object is allowed to be released only if it is no longer used. If operations from different threads are not executed in the correct order, the program will enter an invalid state. More details will be presented in §II-C3.

[3]https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt

TABLE I: Statistics of root causes and types of bugs.

	R1: Wrong Program State Assumption	R2: API Misuse	R3: Lock Dependency Issue	R4: Other
P1: Atomicity Violation	**66**	3	0	1
P2: Order Violation	**23**	2	0	0
P3: Deadlock	2	1	7	1
P4: Miscellaneous	0	20	0	3

Note: The sum of data in the table is greater than the number of bugs (i.e., 122), because some complex bugs have multiple causes.

P3: Deadlocks. Deadlock is a situation where several threads circularly wait for resources held by each other. Apart from resource contention, threads circularly waiting for messages can also result in a communication deadlock.

P4: Miscellaneous. There are some other types of concurrency bugs, such as thread starvation and livelock. More details of these types of bugs could be found in studies [5], [6].

Note that, a common type of concurrency bugs – data race – is not listed in this classification. From the definition, we know that a data race occurs when two or more threads access the same memory location *at the same time* and at least one access is writing. In fact, data races are further divided into atomicity and order violations in the aforementioned classification.

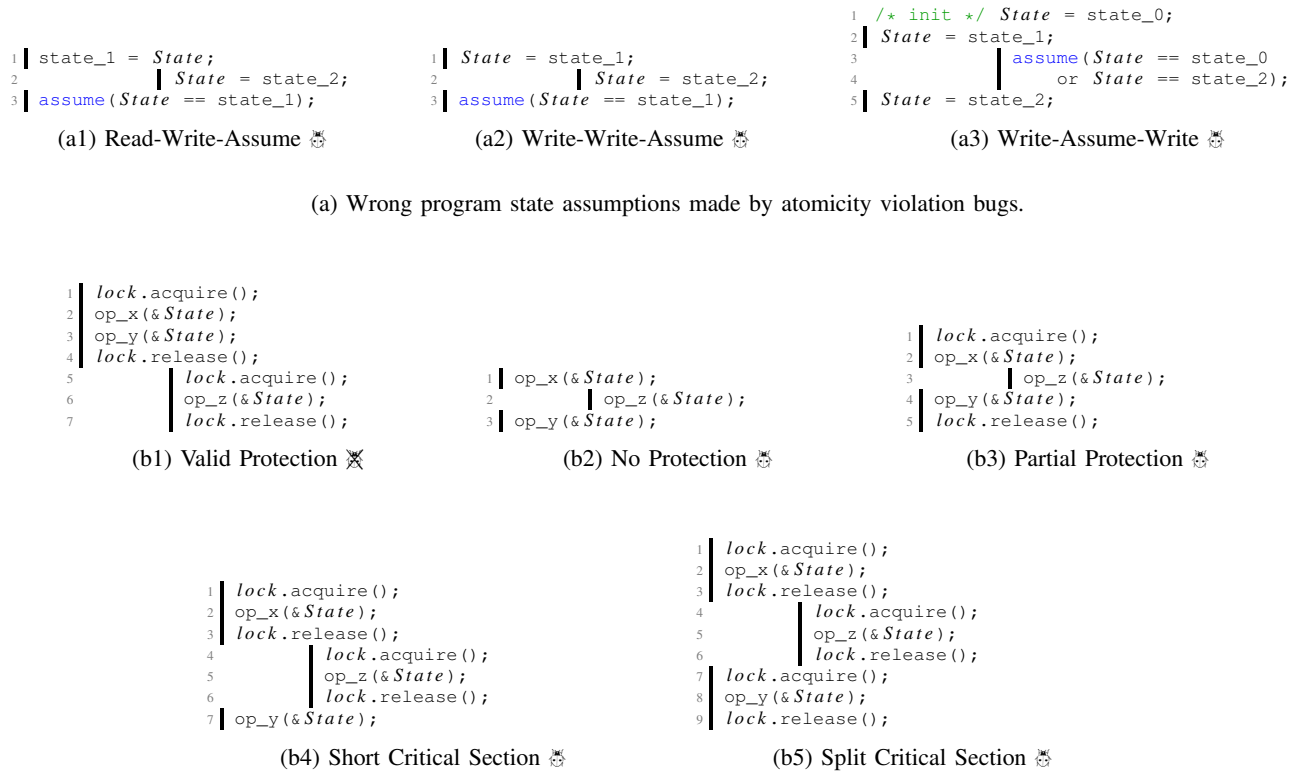
3) Statistics and Justification: Table I shows the statistics of root causes and types of all 122 bugs we studied. We can see that wrong program state assumption is the major cause of concurrency bugs. Most bugs caused by wrong assumptions fall into the bug types of atomicity or order violation, while most atomicity or order violation bugs result from wrong assumptions.

Based on this observation, we focus on building a benchmark of concurrency bugs that are caused by wrong program state assumptions and fall into atomicity or order violation. Such bugs cover about 69% of bugs we studied, which we think is a large proportion of concurrency bugs in practice.

C. Bug Modeling

In this section, we introduce the code patterns of bugs that we will cover in the benchmark, to facilitate further automated bug injection. Specifically, we will describe the common patterns of atomic violation and order violation bugs caused by wrong program state assumptions. First of all, we propose a program state modeling language to formally describe them. Then, we detail the description of each code pattern with this model, which can be used as templates to automatically generate concurrency bugs.

1) Program State Modeling Language: Before digging into the detailed code patterns, we first introduce a general language. Here, we denote the program state as the aggregation



(a1) Read-Write-Assume (a2) Write-Write-Assume (a3) Write-Assume-Write

(a) Wrong program state assumptions made by atomicity violation bugs.

(b1) Valid Protection (b2) No Protection (b3) Partial Protection

(b4) Short Critical Section (b5) Split Critical Section

(b) Valid and invalid critical section protections against atomicity violations.

Fig. 1: Code patterns of atomicity violation bugs.

of all variables shared between threads in a program, using the terminology $State$. Each shared variable is a member of $State$, for example, $State.var_0$. An assignment in the form of $State = S_0$ means transforming to a new program state S_0. An assignment in the form of $State.var_0 = x$ means transforming to a new program state where only $State.var_0$ is changed to x. Assumptions made by developers can be represented by statements like $\texttt{assume}(cond(State.var_0, State.var_1))$, which means the program state is assumed to meet a certain condition defined by the boolean function $cond$.

When using this language to describe code patterns in the following sections, we put Thread 1 on the left side, Thread 2 on the right side, and the initial states on the top. We mark bug and bug-free code patterns with icons and respectively.

2) Atomicity Violation: The code patterns of atomicity violation bugs can be characterized from two dimensions: what assumptions are made, and how critical sections are used, as shown in Fig. 1. Note that these two dimensions are orthogonal.

a) Dimension 1: Wrong Assumptions: Atomicity violations are often caused by three types of wrong assumptions, as shown in Fig. 1a.

