Efficient Scalable Thread-Safety-Violation Detection

Finding thousands of concurrency bugs during testing

Guangpu Li
University of Chicago

Shan Lu
University of Chicago

Madanlal Musuvathi
Microsoft Research

Suman Nath
Microsoft Research

Rohan Padhye
University of California, Berkeley

Abstract

Concurrency bugs are hard to find, reproduce, and debug. They often escape rigorous in-house testing, but result in large-scale outages in production. Existing concurrency-bug detection techniques unfortunately cannot be part of industry's integrated build and test environment due to some open challenges: how to handle code developed by thousands of engineering teams that uses a wide variety of synchronization mechanisms, how to report little/no false positives, and how to avoid excessive testing resource consumption.

This paper presents TSVD, a thread-safety violation detector that addresses these challenges through a new design point in the domain of active testing. Unlike previous techniques that inject delays randomly or employ expensive synchronization analysis, TSVD uses lightweight monitoring of the calling behaviors of thread-unsafe methods, not any synchronization operations, to dynamically identify bug suspects. It then injects corresponding delays to drive the program towards thread-unsafe behaviors, actively learns from its ability or inability to do so, and persists its learning from one test run to the next. TSVD is deployed and regularly used in Microsoft and it has already found over 1000 thread-safety violations from thousands of projects. It detects more bugs than state-of-the-art techniques, mostly with just one test run.

CCS Concepts • **Software and its engineering** → **Software maintenance tools**; Empirical software validation; • **Computer systems organization** → *Reliability*;

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359638

Keywords thread-safety violation; concurrency bugs; debugging; reliability; scalability

ACM Reference Format:
Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient Scalable Thread-Safety-Violation Detection: Finding thousands of concurrency bugs during testing. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3341301.3359638

1 Introduction

Concurrency bugs are hard to find, reproduce, and debug. Race conditions can hide in rare thread interleavings that do not occur during testing but nevertheless show up in production. Thus, it is not uncommon for even the simplest of concurrency bugs to escape rigorous testing but result in large-scale outages in production. Even when detected, these bugs are hard to reproduce in the small. As such, tools for finding concurrency bugs is an actively studied research area [7, 22, 43, 47, 58] with many open challenges.

This paper specifically deals with a class of concurrency errors we call *thread-safety violations* or TSVs. Libraries and classes specify an informal *thread-safety* contract that determines when a client can and cannot concurrently call into the library/class. A TSV occurs when a client violates this contract. Figure 1 shows an example. The implementation of the `Dictionary` class allows multiple threads to concurrently call read operations, such as `ContainsKey`, but requires write operations, such as `Add`, to only be called in exclusive contexts. By violating this contract, Figure 1 contains a TSV.

This paper is motivated by the prevalence of such TSVs in production code. In fact, our evaluation discovered that the pattern in Figure 1 is fairly common as developers erroneously assume that concurrent accesses on different keys are "thread safe." TSVs, while common, can have drastic consequences, including unexpected

Guangpu Li is supported by an SOSP 2019 student scholarship from the National Science Foundation.

```
1  // Dictionary dict
2  // Thread 1:
3  dict.Add(key1, value)
4  // Thread 2:
5  dict.ContainsKey(key2)
```

Figure 1. A thread-safety violation (TSV) bug

crashes or worse, silent data corruption that are difficult to diagnose and fix.

This paper describes TSVD, a dynamic-analysis tool for detecting TSVs. The tool is specifically designed to be used in an integrated build and test environment, such as Bazel [4] or CloudBuild [15]. Here, hundreds to thousands of machines build and run tests on software modules from thousands of engineering groups. We have designed the system end to end to operate at this scale. For instance, it is important to design tools that produce *little to no false bug reports*. A small loss of productivity per-user in chasing false bugs can quickly multiply to huge productivity losses across the organization. Similarly, it is important to design tools that incur *small resource overhead*. For instance, if a tool incurs a 10× run-time overhead or requires rerunning the same test 10× times, then the test environment needs to allocate 10× more machines to provide the same turnaround time to its users.

Since TSVs generalize the notion of data races to coarser-grain objects, techniques for detecting TSVs are naturally related to data-race detection (Section 2 has more discussion). But existing approaches fall short for our purposes.

Considering the goal about having little to no false bug reports, the approach of *active* delay-injection [14, 47, 58] is promising. This approach directs program threads towards making conflicting accesses by delaying threads at strategic locations at run time. Consequently, this approach only reports bugs that are validated at runtime, and thus are not false bugs.

Considering the goal about having small overhead however, existing active delay-injection techniques do *not* work. To better understand their limitations, we can broadly classify them based on the inherent tradeoff between the cost of injecting too many delays versus the sophistication of analysis required to select where to inject delays:

At one extreme, as illustrated in the lower-right corner of Figure 2, tools like RaceFuzzer [58] and CTrigger [47] conduct static or dynamic analysis of memory accesses and synchronization operations in a program to identify potential buggy locations, and selectively inject delays only at these locations. While this reduces the number of delays injected, these techniques bear the cost of performing analysis, either through run-time overhead for dynamic analysis or the inscalability of precise static analysis required for identifying potentially conflicting accesses.

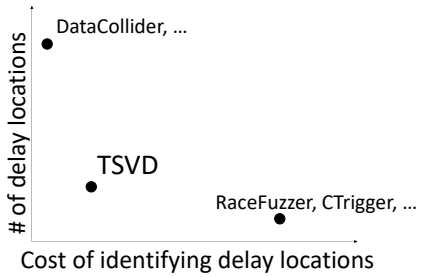


Figure 2. The design space of active testing

At the other extreme, as illustrated in the upper-left corner of Figure 2, tools like DataCollider [14] conduct little to no analysis, but instead probabilistically inject delays at many program locations. Such tools require many repeated executions of the same test to find a bug, which we deem unacceptable in our context as described above.

The goal of TSVD is to explore the middle ground in the design space of delay-injection tools, as illustrated in Figure 2. TSVD uses lightweight instrumentation and dynamic analysis to identify potential conflicting accesses without paying the overhead while being effective in finding bugs.

First, TSVD performs a lightweight *near-miss* tracking to identify potential thread-safety violations. It dynamically tracks threads that invoke conflicting methods to the same object close to each other in *real time*, without monitoring or analyzing synchronization operations.

Second, TSVD uses a *happens-before (HB) inferencing* technique that leverages the feedback from delay injections to prune call pairs that are likely ordered by the happens-before relation [34] and thus cannot execute concurrently. The key observation is that if there is a HB relationship between two events a and b, then a delay in a will *cause* a delay in b. By inferring this causality, TSVD avoids injecting delays at these pruned pairs subsequently. Note, that unlike HB analysis performed by dynamic data-race detectors, HB inference does not require modeling, identifying, and analyzing synchronization operations in a program, which is expensive and complicated given the wide variety of synchronization mechanisms in production software.

An interesting benefit of the near-miss tracking and the HB inferencing is that it only requires *local* instrumentation. That is, only the libraries/classes whose thread-safety contract is checked need to be instrumented. All other parts of the code, including synchronizations such

as forks, joins, locks, `volatile` accesses, and ad-hoc synchronization, do not need to be instrumented. This modularity dramatically reduces the runtime overhead. Furthermore, it enables incremental "pay-as-you-go" thread-safety checking of different classes and libraries, greatly simplifying the deployment of TSVD to thousands of engineering groups, who can configure the tool based on their needs.

