RaceInjector: Injecting Races To Evaluate And Learn Dynamic Race Detection Algorithms

Michael Wang[1], Shashank Srikant[1,2], Malavika Samak[1], Una-May O'Reilly[1,2]
{mi27950, shash}@mit.edu {malavika, unamay}@csail.mit.edu
[1]CSAIL, MIT [2]MIT-IBM Watson AI Lab

Abstract

There exist no sound, scalable methods to assemble comprehensive datasets of concurrent programs annotated with race conditions. As a consequence, it is unclear how well the multiple heuristics and SMT-based algorithms, that have been proposed over the last three decades to detect data races, perform. To address this problem, we propose RaceInjector—an SMT-based approach which, for any given program, creates arbitrarily many program traces of it containing injected race conditions. The injected races are guaranteed to follow the given program's semantics. RaceInjector hence can produce an arbitrarily large, labeled benchmark which is independent of how detection algorithms work. We demonstrate RaceInjector by injecting races into popular program benchmarks and generating a small dataset of traces with races in them. Among the traces RaceInjector generates, we begin to find counterexamples which four state-of-the-art race detection algorithms fail to detect. We thus demonstrate the utility of generating such datasets, and recommend using them to train machine learning-based models which can potentially replace and improve upon existing race-detection heuristics.

Keywords: Dynamic race detection algorithms, Race injection, Dataset generation, SMT-solvers

1 Introduction

Race detection in concurrent programs using their execution traces, *i.e.* dynamic analysis, has been shown to be in NP-hard [22]. Practical algorithms designed to detect race conditions hence rely either on heuristics [16, 18, 21, 23, 26, 27] or SMT-solvers [6, 10, 15, 24, 30]. The goal of these algorithms is to start with a trace, and determine if two conflicting accesses in different threads to a shared variable can occur concurrently in an alternate execution of the program. Despite numerous such algorithms having been proposed over the last few decades, it is surprising that there exist no comprehensive benchmarks comprising industry-grade software projects which have races annotated in them—either annotated in the source code or in the traces—which would help rigorously evaluate these algorithms. Consequently, it is unclear what the true classification accuracy rates (true-positive, true-negative, *etc.*) of these algorithms are. For instance, none of the larger benchmark datasets such as DaCapo [2], which have all been repeatedly and extensively used in the evaluation of multiple race detection algorithms cited above, have any ground-truth annotations.

One key reason for the lack of such datasets is the absence of sound, scalable methods to assemble them. A few prior works [8, 12, 19, 31] have released expert-annotated datasets containing races. However, they are too small to be effective. Jacontebe [19] contains a total of 19 data race bugs, RadBench has 10 bugs, while GoBench [31] has 103 bugs for the Go language. With such small datasets, it is hard to evaluate if current algorithms commonly miss any race patterns.

Another approach to assembling such datasets has been to run SMT-based race detection algorithms on industry-grade software projects [8]. SMT-based race detection approaches, by their design, can provably detect true-positive race conditions. The detected true-positive race conditions are used as annotations and released as datasets. A major limitation of these approaches however is the relatively small number of constraints that SMT-solvers can solve at once. Longer, more complex real-world software produce longer execution traces, which in turn non-linearly increase the number of constraints which SMT-solvers have to solve. Detection algorithms typically circumvent this limitation by breaking the trace into fixed-length windows and solving each window as if they are independent of others [11]. The assumption of independence of windows is not practical, thus limiting the number and quality of races that can be detected. Moreover, assembling a dataset using race conditions detected by known detection approaches limits the nature of races we can test other detection algorithms on. We further discuss other prior works in Section 5.

As a direct consequence of this absence of scalable methods to assemble comprehensive annotated datasets, we argue that the metrics that have been employed to evaluate any new race detection algorithm do not accurately represent their performance. Further, we argue that the lack of such annotated datasets has potentially stifled the state-of-the-art.