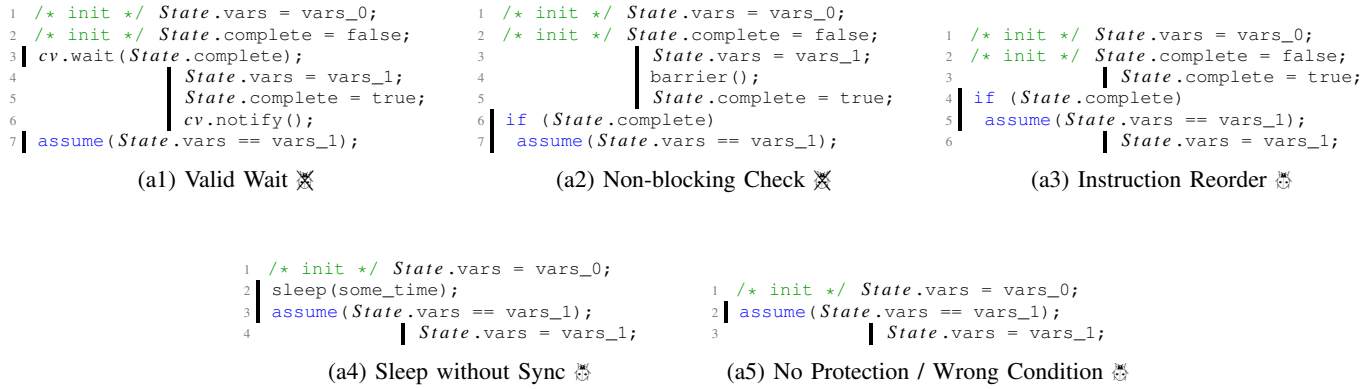
Read-Write-Assume (RWA). In Fig. 1a1, Thread 1 reads from the program state and saves it into a temporary variable $state_1$. Then shortly it assumes the current program state is the same as $state_1$, since no changes are made in this thread. However, it does not expect that Thread 2 (on the right) could change the program state in between the read site and the assumption site.

Write-Write-Assume (WWA). In Fig. 1a2, Thread 1 changes the program state to a new state $state_1$, and then shortly it assumes the program state is still $state_1$. However, Thread 2 could change the program state in between the write and the assumption sites.

Write-Assume-Write (WAW). In Fig. 1a3, Thread 1 intends to perform an action that requires two continuous write operations to the program state. Thread 2 assumes that neither or both operations have been executed. If Thread 2 gets executed when only one operation completes, it will get an invalid intermediate state.

b) Dimension 2: Invalid critical section protections: Atomicity is often enforced by mutual exclusion locks. A pair of lock acquiring and releasing statements creates a critical section that no more than one thread can enter. Suppose that Thread 1 wants to guarantee the atomicity of two operations op_x and op_y and stops the operation op_z of Thread 2 from breaking it. We show a valid protection and the common code patterns of invalid protections in Fig. 1b.

Valid Protection. As shown in Fig. 1b1, a valid protection



(a1) Valid Wait ✗ (a2) Non-blocking Check ✗ (a3) Instruction Reorder ☹

(a4) Sleep without Sync ☹ (a5) No Protection / Wrong Condition ☹

Fig. 2: Code patterns of order violation bugs.

should put two operations op_x and op_y in one critical section and put op_z in another, so that op_z can only be executed before or after them rather than in between them.

No Protection (NoLock). In Fig. 1b2, the code is not protected at all, so threads can interleave arbitrarily. Atomicity is violated if the instructions are executed in the shown order.

Partial Protection. In Fig. 1b3, only one thread is protected with a lock. As the other thread is unaware of the lock, it runs regardless of the critical section.

Short Critical Section. In Fig. 1b4, both threads try to protect the operations but at least one important operation is still outside critical sections. The critical section in Thread 1 is not large enough, giving Thread 2 a chance to interfere.

Split Critical Section. In Fig. 1b5, the two operations in Thread 1 are supposed to be atomic as a whole, but they are put into separated critical sections, so Thread 2 can still interfere.

3) Order Violation: As shown in Fig. 2, there are three common code patterns of order violations.

Valid Wait. In Fig. 2a1, the correct order is ensured by waiting. Thread 1 blocks until $State.complete$ becomes true, so that the assignment to $State.vars$ in Thread 2 can be executed before the assumption. After setting $State.complete$, Thread 2 uses the `notify` primitive to wake up Thread 1.

Non-blocking Check. In Fig. 2a2, the correct order is achieved in a non-blocking way. In practice, people may care about the performance overhead of waiting, so they check the condition without blocking the thread.

Instruction Reorder. Note that instructions may get reordered by either the compiler or the processor, for performance reasons. As shown in Fig. 2a3, even if the condition is set after the variable assignment in Thread 2, it may get executed earlier due to instruction reordering. This can be resolved by adding a memory barrier before setting $State.complete$, as shown in Fig. 2a2.

Sleep without Sync. In Fig. 2a4, the programmer falsely uses `sleep` instead of `wait` to synchronize the order, as shown in Fig. 2a4. However, when the computer is busy, the operating system can leave Thread 2 unscheduled for a long time. So the assumed condition can still be unsatisfied even after `sleep` finishes.

No Protection / Wrong Condition (NoWait). In Fig. 2a5, Thread 1 does not check the order condition at all, or checks a wrong condition, which is equivalent to no protection. In practice, the assumption in Thread 1 is often placed after some time-consuming operations, or in a rarely used function. So the incorrect execution order is unlikely to trigger, leaving a hidden bug.

TABLE II: Statistics of the code patterns for bugs we studied.

(a) Atomicity Violation

	RWA	WWA	WAW	*
NoLock	11	4	1	3
Partial	6	0	2	3
Short	4	0	2	1
Split	8	9	3	0
*	5	1	0	3

(b) Order Violation

Reorder	1
Sleep	1
NoWait	17
*	4

⋆ means the code pattern is unclear.

4) Statistics: Table II shows the statistics of the code patterns for bugs that we focus on, i.e., atomicity and order violations caused by wrong program state assumptions. We can see that the code patterns we summarized cover the majority of the bugs we studied. Further, we will build a concurrency bug benchmark with the code patterns presented in this section.

III. Automated Bug Injection

In this section, we illustrate the design of RaceBench, an automatic concurrency bug injection tool.