Another key benefit of TSVD is that it is *independent* of the underlying concurrency programming model and *oblivious* to synchronization semantics, and thus can work across different programming models, such as threads, task-based programming, asynchronous programming, etc., and be applied to software using a variety of synchronization primitives.

Finally, TSVD compacts its delay injections, delay-location identification and adjustment all into the *same* run. In contrast, RaceFuzzer and CTrigger, require at least two runs, first to do the analysis and the second to inject delays. In many cases, as our evaluation shows, TSVD finds a large fraction of bugs in the first test run, making the best use of testing resources. This also allows TSVD to be effective where running many repetitions of the same test is not feasible due to large test setup and shutdown overheads.

We evaluate TSVD on tests from roughly 43,000 software modules developed by various product teams at Microsoft. By design, TSVD produces no false error reports, with each report containing stack traces of the two conflicting operations that violate the thread-safety contract. Overall, TSVD has found over 1,000 previously-unknown bugs, involving over 1,000 unique static program locations and over $20K$ unique call-stacks. To validate our results, we drilled into reports from four product teams in Microsoft. These groups confirmed that all the bugs reported are real bugs, 48% have been classified as high-priority bugs that would have otherwise resulted in service outages, and 70% have been fixed so far. Our evaluation also shows that TSVD introduces an acceptable overhead to the testing process — about 33% overhead for multi-threaded test cases while traditional techniques incur several times slowdowns.

TSVD is incorporated into the integrated build and test environment at Microsoft. Over last one year, 1,500+ software projects under active development at Microsoft have been tested with TSVD. An open-source version of TSVD is available at https://github.com/microsoft/TSVD.

2 Motivation and Background

2.1 Why Concurrency Testing in the Large?

The broader goal of this paper is to find concurrency errors during testing, ideally "in the small." Our specific goal here is to build a concurrency testing tool as part of integrated build and test environments that are becoming popular in software companies today [4, 15]. Such an environment provides a centralized code repository for all the different engineering groups. The code is incrementally built on every commit and on specific successful builds; a collection of servers automatically run unit and functional tests. It is not uncommon to have hundreds of servers dedicated to run tests from tens of thousands of modules written by many different groups.

Testing at this scale introduces key challenges: to minimize productivity loss, tools should have little/no false positives; to reduce overheads, the tools should have minimal runtime overhead and not require rerunning the test too many times; for wide adoption across a variety of product groups, the tool should be "push button" requiring little or no configuration and support a variety of programming models. TSVD discussed in this paper is one such tool.

2.2 Why Thread-Safety Violations?

A data race occurs when two threads concurrently access the same variable and at least one of these accesses is a write. Thread-safety violations are a generalization of data races to objects and data structures. Each class or library specifies, sometimes implicitly, a *thread-safety* contract that determines the set of functions that can be called concurrently by threads in the program. For instance, a dictionary implementation might allow two threads to concurrently perform lookups, while another implementation might disallow such calls if lookups can sometimes trigger a "rebalance" operation. In the extreme, a lock-free implementation might concurrently allow any pairs of calls.

A thread-safety violation occurs when the client fails to meet the thread-safety contract of a class or library, as discussed earlier in Figure 1. While the notion of thread-safety contract can be more general, in this paper we will assume that the methods of a data structure can be grouped into two sets — the read set and the write set, such that two concurrent methods violate the thread-safety contract if and only if at least one of them belongs to the write set. All the data structures we study in this paper have this property.

Thread-safety violations are easy to make but hard to find in code reviews. Sometimes programmers deliberately make such concurrent accesses due to their misunderstanding of the thread-safety contract. In fact, TSVD

is particularly inspired by a thread-safety violation similar to Figure 1 that caused a significant customer-facing service outage for days and took weeks to debug and fix.

Focusing on thread-safety violations solves some of the issues with data race detection. By definition, every TSV is a concurrency error while a data race can either be a concurrency error or a *benign* one [44] (that is nevertheless disallowed by modern language standards [11, 41]). Since thread-safety violations are defined at a larger granularity of objects and libraries, TSVD needs not to monitor every shared-memory access as data-race detectors do, which is one, but not all, of the reasons for the good performance of TSVD.

Note that in common parlance the term "thread safety" is defined ambiguously. Our notion here is different from high-level data race [46]. Similarly, Pradel and Gross [49] use "thread-safety" as a synonym to linearizability. Instead, throughout the paper, we will use TSVs to refer the violation of the thread-safety contract as defined above.

2.3 Why Not Happens-Before (HB) Analysis?

Happens-before [34] analysis is widely used in data-race and general concurrency-bug detection. In theory, once we capture all the synchronization/causal operations at run time and figure out all the happens-before relationships, we can accurately tell which operations can execute in parallel. Unfortunately, in practice, this is extremely challenging to conduct accurately and efficiently. We discuss the challenges in the context of TSVD below, which motivates the design of HB *inference*, instead of HB analysis, in TSVD. We will also design a variant of TSVD that conducts HB analysis (Section 3.5), which allows us to compare TSVD with different detection techniques.

General challenges First, it is difficult to capture all synchronization actions, as programs developed by thousands of engineering teams use numerous concurrent libraries, system calls, asynchronous data structures, volatile variables, and others for communication and coordination. Missing an HB-edge associated with such an action can result in false bug reports, while a spurious edge can hide true bugs.

Second, the cost of tracking and analyzing all synchronization actions, including many library calls, volatile/atomic variable accesses, and others, is high. Particularly, the analysis of HB relationships in programs that allow asynchronous concurrency, like all C# programs TSVD targets, is extremely memory and time consuming [26, 40, 53, 56].

Challenges for arbitrary task parallelism Previous work lowers the cost of dynamic HB tracking and analysis based on certain assumptions, such as (1) a fixed or small number of threads, or (2) having many more data accesses than synchronization operations [19, 48, 60], or (3) having task forks and task joins in a structured task parallel program restricted to series-parallel graphs only [17, 51, 52].

```
1  async Task<double> getSqrt(double x,
2                Dict<double, double> dict) {
3    if (dict.ContainsKey(x)) {
4      return dict[x];      // fetch from cache
5    } else {
6      Task<double> t = Task.Run(() =>
7          Math.sqrt(x));  // background work
8      double s = await t; // resume when done
9      dict.Add(x, s);     // save to cache
10     return s; }}
11
12 /* Assumes `a`, `b`, and `dict` exist */
13 Task<double> sqrtA = getSqrt(a, dict);
14 Task<double> sqrtB = getSqrt(b, dict);
15 Console.WriteLine(sqrtA.Result   // blocks
16                 + sqrtB.Result); // blocks
```

Figure 3. Example of task parallelism in C#. TSVs can occur due to concurrent accesses of `dict`.

These assumptions however do not hold for the programs TSVD targets: (1) these programs dynamically create many tasks, many more than the number of threads, and dispatch them to execute on concurrent background threads; (2) the number of data accesses no longer dominates that of synchronization operations, which include frequent task creations and joins , because data accesses are traced and checked at the granularity of thread-unsafe method calls and hence have a much smaller quantity in TSV detection; (3) these programs pass around task handles as first-class values and join with *any* task via its handle — forks and joins no longer follow series-parallel graphs. Such paradigms include not only traditional threading libraries like POSIX threads in C, Java's `Thread` class, and `System.Threading` in .NET, but also task-parallel libraries built on top of such primitives, like `java.util. concurrent` and `std::future` in C++.