- **Evaluating race detection algorithms.** Presently, the efficacy of newly proposed race detection algorithms is primarily measured by the increase in the number of identified race conditions when compared to a previous algorithm. Among heuristics-based algorithms [16, 18, 21, 23, 26, 27], each new algorithm has successively proposed a set of rules which purportedly improves upon previous algorithms. The newer algorithms then demonstrate detecting races which the previous algorithms did not. The newly detected races are verified manually by experts. In this process, it is unclear how many true-positives and false-negatives each new set of rules introduce. When an algorithm reports finding N (say) new bugs which a previous algorithm had not found, it is unclear what N is relative to—the total number of bugs which either of these algorithms are supposed to find in the first place. With surprisingly little attention paid to this essential metric, it is unclear what the state of progress in data race detection algorithms has been over the years.

Further, it is unclear whether an improvement proposed using a set of new rules results in detecting newer *classes* of race conditions, assuming there exist multiple semantic classes of use-cases in a program's execution behavior which manifest as race bugs. It is quite possible that a proposed improvement to an existing algorithm, while detecting a few new bugs, may not necessarily cover a significantly larger set of such semantic classes. An annotated dataset with a diverse set of bugs in them is the first step in establishing and quantifying the semantic classes a detection algorithm covers.

- **Training ML models.** We posit that it is possible for data-driven methods to replace the many heuristics which have been proposed over the years for race detection. Heuristics for race detection involve learning to draw edges between events of interest in a program's execution trace, and identifying cliques of connected or disconnected events. These cliques are then used to reason about and infer the existence of potential races. These heuristics typically guarantee soundness only for the first race they detect.

Given the inadequacy of existing guarantees, machine learning (ML) models provide a practical alternative. ML models have been shown to outperform expert-crafted heuristics when reasoning about graph-based data in the domains of compiler optimization [4], network-graph analysis [3], pointer analysis [13], fault localization [20] and more. While such learned models will not be able to guarantee soundness even for the first detected race, they may well offer an improved performance in reasoning with trace-based information over expert-crafted heuristics. However, a key requirement to train any ML model is a sizeable, well annotated dataset.

Our solution–SMT-based race injection. We propose injecting races into existing concurrent software as an approach to scalably create comprehensive, annotated datasets. Specifically, we inject race conditions into an execution trace of a given program. We choose to inject into the execution trace rather than the source code because injecting races into the program source does not guarantee the race manifesting into every execution trace of the program. This makes it difficult to evaluate race detection algorithms that employ dynamic analysis methods. We refer to the execution trace before injection as the **base trace**.

Adding two consecutive events that are conflicting (*e.g.* a read event immediately followed by a write event to the same variable without any synchronization mechanism) is the simplest way to inject a race into the base trace. More difficult is to inject conflicting events that *could possibly* occur consecutively in a different, random execution of the program. Our goal thus is to generate traces in which such conflicting events appear far apart in them, making them non-trivial to detect. To achieve this, we propose RaceInjector, which injects a trivial race condition into any trace, and then uses an SMT-solver to find an alternate, valid reordering of the base trace where the conflicting events appear far apart. This approach is independent of how any race detection algorithm works and the program generating the trace into which the trivial race is injected. Our method importantly guarantees the injection of a race while maintaining the semantics of the base trace. Thus, RaceInjector generates traces with injected races appearing at random, valid locations, mimicking a thread scheduler scheduling a program containing a valid race condition. To ensure RaceInjector generates race conditions that appear arbitrarily far apart in a trace, we propose a method which circumvents practical limitations of SMT-solvers while guaranteeing semantics of the base trace. We describe our approach in detail in Section 3. In this work, we generate a small dataset and demonstrate it on the research questions that it helps address. We also show how it can be easily extended to a comprehensive dataset. Among the few traces we generate, we find traces with race conditions that current state-of-the-art race detection methods fail to detect. This demonstrates one immediate utility of RaceInjector. A sample of these counterexamples can be found at https://github.com/ALFA-group/RaceInjector-counterexamples. In Section 4, we discuss other implications of RaceInjector.

2 Background

In this section, we provide a brief background on traces, race conditions, and the race detection algorithms that we evaluate in this work.

Traces. We assume a sequential consistency memory model [17], where a program trace is a sequence of events on executing a program. An event can be denoted as a tuple `<op, thread, loc>`, where `op` is the operation that is performed, `thread` is the thread which performed the operation, and `loc` is the file and line which performed the event. An `op` can be one of the following: `read(x)`, `write(x)`, `lock(L)` (thread has acquired a lock on `L`), `unlock(L)` (thread has released the lock on `L`), `fork(T)` (a thread has forked a new thread `T`), `join(T)` (a thread `T` has joined the current thread). Nondeterminism in the scheduler can cause one program to have many possible traces.