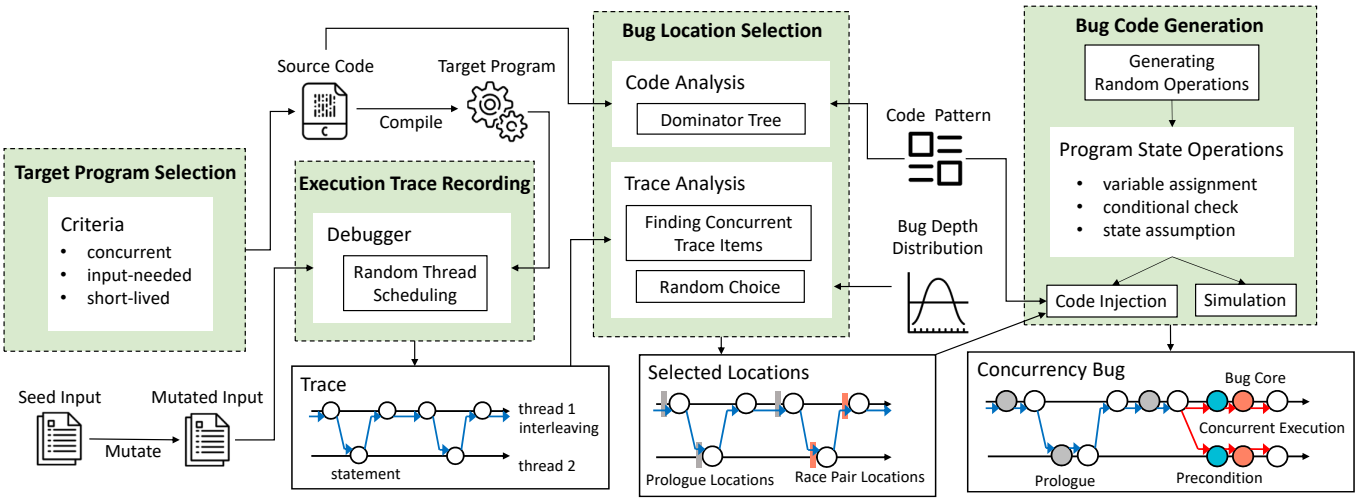


Fig. 3: Overview of the trace-based concurrency bug injection approach taken by RaceBench.

A. Overview

An intuitive idea to add triggerable concurrency bugs is injecting buggy code to a pair of locations that are simultaneously executed by two threads. RaceBench finds such locations from a dynamic execution trace of the program, so they are always reachable as long as the trace is followed. Similar to real bugs, a bug in RaceBench has three conceptual parts: the *prologue*, the *precondition*, and the *bug core*. The prologue prepares a program state to be used later. The precondition checks whether the program state is the expected one. If the precondition is met, the program enters the bug core, which matches one of the previously described code patterns introduced by wrong program state assumptions.

Fig. 3 shows the overall process of automatically adding one bug. Since not all programs are suitable for injecting concurrency bugs, we first conduct *target program selection* to find the suitable programs. Given the source code of the selected program, RaceBench uses a trace-based approach to inject concurrency bugs. In the step of *execution trace recording*, RaceBench runs the program under a debugger, feeds it with an input, and records the execution trace. In the next step *bug location selection*, RaceBench analyzes the code to select a series of locations from the trace, including the prologue code instrumentation locations, and the race pair locations that can be concurrently executed by two threads. Then in the *bug code generation* phase, RaceBench randomly generates program state operations, which are simulated and injected to previously selected locations, forming a concurrency bug.

To trigger the bug, one has to find the correct thread interleaving to pass the precondition and break the wrong program state assumption in the bug core. A specific input may also be required, depending on whether the input is used in the required program state. Note that the thread interleaving that triggers the injected bug is not necessarily the same as the one recorded in the trace. When a bug is triggered, RaceBench will abort the program to make it observable. The bug ID will also be recorded so that the users can easily know which bug is discovered.

```
State.var_0 = 0;
State.var_1 = 0;
State.var_2 = 0;
input = input_bytes();
```

```
int i, x = 0;

for (i = 0; i < 3; ++i) {
  State.var_1 += State.var_0;
  x = x ^ arr[i];
}
if (State.var_1 == 9)
  State.var_2 = State.var_0 + 1;
printf("thread 1\n");

if (x == 0) {
  if (State.var_1 == 9)
    assume(State.var_2 ==
           State.var_0 + 1);
  printf("x=0\n");
}
```

```
State.var_0 = input[5];
printf("thread 2\n");

if (State.var_1 == 9)
  State.var_2 = 20;
some_work();
```

The injected code are in colored boxes: the prologue in gray, the precondition in cyan, and the bug core in red.

Fig. 4: A concurrency bug injected by RaceBench.

Fig. 4 demonstrates an example of a synthetic atomicity violation bug of the `Write-Write-Assume` code pattern. It will be used as the running example throughout this section. We will illustrate how a concurrency bug is injected step by step as follows.

B. Target Program Selection

We select programs that are suitable for the benchmark based on the following requirements:

Concurrent: The program should have multiple threads, i.e., the program can be executed concurrently.

Input-needed: The program should read some inputs. Although a concurrency bug may not require any input,

TABLE III: An execution trace of the example in Fig. 4.

Item ID	Thread ID	Location	New Race Pair (Item IDs)	Program State*
1	1	line 1	n/a	[0, 0, 0]
2	2	line 3	(2,1)	[3, 0, 0]
3	1	line 4	(3,2)	[3, 0, 0]
4	1	line 6	(4,2)	[3, 3, 0]
5	1	line 6	(5,2)	[3, 6, 0]
6	1	line 6	(6,2)	[3, 9, 0]
7	1	line 10	(7,2)	[3, 9, 4]
8	2	line 13	(8,7)	[3, 9, 20]
9	1	line 14	(9,8)	[3, 9, 20]
10	1	line 18	(10,8)	[3, 9, 20]

* The program state is represented as a list of values of variables in $State$. Suppose that the example bug code uses an input file where `input[5]` is 3.

there are also concurrency bugs that only appear under certain ill-formed inputs. This enables the bug discovery tools (e.g., fuzzers) to test the program by exploring the input space if they want.

Short-lived: The program should finish in a short time, i.e., the program is not a daemon and does not contain any endless loops. Many dynamic bug discovery tools (e.g., Muzz [10] and Maple [37]) rely on testing the program multiple times and exploring different thread interleavings and inputs. Long-running programs do not fit them.

We select the baseline programs mainly from PARSEC [7] and SPLASH-2 [35] datasets to build the benchmark. More details of these programs are available in §IV-A. We consider each selected program as a *target program* for further concurrency bug injection.

C. Execution Trace Recording

To ensure that the generated bugs are triggerable, RaceBench records an execution trace and makes it the base to inject buggy code. The trace records a real thread interleaving of the target program. By following the trace, we can make sure that the program can reach and execute the injected code in order.

RaceBench records a trace of the target program by running it under a debugger. The scheduler of the operating system is locked by the debugger, so only a single thread of the program is able to run at the same time. A trace can be obtained by taking the following steps: ① Randomly select one of the threads waiting to be executed; ② Let the selected thread execute a single statement and stops at the next one; ③ Run the above steps repeatedly until the program is terminated or exceeds a time limit. It is obvious that any potential thread interleaving could be picked by this approach.

During tracing, the input used by the target program is not the seed input but a randomly mutated one. The mutated input used in tracing will become an answer for bug discovery tools to find out (if they need to). Notice that it will not be beneficial for bug discovery tools to use the same scheduling or mutation strategy as ours. We obtain the random seed from the operating system dynamically, and a different seed will lead to a different and irrelevant synthetic bug.