For example, in .NET Task-Parallel Library (TPL), asynchronous work units are represented by `Task` objects. Tasks can be forked explicitly through APIs like `Task.Run`, or implicitly through data-parallel APIs like `Parallel.ForEach`, or through C# async/await paradigm. Any task can join with any other task using various mechanisms, such as explicit `Task.Wait` or implicitly blocking on a task's result via `Task.Result`. Task-level parallelism is hence unstructured.

An example To illustrate the arbitrary task parallelism and TSVs, Figure 3 shows a simple C# code snippet that

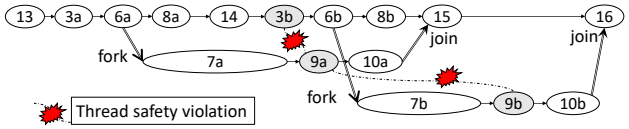


Figure 4. Happens-before graph for Figure 3, assuming neither `a` nor `b` is in the dict. The nodes show the executed line#; subscripts distinguish multiple executions of the same line.

```
1  OnCall(thread_id, obj_id, op_id){
2    check_for_trap(thread_id, obj_id, op_id)
3    if(should_delay(op_id)){
4       set_trap(thread_id, obj_id, op_id)
5       delay()
6       clear_trap(thread_id, obj_id, op_id) }}
```

Figure 5. The trap mechanism used by TSVD and its variants

uses explicit task creation and the async/await paradigm. Its core is the function `getSqrt`, which gets the square-root of its first argument by either retrieving from a table (lines 3–4) or computing from scratch (line 7).

Figure 4 illustrates the execution order of this code snippet when neither `a` nor `b` is in the lookup table, with the *happens-before* relation highlighted in arrows. As we can see, a call to `getSqrt` from lines 13 quickly leads to the fork of a `Task` by line 6 to carry out the computation (7a) in a concurrent background thread, while the `getSqrt` itself returns the spawned task immediately and moves on to the next call of `getSqrt` on line 14, denoted by 8a→ 14 in Figure 4. Later on, when the computation of the forked task completes, the *continuation* after the `await` statement is resumed from line 9 (i.e., 9a and 10a in Figure 4). Meanwhile, any attempt to retrieve the task's result like that on line 15 blocks until the corresponding continuation returns from `getSqrt` after line 10, illustrated by the Join arrow in Figure 4.

As shown in Figure 4, two instances of `dict.Add` (9a and 9b) are concurrent with each other and form a write-write TSV. Similarly, `dict.Add` (9a) and `dict.ContainsKey` (3b) form a read-write TSV. Detecting them via precise happens-before analysis in a large program is difficult due to challenges discussed earlier.

3 Algorithm

This section describes the design of TSVD along with its variants that represent alternative designs and will be used to compare with TSVD in our evaluation.

3.1 Overview

TSVD and its variants share the same *trap framework* shown in Figure 5, similar with that in DataCollider [14].

We assume a static analysis that identifies all the call sites to relevant data structures' methods whose thread-safety needs to be checked in the code (Section 4 will provide more details of such a list of methods used in TSVD). This analysis is straightforward and it reports call sites without checking their calling contexts (e.g., even if they are used within locks). We will refer to these call sites as *TSVD points*. We also assume an instrumentation framework that can instrument the program with calls to TSVD right at these TSVD points. The only interface to TSVD is the `OnCall` with arguments that show the thread making the call `thread_id`, the object being accessed `obj_id`, and the operation being performed `op_id`.

The trap mechanism works as follows. Consider a thread calling the `OnCall` method. On some calls chosen by the `should_delay` function at line 3, the thread sets a trap by registering the current method call in some global table at line 4. A trap is identified by the triple of the identifiers of the thread, the object, and the operation. Then, the thread calls the `delay` method to sleep for some period, during which the trap is "set."

Every other thread on entering `OnCall` checks if it conflicts with the traps currently registered at line 2. Formally, if a trap tid_1, obj_1, op_1 is set and another thread enters the method with the triple tid_2, obj_2, op_2, this pair results in a TSV iff $tid_1 \neq tid_2$, $obj_1 = obj_2$, and at least one of the two operations op_1 and op_2 is a write.

When such a conflict occurs, we have caught both threads "red handed" as they are at their respective program counters making the conflicting method calls to a common object. We can report the TSV with enough information gleaned from the current state for root causing this bug, such as the stack trace of the two threads. In theory, we could report other relevant information such as object content, but we have not found the need to do so in our experience with TSVD so far.

When the first thread wakes up from its delay, irrespective of whether a conflict was found or not, it clears the trap at line 6 and returns from the `OnCall` method.

TSVD and its variants differ in their answers to two key design questions about `should_delay`:

1. Where to inject delays? Which program locations are eligible for `should_delay` to suggest delay injections.
2. When to inject delays? In which program runs and at which dynamic instances of an eligible delay location, should `should_delay` suggest a delay injection.

When making the design decisions above, we have to keep the following goals and constraints in mind:

Runtime overheads — since the `should_delay` function is called inside *every* `OnCall` function, it is important to limit its complexity to reduce the runtime overhead. At the same time, if this function is not selective enough, we will inject too many delays, which in turn increases the runtime overhead.

Number of testing runs — if the `should_delay` function is too selective, then it can either miss bugs or require running the test too many times, increasing the testing-resource requirements.

Accuracy goals — no/little false positives are demanded; few false negatives without consuming too much resource is desired.

Complexity constraints — we would like the tool to work across code from many engineering teams. Specifically, the tool should handle a variety of synchronization mechanisms that developers can use to prevent TSVs. Moreover, sophisticated static analysis for arbitrary C# is challenging. For instance, a `ForEach.Parallel` block can sometimes be translated to a sequential loop when only one physical thread is available, and to a parallel loop otherwise. Determining this statically is a hard problem. For this paper, we restrict ourselves to dynamic techniques. As any dynamic analysis tool, TSVD cannot guarantee to find all TSVs in the program. However, by guaranteeing to not report any false errors, it guarantees "soundness of bugs" [23].

Next, we present the variants that take different design points in the design space in Figure 2. We start with two simple variants that occupy the top-left corner in Figure 2 and serves as baselines for TSVD.

3.2 DynamicRandom

The simplest algorithm takes every TSVD point as an eligible delay location (*where*), and injects delays at random moment (*when*). In other words, the call to `should_delay` return true with some (small) probability. Once deciding to delay a thread, the thread sleeps for a random amount of time.

3.3 StaticRandom

Previous work that uses delay injections to expose data races, such as DataCollider [14], observed that dynamic sampling is ineffective, as it tends to insert a lot more delays along hot paths and ignore cold paths where even more bugs may be hidden. To avoid redundantly introducing delays in hot paths and to focus on cold paths, DataCollider samples memory accesses "statically" — that is, static program locations are sampled uniformly irrespective of the number of times a particular location is sampled. We emulated this algorithm in our StaticRandom variant, where static call sites of relevant data structures' methods are sampled uniformly.

Comparing with DynamicRandom, *where* to inject delays does not change, still all the TSVD points, but *when* to inject delays differentiates code paths from hot paths.

3.4 TSVD

Our motivation for TSVD came after our initial experience implementing DynamicRandom and StaticRandom in our setting. While these variants were finding bugs, many of the bugs required running the same test tens of times, which was unacceptable in our setting. On investigating further, we found that these variants were injecting a large fraction of its delays at the "wrong" places — either when the program is executing in a single-threaded context, say when all other threads are waiting for external events, or when the program was accessing the data structure in a "safe" manner - say by holding the right lock. In both cases, a delay directly translates to an end-to-end slowdown, forcing us to drop the probability of delays to reduce the runtime overhead. This, in turn, resulted in missed bugs or a proportionally more number of runs to find true bugs.