Correct reordering of traces. A trace σ^* is said to be a *correct reordering* of trace σ if it has the following properties:

	thread 1	thread 2
1		w(z)
2		acq(lock)
3		w(x)
4		rel(lock)
5		w(y)
6	w(y)	
7	acq(lock)	
8	w(x)	
9	rel(lock)	
10	w(z)	

Figure 1. Example of a *potential* race condition on lines 1 and 10, an *observed* race condition on lines 5 and 6, and a safe access on lines 3 and 8.

1. *Thread Ordering:* The order of intra-thread events remains the same in both σ and σ^*.
2. *Read-Write Consistency:* For every read event in σ and σ^*, the most recent write event to the variable that is read remains the same. This is to ensure that control flow will remain the same.
3. *Locking Semantics:* σ^* does not violate the semantics of synchronization events, such as locks and unlocks.

Intuitively, σ^* is a correct reordering of σ if any program that produces σ can also produce σ^*.

Data races. A data race occurs when two threads access the same variable without any synchronization, where at least one of these accesses is a write. Race conditions can be classified as *observed* or *potential* race conditions. An *observed* race condition is where a race condition actively occurs in a trace, where two threads attempt to concurrently access a shared variable where at least one of the accesses is a write. Observed data races are trivial to detect. A *potential* race condition is where no data race is observed, but there exists another correct reordering of the trace where an observed race condition could occur. Potential data races are much harder to detect. See Figure 1 for an example. Events 5 and 6 in red are an example of an observed race, where two conflicting events occur simultaneously. Events 1 and 10 in orange are an example of a potential race, where they do not occur consecutively in this trace, but could in another correct reordering. Events 3 and 8 are not racy due to synchronization mechanisms.

Race Detection. We evaluate the following algorithms in our work: Happens-Before (Lamport, 1976 [18]), Schedulable Happens-Before (Mathur *et.al.*, 2018 [21]), Weak Causally Precedes (Kini *et.al.*, 2017 [16]), and SyncP (Mathur *et.al.*, 2021 [23]). Happens-Before (HB) creates a partial ordering in a trace between each intra-thread event, as well as between any critical regions, in the order of their appearance in the trace. A partial order on a set S is a relation on S that is reflexive, anti-symmetric, and transitive. Weak Causally-Precedes (WCP) is a weakening of HB, meaning there are fewer edges. This allows WCP to catch more races while maintaining soundness for the first race. WCP only draws edges between critical regions that have conflicting accesses to a shared variable, and draws the edges between the release events and the critical events. Schedulable Happens-Before (SHB) is a strengthening of HB, and discovers fewer races than HB. However, SHB guarantees soundness past the first race. SyncP performs a scan for any race conditions that do not reverse the order of any critical sections, and is not a partial order. Unlike partial order based techniques, SyncP is unable to detect any races that would require critical sections to be reversed. However, it is guaranteed to catch all races which do not need to reorder the critical sections.

3 Method

In this section, we describe how we inject synthetic data races into program traces. We begin with a motivating example.

Motivating example. The program $\mathcal{P}$ in Figure 2a reads and writes to two variables x and y. One of its possible execution traces is shown in Figure 2b. This is the **base trace** for our injection. Originally, this program does not contain a race condition. We note that a thread switch occurs after event 6. We can trivially inject a race directly after the thread switch by adding two write events to the trace (lines 5-6, Figure 2c), resulting in an *observed* race condition.

To invoke non-trivial reasoning to detect our injected race condition, we propose using an SMT-solver to find a correct re-ordering of the events in a trace such that (a) the original trace's semantics hold, and (b) the inserted events are *moved apart* by some L events. One such alternate reordering can be seen in Figure 2d. Recall the definition of a data race: two events that access the same variable, at least one of which is a write, that occur in an unsynchronized manner. Our solution to generating these data races has three steps which we describe in detail: **Step 1:** Instrument and execute a program; collect base traces of relevant events. **Step 2:** Add a trivial data race to a base trace, and finally, **Step 3:** Use an SMT-solver to make the added race harder to detect.