A trace is denoted as an ordered list of items, each of which represents one statement to be executed. Each item is denoted as a tuple `<tid, loc>`, where `tid` is the unique ID of the running thread and `loc` is the location (file name and line number) of the source code being executed. Table III shows an example execution trace of the aforementioned running example. We omit the file names in `loc` for simplicity as we only have one file in this case.

D. Bug Location Selection

A concurrency bug exists only if the vulnerable code can run in different orders, and only triggers in certain special orders. Thus, we choose to add the bug core to a pair of locations (called a race pair) that can be executed by two threads concurrently. When two threads reach the locations at the same time, either part of the code can run first, so the outcome will depend on the execution order.

Then the question is how to find such race pair locations. The clues are in the execution trace. For each item in the trace, RaceBench looks for previous items that have a different thread ID. Such items are able to execute concurrently, and thus suitable for injecting bug core. The preconditions are also injected before the bug core. Table III shows the newly identified race pairs for each trace item.

Some bug cores can make use of multiple race pairs. If there are two operations (op_x and op_y) to be performed in one thread, both op_x and op_y should be able to run concurrently with an operation op_z in the other thread, but op_x and op_y do not have to be adjacent. In real bugs, there can be other code in the middle of the two operations. To imitate it, we can find a neighboring race pair that shares a common trace item with the current race pair, so that op_z can be placed in the common trace item. For example, in Table III, $(8, 7)$ and $(10, 8)$ are neighboring race pairs because they share an item 8, and op_x, op_z, and op_y can be placed at 7, 8, 10 respectively.

However, simply finding a second race pair in this way may introduce unexpected solutions to the injected bug. Although the second operation op_y is after the first one op_x in the current trace, it may not be so in other possible traces. For example, if the program in Fig. 4 is able to reach lines 15–17 without executing lines 8–9 (via other paths), the assumption can be broken no matter the thread interleaving. To guarantee the injected bug is a concurrent bug, RaceBench only picks locations having domination relationship [29] as candidates to inject dependent operations.

Specifically, based on the characteristics of different code patterns, as shown in Fig. 1 and Fig. 2, we find different code patterns require different domination dependency.

- None: The code does not have two operations to be performed in one thread. Bugs of the code pattern NoWait could use this domination.
- Pre-Dom: The former operation op_x should dominate the latter operation op_y. Bugs of the code pattern RWA, WWA, and Sleep could use this domination.
- Post-Dom: The latter operation op_y should post-dominate the former operation op_x. Bugs of the code pattern WAW and Reorder could use this domination.

Then, we analyze the dominator tree of target programs with the LLVM [16] compiler, and select race pairs of different domination relationships and inject bugs of corresponding code patterns.

Further, RaceBench randomly selects multiple locations before the race pairs, to insert the prologue code (of multiple code snippets) that initializes the program state used by the precondition and the bug core. Apart from initialization, the prologue is also responsible for adjusting the difficulty of the bug. Because there are multiple prologue locations and each is selected from any thread, the more prologue locations we select (and split the prologue code to each location), the more complex the expected thread interleaving that accomplishes the prologue setting becomes. Table III marks these locations with a gray frame.

E. Bug Code Generation

RaceBench can generate atomicity violation and order violation bugs caused by wrong program state assumptions. The code patterns described in §II-C will be used to produce the bug cores, which are representative of the core defective code in real bugs. To integrate the bug core into the target program, RaceBench also adds peripheral code that prepares the program states and forms preconditions, which helps to grasp constraints in the original control flow of the program.

To produce a concurrency bug, there are three types of operations to be added to the source code.

Variable Assignment. Transform the program state by assigning the shared variables.

Conditional check. Check if the current program state is the expected one.

State Assumption. This is a more critical conditional check. Once an assumption fails, RaceBench recognizes that a bug has been triggered.

These operations are the building blocks for the prologue, the precondition, and the bug core. The prologue is implemented as a series of assignments. The precondition is a set of conditional checks placed in the front of each code piece of the bug core. The bug core is generated by substituting the variables and statements in the code patterns with concrete ones, which can involve all three kinds of operations.

The values of the assignments are computed from operations, including addition, subtraction, xor, etc. The operands can be immediate numbers, input data, or variables in the current program state. If the input data is used, it indicates that the bug is only triggerable under certain inputs. A variable can be assigned by different threads for multiple times, so its value becomes interleaving dependent.

Although we do not use the original variables in the target program (because we cannot control their values), RaceBench uses the data in the target program implicitly. As the bug codes are injected along the trace, they inherently keep the path constraints in the target program. The example bug in Fig. 4 is closely related to the original control flow of the program. A part of the bug code is placed under an `if` condition that specifies a constraint on the xor-sum of array `arr`'s elements.

The conditional check takes one variable from the program state and checks whether it is the expected value. It is not necessary to test many variables in one check, because we can always use assignments beforehand to blend the values of different variables.

```
1 /*init*/ State.var_1 = 0;
2 State.var_1 = 1;
3     // assume
4     State.var_2 =
5       State.var_1 == 0 or
6       State.var_1 == 2;
7 State.var_1 = 2;
8 next(State.var_2);
9     next(State.var_2);
```

(a) Chaining

```
1 /*init*/ State.var_1 = 0;
2 State.var_1 = 1;
3     // assume
4     State.var_2 =
5       State.var_1 == 0 or
6       State.var_1 == 2;
7     if (State.var_2)
8       next();
9 next();
10 State.var_1 = 2;
```

(b) Nesting

Fig. 5: Examples of combined assumptions.

The assumptions are realized in two ways. The trivial way is to call a `bug_trigger` function if the assumption is broken. The argument of the function is a bug ID, indicating which bug has been triggered. Inside `bug_trigger`, RaceBench records the bug ID, and then aborts or continues the program according to configurations. Another type of assumption is used to implement combined assumptions (we will explain it later). The falsely assumed program state can be saved into a program state variable to be used later.

Given the generated code, RaceBench simulates them to get the expected program state for each trace item. The expected program state is used to fill the conditional checks with correct values. During simulation, RaceBench walks along the trace, maintains the current program state, and runs the generated operations at the current location to compute the next state. We should not inject and simulate the code at the same time, because the code locations can also appear earlier in the trace. Therefore, RaceBench walks through the trace for two passes: in the first pass, it decides what code to generate; and in the second pass, it performs the simulation. The simulated program states of the example code are listed in Table III.

Sometimes a concurrency bug is the result of multiple assumptions in the code. The core of such complex bugs can be regarded as the combination of multiple code patterns. During the empirical study, we find two types of combination: chaining and nesting. When chaining two assumptions, the second one is breakable only after that the first one has been broken (outside its race window[4]). Nesting has a stricter condition that the second assumption is available only inside the race window of the first one. When entering the race window, a thread gives chances to access the wrong program state, and when the race window is closed, the chances disappear. RaceBench can generate combined assumptions by saving the assumed condition in a variable and using it to affect the next assumption, as shown in the example in Fig. 5.