TSVD is specifically designed to avoid these problems. As illustrated in Figure 2, TSVD explores a new design point. It uses *local* instrumentation and no synchronization modeling or happens-before analysis to reduce the runtime overhead, while identifying potential candidates for delay injection. By focusing on more likely candidates of thread-safety violations, it can achieve much better bug-exposing capability with no more, much less in fact, resource consumptions.

3.4.1 Where to inject delays?

At high level, TSVD uses lightweight analysis to dynamically maintain a set, called *trap set*, of *dangerous* pairs of program locations that can likely contribute to thread-safety violations and injects delays at only these locations.

During a testing run, the size of the trap set grows as TSVD discovers new dangerous pairs, or shrinks as TSVD prunes existing dangerous pairs.

TSVD identifies a pair of program locations as a dangerous pair if (1) they correspond to a near-miss TSV (§3.4.2) — intuitively, TSVD hopes to turn previous near misses into true conflicts, and (2) at least one of these two locations runs in a *concurrent phase* with multiple active threads of the program (§3.4.3).

A dangerous pair is pruned from the trap set if (1) TSVD infers a likely happen-before relationship between the two locations (§3.4.4) and hence the pair is unlikely to violate thread-safety, or (2) a violation is already found at the pair.

Finally, the delay injection is probabilistic—each program location *loc* in the trap set is associated with a

probability P_{loc} at which delay is injected at location loc. TSVD sets $P_{loc} = 1$ when a dangerous pair containing loc is added to the trap set, and the probability is decayed with time (§3.4.5). This probabilistic design can help avoid excessive overhead caused by too many delays for dangerous pairs that are actually not bugs.

We now describe the lightweight mechanisms TSVD uses to infer the key components required to make the above decisions. Note that, our evaluation section will show thorough parameter-sensitivity study of *all* the thresholds/parameters mentioned below.

3.4.2 Identifying near misses

Intuitively, if a pair of program locations never even get close to creating a thread-safety violation under the intensive testing environment, they might just never be able to create one due to underlying program semantics and synchronization operations. On the other hand, say an access[1] happens at program location loc_1 with the access triple tid_1, obj_1, op_1 at time t_1 and another access happens at program location loc_2 with the access triple tid_2, obj_2, op_2 at time t_2. TSVD considers the pair loc_1, loc_2 as a dangerous pair if $tid_1 \neq tid_2$, $obj_1 = obj_2$, at least one of op_1 and op_2 is a write, and $|t_1 - t_2| \leq \delta$ for some time threshold δ.

To make this judgment, TSVD maintains a list of accesses per object since δ time ago. For implementation simplicity, rather than storing this state in the object metadata, TSVD maintains a global hash table indexed by the object's hash-code containing this list of previous accesses. On each access, TSVD consults this list to identify dangerous pairs.

3.4.3 Inferring Concurrent Phases

Synchronization operations like `forks`, `joins`, `barriers`, and `locks` could lead to sequential execution phases during the execution of a concurrent program, like initialization phase, clean-up phase, join-after-fork phase, and so on, and a TSVD point inside such a sequential phase can never execute in parallel with another TSVD point.

TSVD infers concurrent execution phases by checking the execution history of TSVD points. Specifically, it maintains a global history buffer with a fixed number of most recently executed TSVD points. If the TSVD points in this buffer come from more than one thread, TSVD considers the execution to be inside a concurrent phase.

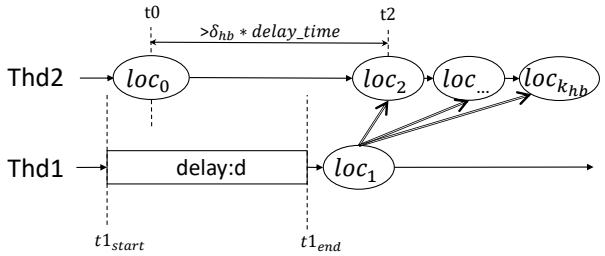


Figure 6. Happens-before inference in TSVD (the thick arrows indicate inferred happens-before relationship)

3.4.4 Inferring likely HB relationship

TSVD's identification of dangerous pairs based on near-misses and concurrent phases can potentially be incorrect. For instance, two accesses that are consistently protected by the same lock can happen close to each other. TSVD will fail to convert these near misses into true conflicts due to the dynamic happens-before (HB) relation between the two accesses. Rather than track synchronizations to calculate this HB relation, TSVD uses the lightweight technique below to infer likely HB relationships.

TSVD's dynamic HB inference works based on the following crucial observation. If loc_1 happens-before loc_2 in an execution, then a delay right before loc_1 will *cause* a proportional delay of loc_2. As shown in Figure 6, consider the case when loc_1 and loc_2 are consistently protected by a lock and loc_1 occurs first. When one injects a delay right before loc_1, this delay occurs when the thread executing loc_1 is still holding the lock. Thus, when the thread executing loc_2 tries to acquire this lock (which it needs to before accessing loc_2), it will block. By dynamically tracking this causality between program locations, TSVD infers *likely* HB relationships between them.

Tracking this causality works as follows. Apart from per-object history described above, TSVD additionally maintains a history of the most recent access made by every thread. Consider a delay d_1 injected right before access loc_1 that starts at $t1_{\text{start}}$ and ends at $t1_{\text{end}}$. At a subsequent access at loc_2 in a different thread Thd_2 happening at time $t2$, we will check when the previous access from Thd_2 occurred, denoted as $t0$, and infer a dynamic HB relationship between loc_1 and loc_2 if (1) there is a long gap between loc_2 and that previous access, $t2 - t0 \geq \delta_{hb} * delay\_time$, a causal delay threshold and (2) the long gap overlaps with the injected delay, $t0 \leq t1_{\text{end}}$. If multiple delays $\{d$ at $loc\}$ satisfy the inferring condition for Thd_2 and loc_2, we contribute the

[1] For the ease of discussion, we do not differentiate an access to an object and a thread-unsafe method call of an object in this section.

long gap in Thd_2 to the most recently finished delay d and infer the HB relationship accordingly. Considering the transitivity of happens-before relationship, the next k_{hb} accesses in thread Thd_2 are also considered as likely happens-after loc_1. Again, k_{hb} is a tunable parameter and we will evaluate its sensitivity in the evaluation section.

3.4.5 Delay Decaying

Of course, all the inferences above, including the near-miss tracking, the concurrent phase inference, and the likely HB-relationship inference, still do not guarantee to prune out all non-TSV pairs. Consequently, for every dangerous pair $\{p_1, p_2\}$ in the dangerous list, every delay injection at either p_1 or p_2 that fails to make p_1 and p_2 execute in parallel with the same object accessed will cause TSVD to decay the probability of future delay injection at p_1 and p_2. When the probability of a location loc drops to 0, all its related pairs are removed from the trap set.

3.4.6 When to inject delays?

Different from previous work that separates delay planning and delay injection into different runs [47, 58] and/or using many runs to gradually inject delays to all planned locations [14, 47, 58], TSVD conducts its delay planning *and* injections in the *same* run, making the best use of testing resources.