Step 1. Trace collection. We start by logging a sequential trace of data accesses and thread synchronizations in a program. See Figure 2b for an example trace. Races are then injected into the collected *base traces* and analyzed. This decoupling of instrumentation and race injection allows for several instrumentation frameworks to be used. We use MCR [9] which instruments using the ASM framework [1]; RoadRunner [7] which also instruments with ASM, and Calfuzzer [14] which instruments using the SOOT compiler framework [29]. SOOT and ASM allow the instrumentation frameworks to modify the bytecode and intercept relevant events as they occur during execution.

Step 2. Adding a trivial race. To modify a base trace to add a trivial race condition, we insert two new write events right where there is a context switch between threads. See Figure 2c for an example of modifying the base trace in Figure 2b. The writes are made to a new, dummy variable to ensure the semantics of the original program remains the same. We only inject one race into the base trace at a time before saving it.

```java
class Test {
        static int x;
        static int y;

        void inc1() {
             synchronized(lock){
                 x++;
             }
        }

        void inc2() {
             synchronized(lock){
                 y++;
             }
        }
    }

public static void main(String[] args) {
  Test test = new Test();
  fork { test.inc1(); }
  fork { test.inc2(); }
}
```

(a) A simple program $\mathcal{P}$ with two threads and no pre-existing race conditions

	thread 1	thread 2
1		`acq(lock)`
2		`r(x)`
3		`w(x)`
4		`rel(lock)`
5	`acq(lock)`	
6	`r(y)`	
7	`w(y)`	
8	`rel(lock)`	

(b) Original execution trace of $\mathcal{P}$ (Fig 2a)

	thread 1	thread 2
1		`acq(lock)`
2		`r(x)`
3		`w(x)`
4		`rel(lock)`
5		`write(z)`
6	`write(z)`	
7	`acq(lock)`	
8	`r(y)`	
9	`w(y)`	
10	`rel(lock)`	

(c) Execution trace for Figure 2a, with a trivial race injected.

	thread 1	thread 2
1	`write(z)`	
2	`acq(lock)`	
3	`r(y)`	
4	`w(y)`	
5	`rel(lock)`	
6		`acq(lock)`
7		`r(x)`
8		`w(x)`
9		`rel(lock)`
10		`write(z)`

(d) The trace after running a solver to move the racy events apart.

Figure 2. A sample program $\mathcal{P}$ (left), and how we can inject a race condition in its execution trace (right).

Step 3. Using an SMT-solver to move apart the added race. After having injected a trivial race comprising consecutive conflicting events, the goal is to then find an alternate valid interleaving where the race-events are farther apart.

We set up n SMT variables v, where each variable $v_j \in [1, n]$ corresponds to an event that appears in the base trace containing a total of n events. The value of v_j signifies the location index where the event should appear. In trace 2b for example, if v_1 corresponds to event `w(x)`, the assignment for v_1 corresponding to the trace would be 3, the location index `w(x)` appears in the trace. Similarly, if v_2 corresponds to event `w(y)`, the assignment of v_2 corresponding to the trace is 7. For a trace σ, we then formulate a constraint equation in a way that solving the constraints yields a valid assignment made to each v_j which results in an alternate trace σ^*. Our constraints must ensure that the alternate trace is a correct reordering as defined in Section 2 (thread ordering, read-write consistency, locking semantics). These constraints have been commonly defined in race detection to find alternate reorderings [11, 25, 30]. Readers can refer to Said et al. [25] for details. However, we introduce the following constraints in order to inject race conditions into base traces:

- **Distance between conflicting events.** We supply a hyperparameter L which constrains the distance between the inserted racy events.
- **Additional constraints.** We additionally ensure that the indices assigned to each v_j is positive, unique, and lies in the interval $[1, n]$.

We supply a conjunction of these constraints to an SMT-solver which produces an assignment to each v_j. These assignments correspond to a new, valid reordering of each event appearing in σ, thus resulting in a new trace σ^*. Further, σ^* contains the previously trivially injected race events now at least L events apart, and ensures the same execution semantics as that of σ. We elide details of the symbolic encodings of these constraints for the sake of brevity.