IV. Evaluation

We utilized RaceBench to construct a concurrency bug benchmark, and used it to evaluate the performance of exist-

[4] A race window is a small period in which the instructions of two threads are racing.

TABLE IV: Target programs in RaceBench.

Name	From	#Lines	#Threads	Category
blackscholes	PARSEC	512	4	Finance
bodytrack	PARSEC	9,728	5	Body Tracking
canneal	PARSEC	4,538	4	Chip Design
cholesky	SPLASH-2	5,239	4	Matrix Computation
dedup	PARSEC	5,210	12	Compression
ferret	PARSEC	15,745	7	Similarity Search
fluidanimate	PARSEC	5,712	5	Animation
pigz	Internet	8,406	6	Compression
raytrace	PARSEC	29,031	5	Raytracing
raytrace2	SPLASH-2	11,491	4	Raytracing
streamcluster	PARSEC	2,531	9	Clusterization
volrend	SPLASH-2	18,112	4	Volume Rendering
water_nsquared	SPLASH-2	2,053	4	Fluid Simulation
water_spatial	SPLASH-2	2,680	4	Fluid Simulation
x264	PARSEC	35,535	6	Video Encoding

TABLE V: Bug injection time and runtime overhead.

Program	Average bug injection time (s)	Running time ($*10^{-3}$s)		
		without bugs	with bugs	overhead
blackscholes	9.96	0.83	1.21	44.7%
bodytrack	60.47	26.66	27.89	4.6%
canneal	44.73	1.30	2.09	60.5%
cholesky	68.08	7.98	17.58	120.3%
dedup	24.46	14.82	21.30	43.7%
ferret	25.91	4.41	6.26	42.0%
fluidanimate	35.20	14.67	37.76	157.4%
pigz	24.42	0.56	0.82	47.6%
raytrace	76.05	3.76	6.32	67.8%
raytrace2	71.06	23.67	89.49	278.1%
streamcluster	121.93	36.88	41.62	12.9%
volrend	20.18	3.63	14.54	300.7%
water_nsquare	31.68	31.52	40.43	28.3%
water_spatial	26.20	26.92	33.23	23.4%
x264	31.91	3.42	16.28	376.3%

ing concurrency bug discovery tools. The following research questions will be discussed:

RQ1: Efficiency of RaceBench. How long does it take to inject bugs? What is the overhead of the additional code?
RQ2: Bug depth. How are the bug locations distributed over the execution trace? How many interleavings are needed to trigger the injected bugs?
RQ3: Performace of tools. How well do existing tools work on the benchmark generated by RaceBench?

A. Building Benchmark

We build a benchmark RaceBench to be used in the evaluation. We collect programs mainly from the PARSEC [7] (version 3.0) and SPLASH-2 [35] datasets. These datasets contain a variety of multithreaded programs originally used to measure the performance of multicore machines. After filtering out the programs according to §III-B, we finally select 15 target programs and turn them into a concurrency bug benchmark. More details are in Table IV.

We add 100 concurrency bugs to each target program. Notice that our approach does not have limitations on the number of bugs. While injecting bugs, we save the proof-of-concept input files and thread interleavings, so the users can verify that the bugs are triggerable.

Because the programs contain a large number of bugs, the users may want to choose a subset of bugs to evaluate their tools. Therefore, we set preprocessor macros to control which bugs to enable. When a bug is triggered, the behavior of RaceBench can also be configured via macros. RaceBench can either abort the program immediately or continue running to try to trigger other bugs.

RaceBench also provides a reproducer script. Given an execution trace as described in §III-C, the script runs the target program and schedules the threads as specified by the trace, until it reaches a bug. It can be used to verify bugs found by bug discovery tools. We highly encourage such tools to record the scheduling order if possible. Although such data is not a necessity to use the benchmark, they are very helpful for programmers to reproduce the bugs.

B. Efficiency of RaceBench

We assess the average time cost of injecting one concurrency bug. The results are listed in Table V. The majority of time is spent on recording the execution trace, so the cost is mostly determined by the size and the computational complexity of the target program. As we record a different trace for every single bug, the total time to build a benchmark is proportional to the number of bugs to add. On average, our system can inject a bug within one minute, efficient enough for the purpose of building benchmarks.

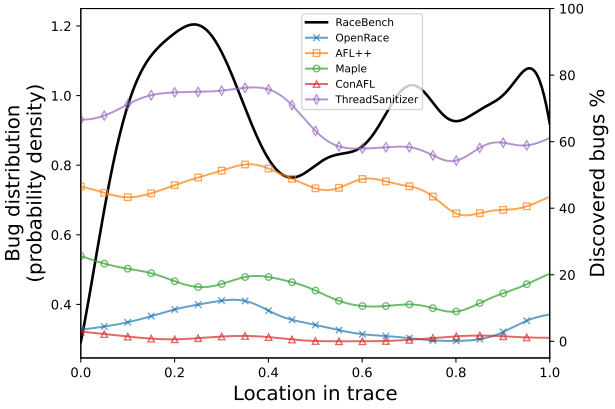
To evaluate the runtime overhead of the bugs, we gather the execution time of the target programs before and after injecting bugs. In Table V, we can see that different target programs have quite different overheads, ranging from 4.6% to 376.3%. This is because the overhead depends on the injection locations. If the bug code is injected into a busy loop, the code will be executed many times. Target programs with a lot of loops are more likely to have a large overhead. It also depends on the code pattern. If the code uses time-consuming statements, such as `lock` and `sleep`, it will have a larger runtime overhead.

C. Bug Depth

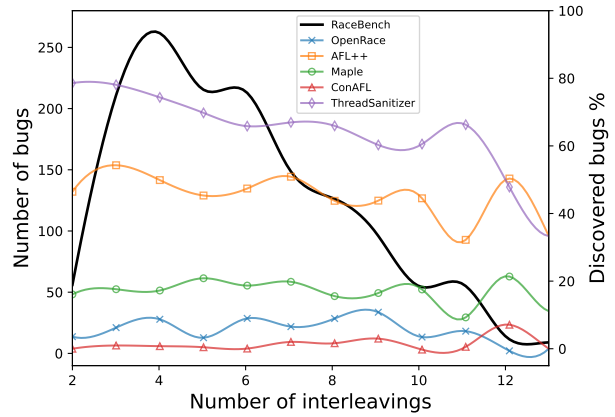
The bug depth is a measurement of the difficulty of a bug. We evaluate it from two aspects: trace depth and interleaving depth.

The idea of trace depth originates from the LAVA [11] bug dataset. It is measured by the proportion of instructions in the trace to be executed to reach the bug locations. If the proportion is small, the bug is placed near the entrance of the program, so it is easily reachable. If the proportion is large, to find the bug, tools have to correctly diagnose a lot of instructions, so it is likely to be hard. Fig. 6a shows the distribution of bugs over different trace depths in our dataset. Users can control the trace depth by tweaking RaceBench's location selection strategy. The distribution is also affected by the internal structures of the target programs. For example, the program may not create many threads when it just starts, so we have fewer bugs injected near the entrance.