Planning & injection in the same run Once TSVD identifies a dangerous pair of program locations (that nearly missed each other), TSVD does not wait until the next run to expose a potential thread-safety violation between them. The rationale is that most instructions execute more than once and hence most bugs have multiple chances to manifest. This is confirmed by our experimental results in Section 5 showing that TSVD hits the same bug multiple times. Consequently, TSVD can still try to expose a bug after an initial near-miss. Specifically, whenever a program location loc is going to be executed, TSVD checks if loc is part of the current trap set. If it is, TSVD will insert a delay at loc with probability P_{loc}.

Multiple testing runs The above delay injection scheme would miss a bug if the near-miss situation observed by TSVD was actually the only chance to expose the bug; i.e., if the dangerous pair of locations never run together after the pair is identified. When the testing resource allows, TSVD runs the program for another time, carrying the knowledge obtained from the first run, to help discover these bugs.

During the first run, TSVD records its trap set in a persistent *trap file*. At the end of the run, the trap file contains the final set of dangerous pairs. At the beginning of the second run, TSVD initializes its trap set from the file, allowing it to inject delays at pairs even at their first occurrences.

Parallel delay injection There are clear trade-off among injecting only one delay or multiple delays throughout a run, and allowing only one thread or multiple threads to be delayed at a time: when multiple delays are injected, they could cancel each other's effect; when few delays are injected, we will need too many runs to test all possible delay situations.

TSVD decides to conduct delay injection in an aggressive way — delays are inserted strictly following the dangerous pair list and the decay-probability scheme, regardless of whether another thread is already blocked. Note that, although this aggressive strategy can cause some injected delays to overlap with each other, the overlaps will only be partial and are unlikely to cancel each other out. This is because different threads will be unblocked at different time and our decay-probability scheme also helps avoid too many threads blocked at the same time. On the other hand, an alternative design that strictly avoids delay overlaps would lead to too few delay injections and hence hurts our chance of exposing bugs within the tight testing budget, as we will demonstrate in the evaluation section.

3.5 TSVD with happens-before analysis

To compare TSVD with existing techniques that rely on HB analysis, we designed a variant of TSVD, referred to as TSVD_{HB}, that follows the approach of RaceFuzzer [58]. Note that, TSVD_{HB} uses a few optimizations to speed up traditional HB analysis following the type of bugs it targets.

Where to inject delays? Just like that in TSVD, TSVD_{HB} dynamically maintains a trap set, with every program location appearing in the trap set a delay-injection candidate. At run time, TSVD_{HB} adds a pair of program locations $\{loc_1, loc_2\}$ into the trap set, if the corresponding accesses tid_1, obj_1, op_1 and tid_1, obj_1, op_1 satisfy that $tid_1 \neq tid_2$, $obj_1 = obj_2$, op_1 and op_2 conflict, and most importantly they do not have happens-before relationship with each other.

TSVD_{HB} monitors synchronization operations, such as locks, forks, and joins, and uses vector clocks [18] to compute the happens-before relation between accesses.

TSVD_{HB} optimizes traditional race detection to speed up its HB analysis following the observation that, different from traditional race detection, where execution traces contain few synchronization operations but many memory accesses, TSVD_{HB} faces many synchronization operations (the large number of `async/await` in C# cloud service modules) yet relatively few accesses (i.e., TSVD points — invocations of thread-unsafe methods of some data structures).

The first optimization is to increase local timestamps at accesses (TSVD points), which are relatively few at run time, but not at synchronization operations, which are relatively frequent, opposite from traditional race detection [19].

The second optimization is that TSVD_{HB} uses immutable data structures—AVL tree-maps—to represent vector clocks, unlike traditional implementations which use mutable arrays or hashtables. For vector clocks with n components, a message-send (or other similar types of synchronization) event requires an On-time/memory copy with traditional mutable tables, whereas immutable clocks can be passed by reference in $O1$-time. On the flip side, TSVD_{HB} requires $O\log n$ time/memory to perform increment operations, whereas mutable clocks can be updated in-place in $O1$ time. Fortunately, TSVD_{HB} restricts increment operations to infrequent TSVD points only. Finally, message-receive (or other similar type of synchronization) events require On time with both data structures, since an element-wise max must be computed. Fortunately, in the relatively common case where tasks fork and join without going through any TSVD points, the vector clocks associated with the join-message and with the receiving thread are exactly the same object; such an operation can be optimized to $O1$ by simply checking for reference equality.

When to inject delays Unlike RaceFuzzer, which uses many runs that are each targeted at a specific pair of potential data races, TSVD_{HB} aims to expose bugs in a small number of testing runs just like TSVD. Consequently, just like that TSVD, TSVD_{HB} injects delays in the same run as the happens-before analysis and trap-set fill-up; it can delay multiple threads simultaneously. When one testing run is finished, those remaining pairs in the trap set are recorded and fed to the testing run, if there is one. Like TSVD, TSVD_{HB} uses a probability decay to prune possibly spurious dangerous pairs.

4 Implementation

We have implemented TSVD for .NET applications (e.g., C# and F#). It has two key components: (1) TSVD runtime library that implements the core algorithm, and (2) TSVD instrumenter that, given a .NET binary, automatically incorporates TSVD runtime library into it by static binary instrumentation. We chose static instrumentation of application binary as this enables us to use TSVD instrumented binaries with unmodified, existing test pipeline (unlike framework instrumentation [65], which requires updating test pipeline with instrumented Common Language Runtime (CLR) or dynamic instrumentation [2], which requires modification of test environment). While our implementation is specific to .NET, we believe the techniques underlying TSVD can be easily implemented for other languages and runtimes.

```
1  List<int> listObject = new List<int>();
2  listObject.Add(15);
```
(a) Original code

```
1  List<int> listObject = new List<int>();
2  int op_id = GetOpId();
3  Proxy_123(listObject, 15, op_id);
```
(b) Instrumented code

```
1  void Proxy_123(Object obj,int x,int op_id) {
2    var thread_id = GetCurrentThreadId();
3    var obj_id = obj.GetHashCode();
4    OnCall(thread_id, obj_id, op_id);
5    obj.Add(x); }
```
(c) Proxy method

Figure 7. TSVD instrumentation

TSVD Instrumenter This takes a list of application binaries and a list of target thread-unsafe APIs. In the current prototype of TSVD, we focus on 14 C# thread-unsafe classes (e.g., List, Dictionary) and manually identified 59 of their APIs to be write-APIs and 64 as read-APIs. This process is straightforward for these data-structure classes and has a small cost that is easily amortized by reusing this list for all C# program testing. It then instruments the binaries by replacing each thread-unsafe API call with an automatically generated proxy call, as shown in Figure 7. The proxy wraps the original call; but, before making the original call, it calls the `OnCall` method (Figure 5) implemented in TSVD runtime library. TSVD uses Mono.Cecil [28] to do the instrumentation.

The TSVD instrumenter can be used as a command line tool or with Visual Studio (as a post-build step). To allow a developer to use TSVD without additional configuration, TSVD comes with an extensible list of thread-unsafe APIs in .NET standard libraries. The list also includes information about whether the APIs are read- or write-APIs.

TSVD Runtime This implements the `OnCall` method, which executes the core TSVD algorithm (Figure 5). In addition, (1) it logs runtime contexts when a bug is detected. The contexts include bug locations (e.g., method name and source code line number) and stack traces of racing threads. (2) It tracks total delay injected per thread and per request so that one can limit the maximum delay per thread or request. This helps in avoiding test timeouts. (3) It tracks test coverage of instrumented APIs. (4) Finally, it updates signatures of instrumented signed binaries.