Moving events arbitrarily apart in a trace. The number of constraints in the conjunction described above which generates σ^* is typically prohibitively large for SMT-solvers to solve. Our insight to circumvent this practical problem is to incrementally move the introduced conflicting events farther apart. We start with reordering the conflicting events (which initially appear consecutively when injected) and the events surrounding it in a window of fixed size. For the events in this window, we generate the constraints described above and run RaceInjector. We choose a window size in a way that the number of constraints does not overwhelm the solver. Once RaceInjector generates a reordering for the events in the window, we slide the window over by a fixed length and run RaceInjector again on the events that appear in the shifted window. We ensure the shifted window contains at least one of the two conflicting events we introduce, which will have been reordered from their initial, consecutive indices. Running RaceInjector iteratively over smaller, fixed-length windows k times is computationally much more efficient than running the solver on a large number of events just once—the number of constraints tend to grow superlinearly

	Base traces			RaceInjector-generated traces		
Program	**#Inj. pts**	**Length**	**#Thrd**	**#Gen. traces**	**Avg race dist.**	**Max race dist.**
ArrayList	207	677	27	207	$128_{\pm 111}$	558
TreeSet	130	756	22	130	$122_{\pm 115}$	526
LinkedList	1767	14937	451	160	$112_{\pm 124}$	851
Stack	2036	11372	451	100	$87_{\pm 74}$	458
Jigsaw	3394	97110	78	467	$693_{\pm 777}$	7396

Table 1. Overview of RaceInjector results on a benchmark of programs. Column 1 lists the different program benchmarks in which RaceInjector injects races. Columns 2,3,4 describe the base traces. The remaining columns describe the traces generated by RaceInjector. *Inj. pts.* refers to the number of injection points available in the base trace; *Thrd* the number of program threads.

Algorithm	ArrayList	TreeSet	Jigsaw	Stack	LinkedList	# Missed
HB (1979) [18]	✓	✓	✓	✗	✗	60 (5.6%)
SHB (2018) [21]	✓	✓	✓	✗	✗	64 (6%)
WCP (2017) [16]	✗	✓	✗	✗	✗	21 (2%)
SyncP (2020) [23]	✓	✓	✓	✗	✗	22 (2%)

Table 2. Counterexamples generated by RaceInjector. A ✓ signifies there exists at least one trace among the RaceInjector-generated traces which is not detected by the corresponding algorithm. *# Missed* reports the number of traces the algorithm misses to detect (percentage mentioned within parenthesis).

with the number of events needed to reason about. We elide a proof of correctness of our approach for the sake of brevity.

4 Results & Discussion

We demonstrate RaceInjector by using it with a suite of program benchmarks used in prior works to generate a sample of base traces containing race conditions (Section 4.1). Among the generated traces, we also find counterexamples which state-of-the-art race detection algorithms fail to detect (Section 4.2).

4.1 Generated dataset: Quantitative description

We employ RaceInjector to generate only a small, demonstrative dataset comprising ~1000 total traces in this work. This is nonetheless sufficient to show the ease with which RaceInjector can be extended to generate a comprehensive dataset. We ran our experiments without any parallelization using Java version 11.0.18, on a CPU running Ubuntu 18.04 with 96 GB RAM.

Base traces. We run each of the five program benchmarks listed in Table 1 once on the testcases from `Calfuzzer` [14]. This instruments and generates one execution trace each for each of the five programs. Table 1, columns 2, 3 and 4 document statistics of each program's base trace. Each program has a different number of threads (column 4). Consequently, each trace presents a different number of points of injection (column 2) to introduce a trivial race—these are points at which thread context-switches occur. We run RaceInjector on each program's trace (except for Jigsaw) for one hour, with a goal of injecting races into as many entry points as possible within the allotted one hour. Since Jigsaw is a significantly larger program than the rest, as seen by the length of an average trace generated (column 3), we run RaceInjector for ~10 hours instead on the Jigsaw trace.