Interleaving depth is the number of thread interleavings needed to trigger the bug [9]. It is an important indicator of concurrency bug difficulty, as the more interleavings a bug has,



(a) Trace depth



(b) Interleaving depth

Fig. 6: Bug depth (difficulty) and the percentage of bugs found by different bug discovery tools.

the more difficult it will be. In RaceBench, the interleaving depth is determined by the number of threads selected in the prologue and the bug core, so it is also controllable. As shown in Fig. 6b, our dataset covers a wide range of interleaving depths, which we consider to be adaptive for bug discovery tools of different capabilities.

D. Performance of Existing Tools

We evaluate five automatic bug discovery tools on RaceBench.

- OpenRace [32], a static program analysis framework aiming to detect data races.
- AFL++ [24], a widely used fuzzer that combines state-of-the-art fuzzing techniques. It does not have special treatments for concurrency bugs.
- ConAFL [20], a thread-aware fuzzer that controls the thread scheduling of the target programs.
- TSan (ThreadSanitizer) [31], a compiler instrumentation module that detects data races during run-time.
- Maple [37], a concurrent program testing tool that dynamically explores thread interleavings with race pair coverage heuristics.

TABLE VI: Number of bugs discovered by tools.

	OpenRace	AFL++	ConAFL	TSan	Maple
blackscholes	1	32	1	25	12
bodytrack	0	51	0	70	5
canneal	5	51	0	57	25
cholesky	Error	46	6	76	20
dedup	84	63	0	72	20
ferret	0	60	0	84	19
fluidanimate	10	51	4	85	19
pigz	Error	86	0	87	25
raytrace	0	54	0	60	19
raytrace2	Error	55	0	44	23
streamcluster	5	Error	Error	86	17
volrend	Error	28	0	98	13
water_nsquare	Error	17	0	27	4
water_spatial	Error	49	3	93	21
x264	Error	61	2	74	21
total	105	704	16	1038	263

We mark some cells as *Error* if the bug discovery tool crashes itself.

The experiments are conducted on Intel Core i7-11700 16-thread CPUs. We run the above tools referring to their respective usages. For fuzzers (AFL++ and ConAFL), we fuzz each target program for 24 hours. Instead of relying on their reports, we count the discovered bugs with RaceBench's logs. Before ConAFL starts fuzzing, it requires the user to mark potential races in the source code (with any static race analysis tools). To avoid any biases resulting from static analysis, we directly provide it the correct race pairs. OpenRace and TSan analyze the target programs and report any detected potential races. We compare their outputs with RaceBench's code locations. If all locations of the race pairs of a bug are covered, we consider the bug is found out. Maple keeps running the target program in different thread interleavings until the program crashes. Both Maple and TSan require a given input to start the program. So we directly provide them with the correct input files, let them test for a maximum time of 24 hours, and check what bugs have been triggered.

1) Number of bugs discovered: The number of discovered bugs for each tool is listed in Table VI. TSan discovers 69.2% bugs and outperforms others. But it also reports lots of races in the target programs beyond RaceBench's bug list, and we have no way to verify without huge manual effort, so we only evaluate the recall rate. AFL++ discovers 46.9% bugs. However, as a general fuzzer, AFL++ does not record any kinds of scheduling or interleaving order, so its reports are not that helpful to reproduce and diagnose concurrency bugs. Maple guarantees that the reported bugs can be reproduced. During the experiments, we notice that Maple has a significant runtime overhead (as shown in Table VII), which explains why it only discovers 17.5% bugs. OpenRace analyzes fast but it crashes itself on some programs, resulting in a low recall rate of 7.0%. ConAFL does not perform well as it only discovers about 1.1% bugs. We will dig into the reasons of its poor performance later in §IV-D4.

2) Throughput of tools: The execution speed of the tools is shown in Table VII. Different from other tools, OpenRace analyzes the program statically for once and does not gain any benefits from running again. It seems that TSan and Maple have a relatively higher runtime overhead, but we think the overhead may be not completely root in their algorithms. We

TABLE VII: Throughputs of programs being tested by different bug discovery tools, and their ratios comparing to AFL++.

	OpenRace	AFL++	ConAFL		TSan		Maple	
	(s)[†]	(exec/s)	(exec/s)	relative[*]	(exec/s)	relative[*]	(exec/s)	relative[*]
blackscholes	8.02E-01	3.46E+03	1.26E+03	36.405%	3.03E+01	0.875%	1.97E+00	0.057%
bodytrack	1.86E+01	7.39E+00	1.38E+03	18617.363%	3.17E+00	42.897%	9.80E-01	13.274%
canneal	2.92E+00	4.72E+03	2.04E+02	4.332%	2.04E+00	0.043%	6.89E-01	0.015%
cholesky	-	4.85E+00	1.05E+03	21638.655%	5.59E+00	115.131%	6.87E-01	14.158%
dedup	2.07E+02	1.86E+04	3.65E+02	1.965%	7.01E+00	0.038%	1.79E+00	0.010%
ferret	8.63E+01	2.43E+04	1.09E+03	4.461%	3.59E+00	0.015%	1.20E+00	0.005%
fluidanimate	1.47E+00	1.72E+02	1.26E+02	73.384%	6.20E+00	3.595%	1.36E-01	0.079%
pigz	-	9.32E+03	1.21E+03	12.946%	2.07E+01	0.223%	1.41E+00	0.015%
raytrace	1.95E+01	4.43E+02	2.43E+02	54.776%	1.43E+00	0.324%	2.59E-01	0.058%
raytrace2	-	1.61E+01	7.68E+01	478.040%	1.71E+00	10.657%	1.23E-01	0.765%
streamcluster	1.18E+00	-	-	-	4.07E+00	-	1.04E+00	-
volrend	-	3.92E+01	2.17E+02	554.581%	5.04E+00	12.859%	9.73E-01	2.481%
water_nsquare	-	6.94E+00	1.70E+02	2443.680%	4.55E+00	65.527%	7.43E-03	0.107%
water_spatial	-	8.06E+00	1.33E+02	1649.200%	7.30E+00	90.582%	6.86E-01	8.505%
x264	-	1.45E+03	1.15E+03	79.756%	9.58E+00	0.663%	1.74E-01	0.012%

[†] As a static analysis tool, OpenRace only analyzes each program for once.
[*] The relative speed of a tool is measured by $speed(\text{tool})/speed(\text{AFL++}) \times 100\%$.

notice that MAPLE uses Python's `subprocess` module to start processes and we use the same module to start programs instrumented with TSAN too. Moreover, the two evaluated fuzzers (AFL++ and CONAFL) run really fast, and in some cases they are even faster than running the programs natively. This is because the fuzzers both use a fork server [39] implemented in C, which greatly reduces the startup time of target programs.