While using TSVD in Microsoft, we observed that many asynchronous programs using async/await run synchronously in test settings, and therefore, TSVD cannot find thread-safety bugs that could manifest in production runs when the programs do run asynchronously. We found that the underlying reason is a .NET runtime optimization that executes fast async functions synchronously. This affects many tests that replace async I/O calls with fast mock implementations. To address this, TSVD instruments async/await code to force all async functions, slow or fast, to run asynchronously. Note that, in our experiments (Section 5), this strategy will be applied to not only TSVD but also all the techniques that will be compared with TSVD.

An open-source version of TSVD is available at https://github.com/microsoft/TSVD.

5 Evaluation

5.1 Methodology

Benchmarks For our evaluation, we collected approximately $43K$ software modules from 1,657 projects under active development at Microsoft. We call this the *Large* benchmark. These modules are regularly tested in an integrated build and test environment. Each module contains a collection of unit tests as well as all binaries required to execute the tests. Together, they contain a total of around $122K$ multi-threaded unit tests and $89K$ program binaries. To test a module with TSVD, we instrument its binaries and run its existing unit tests. 2% of the modules also included long-running end-to-end tests, that we also run. Of all the software modules that were available in the integrated environment we selected those modules which a) were written in a .NET language such as C# or F# as TSVD is currently implemented for .NET programs, b) passed all their tests over the past 2 builds, ensuring that these modules are stable and well-tested, and c) used multithreading primitives or C# `async`/`await` mechanisms at least once. We will use this to evaluate the bug-exposing capability and performance of TSVD.

To evaluate TSVD against alternate designs and parameter settings, we sampled 1000 modules randomly from the Large benchmark. We call this the *Small* benchmark. Together, this suite consists of 3350 multi-threaded unit tests.

Experiment Setting We run the small benchmark suite on a small server (S1), with Intel(R) Xeon(R) E5-1620 CPU, 16G Memory, and 1T SSD. Since all the software modules under test are independent with each other, we run 10 modules at a time on S1. We run the big suite on a large server (S2), with Intel(R) Xeon(R) E5-2650 CPU, 128G memory, and 6T SSD. Both S1 and S2 use Windows10 operating system.

Test Targets	
# of projects	1,657
# of test modules	$\approx 43K$
# of binaries	$\approx 89K$
# of tests	$\approx 122K$
Bugs found	
# of unique bugs (location pairs)	1,134
# of unique bug locations	1,180
# of unique stack trace pairs	21,013
% of projects with bugs	9.8%
% of modules with bugs	1.9%
Bugs properties	
% of read-write bugs	48%
% of same location bugs	34%
% of bugs in `async` code	70%
Avg. (Median) occurrence of a bug location	36(4)
Avg. (Median) # stack trace pairs/bug	18.5(3)
Avg. Stack depth	9.1
% of `Dictionary` bugs	55%
% of `List` bugs	37%

Table 1. Summary of bugs found by TSVD tools.

5.2 Overall Results

Bugs Found Over last few months, we have been using variants of TSVD on the Large benchmark. Table 1 summarizes the results. Overall we found 1,134 bugs, uniquely identified by the pair of static program locations participating in the TSV, involving 1,180 unique static program locations.[2] 1,009 of these bugs were found using the core TSVD we present in this paper; while the remaining bugs came from our previous attempts of using variants of TSVD, namely DynamicRandom and TSVD_{HB}.

For each bug, TSVD produces a pair of stack traces that shows two threads actively participating in a thread-safety violation. Note that the same bug can manifest at multiple stack trace pairs. On the other hand, multiple bugs in a high-level data structure might show up as a conflict in a low-level data structure with the same program-location pair. Overall, we found the above bugs in 21,013 different stack trace pairs (i.e., 18.5 stack trace pairs per bug). Due to our inability to identify whether two different stack-trace pairs correspond to the same "bug" or not, we will use the conservative program-location pair for unique bug counts.

On average, we found at least one bug in 9.8% of the projects and 1.9% of the modules. As TSVD find these

[2] Note that unsynchronized accesses to n locations can theoretically result in n^2 location pairs. However, our reported bug count (i.e., 1,134 location-pairs) is not bloated by this phenomenon (for 1,180 locations).

bugs while running existing tests provided by developers, TSVD does not report false error reports.

Bug Validation We contacted four product teams in Microsoft to validate *all* bugs TSVD found in their software (total 80). *All* of the bugs were confirmed as *true* bugs. Of these, 77 were *new* bugs previously not known to the developers and the remaining 3 were concurrently found by other means. We are still in the process of contacting other teams.

Bug Quality 38 of these bugs were confirmed as "high priority." This internal classification of bugs signifies that these bugs, if manifested in production, would result in a customer-facing outage and thus needs to be fixed immediately. These bugs were immediately fixed by the developers. 22 bugs were of "moderate priority" of which 18 have been fixed so far. The remaining 20 bugs were in non-critical code, such as the test driver or mock code written for unit testing. While these bugs are in lower priority, they nevertheless need to be fixed for preventing nondeterministic test failures.

Actionable Reports Overall, the developers found the error reports to be sufficiently actionable. The stack traces of the two conflicting threads provide enough detail to root cause the problem. In many cases, the fix involves introducing additional synchronization or replacing the data-structure with a thread-safe version of the data structure.

Apart from the error reports, TSVD also reports statistics on the instrumentation points that were hit during the test in any context and in a concurrent context. One team found these "coverage" statistics to be very useful and identified a few blind spots in their testing, such as critical parts only called in sequential contexts during unit testing etc.

Bug Nontriviality Note that all these software teams employ their usual production test pipelines to uncover various types of bugs. The bugs TSVD found had been in their systems for many months but remained undetected by the test pipelines. To further confirm that the bugs TSVD finds are not easy-to-find, we worked closely with one of the teams to track and compare the bugs found by TSVD and those found by their usual test pipeline. During a 3-months period, TSVD found 15 thread-safety bugs (all confirmed as "priority one" and immediately fixed) in their code, none of which was found by the test pipeline during the same period.

Bug Characteristics 49% of the 1,134 bugs we found were due to concurrent read-write conflicts on thread-unsafe objects. An interesting root cause of such bugs is locking only writes, but not reads, to a thread unsafe object. 70% of the bugs were triggered by code using async/await or Task Parallel Library. Many of these bugs were found benefiting from TSVD's instrumentation design that forces all async functions, slow or fast, to run asynchronously (§4). 34% of the bugs manifest due to two threads executing the same operation. Average stack depth of the bug location from test code is 9.1, indicating that many of the bugs appear deep inside production code. Note that the same bug location can appear in multiple bugs (counted as location-pairs) and the same bug can appear multiple times during the same test run. In our experiments, a bug location appeared 35.6 times on average (median 4 times). Despite repeated appearances, the buggy locations remained undetected with existing test techniques in Microsoft. More than half of the bugs involve the `Dictionary` data structure. Note that .NET's standard library includes thread-safe `ConcurrentDictionary`. Yet, our results show that developers often use thread-unsafe `Dictionary` in multi-threaded applications, without necessary synchronization primitives. Talking to developers, we found one interesting cause behind such incorrect usage: erroneously assuming that two writes to a dictionary would be thread-safe as long as the keys are different.

	# bug				
	Total	Run1	Run2	overhead	# delay
DataCollider	25	22	3	378%	77402
DynamicRandom	13	6	7	178%	31456
TSVD_{HB}	41	25	16	310%	3328
TSVD	53	42	11	33%	22632

Table 2. Comparing TSVD with other detection techniques.