RaceInjector-generated traces. Running RaceInjector results in a total of ~1000 traces (sum of column 5). Columns 5, 6 and 7 in Table 1 document the statistics of the generated traces. Note again, these are all guaranteed to contain race conditions. The set of traces generated by RaceInjector for any one program will all have the same length (column 3) because each trace is just one possible valid reordering of the original. We find the number of traces (column 5) generated in one hour of running RaceInjector is roughly the same across the different program benchmarks (ignoring Jigsaw). In columns 6 and 7, we report the average distance and the maximum distance between the injected conflicting events in the traces generated by RaceInjector, which is measured by counting the number of events between them. We observe the average distances (column 6) to be significantly greater than zero, suggesting that the injected races, which are initially placed consecutively, end up significantly apart in the generated traces. From the standard deviations (subscripts in column 6), we see very high variance in the distance between injected races, suggesting the heterogeneity in the potential race conditions introduced by RaceInjector. In `ArrayList` and `TreeSet`, the maximum distance between the injected race events (column 7) nearly span the length of their base traces (column 3). This demonstrates the flexibility offered by RaceInjector to generate any number of traces with guaranteed races that are varied in the locations they appear in. This is a desirable feature for a high-quality annotated dataset of such concurrent programs.

Discussion: Scaling the dataset. To assemble a comprehensive dataset, we recommend the following procedure: first, execute a program multiple times to obtain different base traces. Second, run RaceInjector on every possible point of injection in each base trace without constraining the time taken to complete this process. Both the steps are easy to parallelize. We plan to extend the race detection algorithms evaluated to include more recent tools such as RPT [28], static race detection methods like Infer [5], and SMT-based methods [11].

Discussion: True-negative samples. In proposing RaceInjector, we only consider the problem of generating true-positive race conditions. In our larger goal to assemble a comprehensive dataset large enough to train machine learning models, we would also need a method to assemble examples of true-negative cases in our dataset. As most pairs of conflicting accesses in software are not races, we can randomly sample such accesses, verify them using simple algorithms like HB, and label them as true-negatives. Machine learning algorithms are tolerant to a small number of mislabeled samples.

4.2 Counterexamples to SOTA algorithms

Table 2 shows that RaceInjector can easily produce counterexamples to state-of-the-art (SOTA) detection algorithms. This is important because it reveals for the first time their sensitivity as well as guides future work to clearly define new rules and algorithms. We can potentially study the contribution of the different rules present in the heuristics of different algorithms in the decisions made by the algorithms. Thus, access to a comprehensive set of counterexamples can potentially empirically justify the different rules implemented by these algorithms, and can also point to equivalences between some of these rules. While we do not directly study the counterexamples, this is a compelling direction for future work.

We now analyze the counterexamples generated by RaceInjector that the different algorithms fail to detect. They are available at: https://github.com/ALFA-group/RaceInjector-counterexamples. The analysis that follows should be interpreted with caution because the number of counterexamples is relatively small (fewer than 100), which does not support statistical comparison tests. We will evaluate these claims rigorously on a larger dataset in future work. This analysis is instead indicative of the questions that can be studied.

SHB vs. HB. SHB guarantees soundness after the first detected race condition it detects in exchange for detecting fewer races overall compared to HB. We should then expect SHB to detect fewer races on average, and conversely miss detecting more races. This is what we observe: SHB fails to detect 64 bugs RaceInjector generates (column 7, Table 2), 4 more than HB. That said, the total percentage of bugs that the algorithms fail to detect are roughly the same (~6%). We will, in the future, also compare whether they fail on the same set of counterexamples.

SyncP vs. WCP. Despite SyncP following and improving upon WCP, we do not see a notable increase in its performance. Both fail to detect 2% of the generated races. On the other hand, both algorithms improve upon HB, so it is expected to see a improvement of ~4% in the races they fail to detect when compared to HB.

`LinkedList` *and* `Stack`. We observe that none of the injected race conditions in `LinkedList` and `Stack` fail any algorithms (columns 5, 6, Table 2). Besides the low number of samples generated, a possible reason could be the large number of unsynchronized threads. These two programs have the largest number of threads relative to the length of their traces (Table 1). We suspect these threads mostly involve unsynchronized accesses, making the injected races relatively easy to detect as well. If in the future our hypothesis that the threads mostly involve unsynchronized accesses holds, we will filter out such injection points to reduce the number of trivially detectable races in our dataset.