3) Impacts of code patterns and bug depth: As shown in Table VIII, the performance of the evaluated tools is greatly affected by the bugs' code patterns. Because the tools define their own rules (e.g., certain memory access patterns and thread API sequences) to recognize bugs, if a rule matches a bug code pattern well, hopefully, it will discover more bugs of that type. Also, different bug patterns in RaceBench can have different difficulties. The performance of AFL++ may be a good reference for estimating the difficulty of bug patterns, because it does not apply any concurrency-specific rules during testing.

In Fig. 6, we draw the percentage of discovered bugs according to the bug depths. There does not seem to be a clear relationship between the trace depth and the difficulty as claimed by LAVA [11]. We think that the proportion in trace does not directly reflect how hard it is to reach it. For example, the dominator nodes in the control flow graph of a target program are always reachable, but they can be scattered anywhere in the trace, even near the exit (depth=1). For interleaving depth, we can see a slightly decreasing trend in discovered bugs as the number of interleavings increases. But still the relationship is not clear as expected. We think the reason is that the tools do not have to strictly follow our trace, so they may trigger the bug with a smaller or larger number of interleavings than our answers.

4) Case Study: We pick CONAFL as a key to understand why existing tools have poor performance.

The evaluated tools are designed for different scenarios. CONAFL chooses a task that is more difficult but more realistic than others. While AFL++ just cares about searching for ill-formed inputs, OPENRACE, TSAN, and MAPLE focus purely on thread interleavings, but CONAFL aims to find out the inputs and thread interleavings together to trigger the bugs. How to balance the cost of exploring input space and thread interleaving space is a great challenge to be addressed in future studies on concurrency bugs.

TABLE VIII: Percentage of discovered bugs of different bug code patterns.

		OpenRace	AFL++	ConAFL	TSan	Maple
Atomicity Violation	RWA	6.1%	47.4%	1.4%	68.5%	14.0%
	WWA	8.3%	43.9%	0.5%	68.4%	14.2%
	WAW	6.1%	45.1%	1.0%	68.1%	23.4%
Order Violation	NoWait	0.0%	32.3%	0.0%	38.7%	25.8%
	Sleep	6.9%	31.0%	6.9%	62.1%	10.3%
	Reorder	2.3%	51.2%	4.7%	65.1%	23.3%

After diving into its implementation, we find that CONAFL applies for a fixed interleaving order at the potential race pair locations, trying to trigger the bugs. This behavior largely constraints the thread interleavings it can explore, so it may ignore some crucial interleavings. Moreover, it sometimes leads to a deadlock-like situation, where a bug requires Thread 1 to run before Thread 2 but another bug requires Thread 2 to run before Thread 1, making no progress and blocking the program.

E. Insights on Future Directions

Based on the evaluation, we would like to highlight a few insights on future directions of how to improve concurrency bug discovery tools.

Reproducibility: It is still hard for developers to locate the concurrency bug even if the program inputs and crash sites are given, because the essential thread interleaving orders are missing. Among the tools evaluated in our paper, only MAPLE provides support for reproducing the reported bugs.

Strategies on scheduling: A better way to schedule the threads is demanded, in order to explore more thread interleavings without introducing deadlocks or high runtime overheads. We suggest adopting a soft thread scheduling

strategy, e.g., using a probabilistic model rather than strict thread priority, so that all possible interleavings have a chance to be tested.

Combining input space and thread interleaving space: Both inputs and thread interleaving orders are important for concurrency bugs in the real world, because some interleavings are only feasible under ill-formed inputs. Thus, tools should coordinate the search strategies of both the input space and thread interleaving space. It will be good for tools to combine them and consider their effects on each other.

V. Related Works

A. Historic Concurrency Bug Benchmarks

Such benchmarks are built and validated with great human efforts, so they are usually limited in size. JaConTeBe [19] contains 47 confirmed bugs from 8 Java projects. GoBench [38] provides 82 real bugs from 9 applications in the Go language. DataRaceBench [18] contains 72 small programs written with the OpenMP library, intended to evaluate race detection tools. For native C/C++ programs, RadBench [13] consists of 10 real concurrency bugs in large projects, such as Apache web server and Chromium web browser; BugBench [22] is a general bug benchmark that includes 4 related to concurrency issues; and [36] also provides 20 real concurrency bugs used in its evaluation.

These historic concurrency bugs are collected from software issue tracking platforms, so they do not appear in a common version of the program, and moreover, they may require complicated configurations and inputs (e.g., network) to run, making them hard to be automated in concurrency bug discovery tool evaluation.

B. Automatic Bug Injection

LAVA [11] firstly proposes the idea of building datasets with synthetic bugs that can be triggered with certain inputs. Although it does not focus on concurrency bugs, it gives us the inspiration that bugs injected along an execution trace are guaranteed to be triggerable.

For concurrency bugs, CCMutator [15] mutates the usage of thread-related APIs to generate buggy code, such as switching the order of lock statements and erasing thread synchronizations. It enables mutation testing for multithreaded programs. But not all of these mutations can become bugs, and the mutated statements may not be reachable, thus not triggerable. DRInject [17] automatically injects data race bugs. With the help of dynamic debugging, it finds code locations that can be concurrently executed. Additional code at these locations will access a global variable to make a data race, which belongs to our Read-Write-Assume code pattern. Some code injected by DRInject are benign races instead of observable bugs, because it only intends to evaluate race detectors which are designed to report any possible races, no matter the reported bugs are false positives or not.

C. Concurrency Bug Discovery Tools

Race detection tools find data races by investigating shared memory accesses and synchronization events. The program is either analyzed statically without actually running it [8], [32], [34], or monitored dynamically during runtime [31], [2]. The commonly used techniques are: (1) point-to analysis [32], [8] that checks whether accesses from two threads can point to the same memory location, (2) happens-before analysis [31], [32], [8] that checks whether statements have to be executed in certain orders, and (3) lockset analysis [31], [2], [32], [28] that checks whether operations are properly protected by locks. As these applied analyses often broaden the range of possible data flows, race detectors report a large number of potential races but usually suffer from a high false positive rate.

Thread scheduling approaches search for a specific execution order to trigger concurrency bugs. PCT [9] scheduling algorithm provides a mathematically proven probability to trigger concurrency bugs. Dthreads [21] schedules threads in a deterministic way by replacing the pthread library to make concurrency bugs reproducible. Maple [37] controls scheduling by adjusting threads' priority and uses a coverage-guided heuristic to explore execution orders.

Mutation-based fuzzing has also been applied to automatically discover concurrency bugs. They often adopt the previously described approaches in a hybrid manner. As far as we know, RaceFuzzer [30] is the first fuzzer for race conditions. It combines race detection with a randomized thread scheduler and focuses on separating harmful races from benign races. ConAFL [20] combines static analysis and fuzz testing. It uses static analysis to locate sensitive concurrent operations and determine the buggy execution order to be tested. Muzz [10] applies thread-aware instrumentations that collect thread interleaving coverage and thread context information. All of these fuzzers are evaluated by self-chosen programs. We hope RaceBench will provide a convenient and fair evaluation for them.