5.3 Comparison with other detection techniques

We compare TSVD with DataCollider and other detection techniques discussed in Section 3 using the 1000-module Small benchmark suite. We compare them in terms of the number of bugs found in two rounds of run and the overhead , which is computed based on the additional amount of time imposed by a tool upon uninstrumented baseline testing runs. Results are summarized in Table 2 (DynamicRandom uses 0.05 probability in its delay injection).

Number of bugs found TSVD found the most number of bugs: 42 bugs in the first round, and 11 additional bugs in the second run. Each of these 11 bugs contains a TSVD point that only executes once during the unit testing, and hence cannot be exposed after the *near-miss* at the first run. DynamicRandom and DataCollider detected significantly smaller number of bugs as they have a small probability (0.05 for DynamicRandom) injecting delay at a TSVD point. They can potentially find more bugs

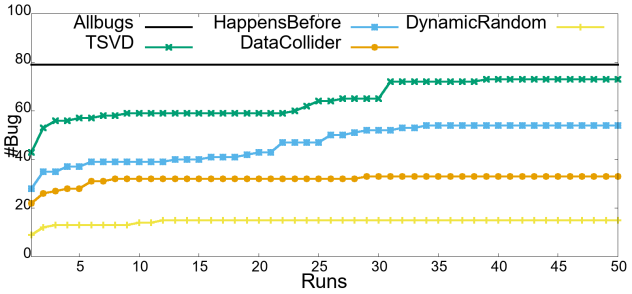


Figure 8. Number of bugs found after more runs

if run additional rounds, but it would take many rounds for them to catch the bug-reporting capability of TSVD. TSVD outperforms TSVD_{HB} because TSVD_{HB} can both miss Happens-Before edges and add spurious ones, just like other dynamic Happens-Before-based techniques [47]. The huge analysis overhead of TSVD_{HB} also interferes more with bug exposing at run time.

We observed a similar trend in a larger benchmark of 8000 modules. In two rounds of run, TSVD found 256 bugs (227 in the first round), while TSVD_{HB} found 192 bugs and DynamicRandom found 79 bugs.

One interesting observation from Table 2 is that TSVD found more bugs in its first round alone than other techniques found in two rounds. Moreover, TSVD's first round found about 80% of all bugs found by all tools. Both these observations hold for our full benchmark suite as well. This justifies our design of not separating delay planning and injection into different runs. It also provides a more resource-conserving option for using TSVD: while testing under severe resource constraints, one can choose to run TSVD for only one round and still capture vast majority of the bugs.

Note that even though TSVD_{HB} found more *new bugs* than TSVD in the second round in Table 2, they both found comparable numbers of *total bugs*: 41 and 53 respectively.

Performance comparison As we can see in Table 2, TSVD not only finds the most bugs, but also with the least overhead. The performance advantage of TSVD over all other variants comes from two aspects. Comparing with DynamicRandom, TSVD injects most delays when program running with multiple threads. DynamicRandom injects many delays in sequential phase which never detects bugs but also introduces huge overhead. Comparing with TSVD_{HB}, TSVD and TSVD_{HB} both inject delay in the concurrent phase, yet TSVD_{HB} spends much time in analyzing the happens-before relationship at run time, which introduces 270% run-time overhead on average in our experiments.

Number of bugs after more runs Given the non-deterministic nature of TSVs and the probabilistic nature of some of these techniques, we ran each of these techniques 50 times with the Small benchmark suite to see how many bugs can be found at the end. As we can see in Figure 8, even after many more runs, the bug detection advantage of TSVD still holds over the other techniques. All the four techniques altogether discovered 79 bugs at the end, with 73 of them discovered by TSVD, 54 of them discovered by TSVD_{HB}, and much fewer by the other two. Furthermore, most of these bugs (close to 70%) can indeed get caught with just two runs of TSVD! We will use these results to help understand the false negatives of TSVD next.

False positives of TSVD TSVD does not report any false positives. Every bug reported by TSVD is a true bug — a thread-safety violation can and is already triggered by TSVD.

False negatives of TSVD Given the non-deterministic nature of TSVs and the tight testing budget TSVD faces, TSVD inevitably has false negatives. Since it is impractical to know the ground truth about how many TSVs exist in our benchmarks[3], we use the 79 bugs discovered by 4 tools at the end of accumulated *200* runs in Figure 8 as our best-effort ground truth. We analyzed the 26 bugs that were missed by TSVD in its merely *2* testing runs and put them into three categories:

(1) Near-miss false negatives. For 19 bugs, the two racing operations execute close to each other only under rare schedules (e.g., a resource usage and a resource de-allocation). In most runs, there is a long time gap between them, like more than a few seconds. Consequently, the near-miss tracking in TSVD did not identify them as a dangerous pair and hence injected no delays to expose the bug in two runs. After running TSVD for 50 runs, 15 out of these 19 bugs were caught.

(2) Happens-before inference false negatives. For 2 bugs, TSVD's HB inference mistakenly treats two concurrent operations as happens-before ordered, and hence lost the opportunity of triggering the bug. Even after 50 runs, these 2 bugs still cannot be detected by TSVD.

(3) Delay-injection false negatives. For 5 bugs, the timing during the two testing runs happens to be that the injected delay was not long enough to trigger the bug. After running a couple of more runs, these bugs were all discovered by TSVD.

Finally, given the design goal of TSVD, it will naturally miss timing bugs that are not TSVs. For example, a recent study of real-world incidents in Azure software

[3]Note that, before the use of TSVD, few TSVs were known in the code base tested by TSVD, which is exactly the motivation behind TSVD.

[37], a similar set of software projects that TSVD experiments target, showed that many more incidents are caused by timing bugs than that in traditional single-machine systems and about half of these timing bugs are related to concurrent accesses to persistent data. The current prototype of TSVD only focuses on in-memory data structures and hence cannot detect these persistent-data bugs. Future research can extend the idea of TSVD to detect other types of timing bugs.

5.4 Evaluating TSVD parameters

We now use the 1000-module small benchmark suite for parameter-sensitivity experiments.

Probabilistic nature of TSVD Since TSVD injects delays probabilistically, one may wonder how much the results of TSVD may vary across different tries. Figure 9(a) shows the number of detected bugs and the overhead of TSVD across 12 tries. In every try, TSVD uses the same default parameter setting and is applied to two consecutive runs, just like that in Table 2. As we can see in the figure, the results indeed vary across tries, but only slightly: the number of detected bugs varies from 52 to 54 and the overhead varies between 30% and 35%, with a median of 53 bugs and 33% overhead.

Near-miss tracking parameters TSVD uses two parameters for tracking near-misses: the number N_{nm} of recent accesses that TSVD keeps for each object and the physical time window T_{nm} that TSVD considers two accesses as a near miss. As shown in Figure 9(b) and (c) (note log-scale of the x-axis), roughly speaking, both bug count and overhead increase with both parameters. Using too small a value (e.g., $N_{nm} = 1$ or $T_{nm} = 1ms$) misses many bugs because of not identifying dangerous pairs. TSVD's default values of $N_{nm} = 5$ and $T_{nm} = 100ms$ finds almost all the bugs, with small overhead. Larger values do not significantly increase bug count, but increases overhead (especially for N_{nm}).