Table 2 indicates that RaceInjector is able to generate race conditions which no SOTA method detects. This implies RaceInjector finds locations in a program trace which are complex to reason about. To finish, RaceInjector makes the widespread adoption of classification accuracy-related metrics (true-positive, false-positive, true-negative, false-negative) now possible when evaluating and comparing race detection algorithms.

5 Related work

Prior work closest to RaceInjector has mostly compiled known bugs that have been found over the years. Because these bugs have already been found, it is difficult to evaluate the capability of new approaches to detect new bugs. Additionally, these datasets are far too small to train a machine learning model, the largest being 985 races in Jbench [8]. JaConTeBe [19] is a benchmark of Java concurrency bugs, which scrapes past papers and aggregates a list of 47 distinct bugs along with their causes. GoBench [31] is a dataset of 103 bugs in Go, scraped from Github. RADBench [12] is a dataset composed of snapshots of open-source software projects with 10 total known bugs. Jbench [8] is a dataset of Java data races, aggregated from artifacts of existing race detection tools, and contains 985 unique data races. Jbench contains 6 real-world applications, and 42 custom testcases that were written during development of previous race detection tools. Typically, all these benchmarks are curated by either expensive manual analysis, or have been assembled using existing tools, which greatly limits their usefulness in evaluating and improving extant race detection algorithms while RaceInjector is fully automated. Additionally, since many of the samples have been curated using existing tools, a machine learning model trained on these samples will be unlikely to outperform the original tools used to find them.

6 Acknowledgments

This work was partially supported by a grant from the MIT-IBM Watson AI lab.

References

[1] [n. d.]. ASM bytecode analysis framework. https://asm.ow2.io/

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[3] Benjamin Bowman, Craig Laprade, Yuede Ji, and H Howie Huang. 2020. Detecting Lateral Movement in Enterprise Computer Networks with Unsupervised Graph AI.. In *RAID*. 257–268.

[4] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

[5] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

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

[7] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toronto, Ontario, Canada) *(PASTE '10)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1806672.1806674

[8] Jian Gao, Xin Yang, Yu Jiang, Han Liu, Weiliang Ying, and Xian Zhang. 2018. Jbench: A Dataset of Data Races for Concurrency Testing. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 6–9. https://doi.org/10.1145/3196398.3196451

[9] Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. *SIGPLAN Not.* 50, 6 (jun 2015), 165–174. https://doi.org/10.1145/2813885.2737975

[10] 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*. 337–348.

[11] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. *SIGPLAN Not.* 49, 6 (jun 2014), 337–348. https://doi.org/10.1145/2666356.2594315

[12] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. 2011. RADBench: A Concurrency Bug Benchmark Suite. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*. USENIX Association, Berkeley, CA. https://www.usenix.org/conference/hotpar-11/radbench-concurrency-bug-benchmark-suite

[13] Minseok Jeon, Myungho Lee, and Hakjoo Oh. 2020. Learning graph-based heuristics for pointer analysis without handcrafting application-specific features. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

[14] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 675–681.

[15] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (oct 2018), 29 pages. https://doi.org/10.1145/3276516

[16] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. *SIGPLAN Not.* 52, 6 (jun 2017), 157–170. https://doi.org/10.1145/3140587.3062374

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

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

[19] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–189. https://doi.org/10.1109/ASE.2015.87

[20] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[21] 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. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.

[22] 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. https://doi.org/10.1145/3373718.3394783

[23] 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. https://doi.org/10.1145/3434317

[24] Jake Roemer, Kaan Genç, and Michael D Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 747–762.

[25] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-Based Analysis. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327.

[26] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.

[27] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. *Sound Predictive Race Detection in Polynomial Time.* Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[28] Mosaad Al Thokair, Minjian Zhang, Umang Mathur, and Mahesh Viswanathan. 2023. Dynamic Race Detection with O(1) Samples. *Proc. ACM Program. Lang.* 7, POPL, Article 45 (jan 2023), 30 pages. https://doi.org/10.1145/3571238

[29] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[30] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*. Springer, 256–272.

[31] Ting Yuan, Guangwei Li, Jie Lu, Chen Liu, Lian Li, and Jingling Xue. 2021. GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 187–199. https://doi.org/10.1109/CGO51591.2021.9370317