VI. Discussion

A. Reproducibility of Discovered Concurrency Bugs

The reproduction and confirmation of concurrency bugs discovered by tools (e.g., fuzzers or detectors) are known as notorious problems, as the non-determinism of target programs is difficult to record or replay. We encourage future concurrency bug discovery tools to have such abilities to help developers find and debug concurrency bugs. In addition to constructing benchmarks to measure the performance of concurrency bug discovery tools, RaceBench also provides the corresponding inputs, thread interleaving orders, and scripts to reproduce the bugs (as mentioned in §IV-A). This will help the tools better confirm their ability to discover concurrency bugs.

B. Identifying Harmful Concurrency Bugs

The code inserted by RaceBench may introduce benign race pairs. For example, there is a race between line 5 and line 11 in Fig. 4. Their orders will not break the state assumption but may cause race detectors to alarm. Multithreaded programs in the real world may also contain benign races. We encourage the bug discovery tools to take additional efforts to identify harmful concurrency bugs.

C. Limitations

1) Bug Categorization: During the empirical study, we find some concurrency bugs lacking descriptions or too hard to understand. So there could be other bug root causes, types, or complicated code patterns beyond our list. RaceBench is only able to generate concurrency bugs caused by wrong program state assumptions. We have modeled bug code patterns involving only two threads, which covers 96% of concurrency bugs according to [23]. We leave it as future work to cover more bug patterns and concurrency bugs related to multiple threads.

2) Biases: Concurrency bugs synthesized by RaceBench can be biased to some degree. For instance, we only use input-needed and short-lived target programs, but real concurrency bugs can also exist in programs that do not require input or run for a long time. Apart from that, RaceBench crashes the target program to indicate a bug has been triggered, but concurrency bugs can also have other outcomes, such as hangup and wrong output. Actually they are features intentionally introduced, because many bug discovery tools rely on them, and we hope the dataset be usable for more tools. As stated earlier, rather than making the synthetic bugs representative in all aspects, we mainly focus on their root causes and code patterns, which are more essential parts of concurrency issues.

3) Bug Injection Approach: The trace-based approach enables RaceBench to inject triggerable concurrency bugs, but it also has some drawbacks. Concurrency bug discovery tools do not have to exactly follow our trace to find the bugs. This is intended but it cuts off the direct relationship between bug depth and bug difficulty. Moreover, RaceBench may have a higher probability to inject code inside loops because they appear more frequently in the trace. To mitigate it, RaceBench only keeps a certain number of looping locations in the trace according to user configurations.

4) Programming Languages: We only consider C/C++ programs in this paper. However, there are also some studies [33], [19] and tools [26], [25], [4] about concurrency bugs based on other programming languages. These languages may have special race protections (e.g., the Send and Sync traits in Rust) or thread synchronization primitives (e.g., the channels in Go), where RaceBench's bug patterns and bug injection approach may no longer be valid. But in general, we think the concurrency bug categorizations and the idea of injecting bugs along execution traces can still be helpful.

5) Report Precision of Tools: In evaluation, concurrency bug discovery tools sometimes report bugs that do not belong to RaceBench. Some of them are original bugs in the target programs and some are false alarms. Because RaceBench only verifies its own bugs, it is used to evaluate the recall rather than the precision of tools. We think this is an intrinsic problem of synthetic bug benchmarks.

VII. Conclusions

In this paper, we propose an automated approach RaceBench to automatically inject concurrency bugs into multithreaded programs. Based on a large-scale empirical study, we build models to represent the code patterns of concurrency bugs caused by wrong program state assumptions. RaceBench generates buggy code according to the patterns and injects them along an execution trace to make the bugs triggerable. We construct a dataset with 15 real-world programs, and use it to evaluate four concurrency bug discovery tools and one general fuzzer. Experiments show that there is still a large room for existing techniques to improve accuracy, reduce runtime overhead, and find strategies to balance the cost of exploring input space and thread interleaving space. Our benchmark also sheds light on the future direction of improvements in concurrency bug discovery. We have open sourced RaceBench and hope they can help improve automatic concurrency bug discovering techniques.

References

[1] Common vulnerabilities and exposures. https://cve.mitre.org/.

[2] Helgrind: a thread error detector. http://web.archive.org/web/20201210064432/https://valgrind.org/docs/manual/hg-manual.html. Accessed: 2020-12-10.

[3] Dirty cow. https://dirtycow.ninja/, 2016. Accessed: 2022-05-02.

[4] Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, mar 2006.

[5] Sara Abbaspour Asadollah, Hans Hansson, Daniel Sundmark, and Sigrid Eldh. Towards classification of concurrency bugs based on observable properties. In *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)*, pages 41–47. IEEE, 2015.

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

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008.

[8] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):144:1–144:28, 2018.

[9] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2325–2342. USENIX Association, 2020.

[11] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 110–121. IEEE, 2016.

[12] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings international parallel and distributed processing symposium*, pages 7–pp. IEEE, 2003.

[13] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. *HotPar*, 11:2–2, 2011.

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2123–2138. ACM, 2018.

[15] Markus Kusano and Chao Wang. Ccmutator: A mutation generator for concurrency constructs in multithreaded C/C++ applications. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineer-*

ing, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013, pages 722–725. IEEE, 2013.

[16] Chris Lattner. LLVM and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[17] Hongliang Liang, Mingyue Li, and Jianli Wang. Automated data race bugs addition. In Lorenzo Cavallaro and Andrea Lanzi, editors, *Proceedings of the 13th European Workshop on Systems Security, EuroSec@EuroSys 2020, Heraklion, Greece, April 27, 2020*, pages 37–42. ACM, 2020.

[18] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. Dataracebench: A benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 178–189. IEEE, 2015.

[20] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. A heuristic framework to detect concurrency vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 529–541, 2018.

[21] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. Association for Computing Machinery.

[22] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5. Chicago, Illinois, 2005.

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

[24] Dominik Maier, Heiko Eißfeldt, Andrea Fioraldi, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[25] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery.

[26] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. Association for Computing Machinery.

[27] Anne Marie Porrello. Death and denial: The failure of the therac-25, a medical linear accelerator. *Computer Science 440*, 2012.

[28] Polyvios Pratikakis, Jeffrey S. Foster, and Michael W. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In Michael I. Schwartzbach and Thomas Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 320–331. ACM, 2006.

[29] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. Association for Computing Machinery.

[30] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[31] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer – data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, NYC, NY, U.S.A., 2009.

[32] Bradley Swain, Bozhen Liu, Peiming Liu, Yanze Li, Addison Crump, Rohan Khera, and Jeff Huang. Openrace: An open source framework for statically detecting data races. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 25–32, 2021.

[33] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Vesal Vojdani and Varmo Vene. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, volume 30, pages 141–155. Citeseer, 2009.

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[36] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *ACM Sigarch Computer Architecture News*, 37(3):325–336, 2009.

[37] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 485–502, 2012.

[38] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. Gobench: A benchmark suite of real-world go concurrency bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 187–199. IEEE, 2021.

[39] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.