HB inference parameters TSVD uses two parameters for HB inference: a causal-delay blocking threshold δ_{hb} and an inference window of k_{hb} accesses (Section 3.4.4). Figure 9(d) shows the effect of varying δ_{hb} from 0 to 0.8. A value too small like 0 infers many non-existing HB relationships, and hence misses many bugs; a larger value has stricter constraints in inferring HB relationship. As shown, the overheads and bug counts do not change much beyond TSVD's default value of 0.5. Figure 9(e) shows the effect of varying k_{hb}. A larger value translates to more HB relationship, reducing the number of dangerous pairs, and eventually the number of bugs and overhead. Too large a value drastically reduces the bug count. TSVD's default value of $k_{hb} = 5$ gives a sweet spot between bug count and overhead.

	# bug			
	Total	Run1	Run2	overhead
TSVD	53	42	11	33%
No HB-inference	45	36	9	84%
No windowing in near-miss	46	35	11	143%
No concurrent phase detection	54	42	12	61%

Table 3. Removing one technique at a time from TSVD

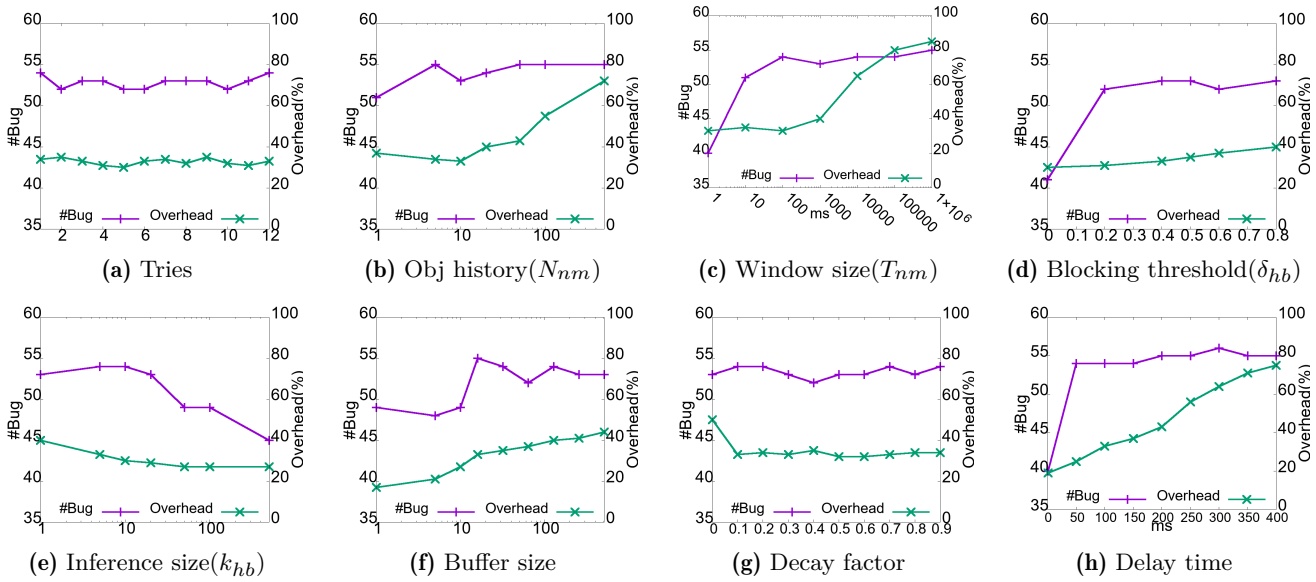
Buffer size of concurrent phase detection Figure 9(f) indicates that the overhead and the number of detected bugs both grow with the size of the global history buffer — with a large buffer, TSVD may mistakenly treat sequential operations as concurrent and generate dangerous pairs that do not lead to any bugs; yet, with a small buffer, real concurrent operations can be mistakenly treated as sequential. TSVD uses a default size of 16 that gives a good trade-off between overhead and bug count.

Delay injection Figure 9(g) shows the impact of decaying delay injection probability at each TSVD point. A particularly bad configuration is when the factor is 0, meaning that TSVD injects delay in all occurrences of these TSVD points without any decay. This configuration would introduce too much overhead: for 3% modules, the overhead of zero decay was more than 100% (maximum overhead we observed was 6600% for one module!). These modules use TSVD points repeatedly and frequently (e.g., they are inside loops). Figure 9(h) shows the impact of the amount of delay TSVD injects at one trap point. As expected, longer delay increases runtime overhead, but also creates more opportunities for conflicting operations to overlap in time. TSVD uses 100ms as the default value.

Effectiveness of various TSVD techniques Table 3 shows how TSVD performs when one of its core components is disabled at a time. (In "No windowing", TSVD treats conflicting accesses by different threads *in the entire history* as near-misses.) As the results show, HB-inferencing and windowing are the most crucial techniques for finding bugs—without them bug counts drop from 53 to 45 and 46. Windowing is the most important factor in reducing overhead—without it, overhead increases from 33% to 143%. Overall, all the techniques are needed for TSVD's effectiveness.

5.5 TSVD CPU/Memory Consumption

To understand the detailed resource consumption of TSVD, we ran every unit test in the Small benchmark suite with and without TSVD, while recording the largest memory usage and average CPU usage in each run. Across all the unit tests, the median increase on maximum memory is 17% and the median increase on average



(a) Tries **(b)** Obj history(N_{nm}) **(c)** Window size(T_{nm}) **(d)** Blocking threshold(δ_{hb})

(e) Inference size(k_{hb}) **(f)** Buffer size **(g)** Decay factor **(h)** Delay time

Figure 9. Sensitivity analysis of various parameters of TSVD.

```
1  Async Task<T> ClientStatusUpdate(int clientID,
2          Status s){
3          ...
4          GlobalStatus[clientID] = s;
5  }
```

(a) Device Manager

```
1  Parallel.ForEach(hostlist,
2       delegate(string host){
3     ConfigLevel cl = GetConfigLevel(host);
4     configureCache[host] = cl;
5     ...
6  } );
```

(b) Network Validation

Figure 10. Examples

CPU utility is 82%. The extra memory is mainly used for keeping the near-miss pairs and the access history of every thread-unsafe object. The extra CPU utility is mainly due to our instrumentation that forces all async functions to run asynchronously, as explained in Section 4. In comparison, without TSVD, the .NET optimization makes many async functions run synchronously, using much fewer cores.

5.6 Examples of bugs found by TSVD

Device Manager As shown in Figure 10(a), a device manager uses a Dictionary `GlobalStatus` to maintain the status of every client, where client ID is the key and the client status is the value. The device manager has a thread responsible for listening from multiple clients. Whenever the manager receives a message from a client, this listening thread will create an asynchronous task shown in Figure 10(a) to update the status of the corresponding client, and continue its listening. A concurrent write violation on the Dictionary class could happen when two clients send messages at similar time, which could then cause two concurrent execution of Line 4 in the figure and hence two concurrent Dictionary-set operations. As a result, the `GlobalStatus` Dictionary could get silently corrupted.

Network Validation Figure 10(b) shows another example of concurrent- write violation on a Dictionary class, which has a very different code pattern from the first example. When a network service starts up, a validator needs to verify the configuration information of every host, which involves reading a host's configuration information (Line 3) and storing it to a `configureCache` Dictionary for further verification. To speed up this process, the validator uses a `Parallel.ForEach` primitive (Line 1) to parallelize the validation for different hosts. The `Parallel.ForEach` automatically generates multiple concurrent threads and when some of these threads concurrently execute Line 4, a concurrent-write violation occurs to corrupt `configureCache`.

Production-Incident Example This is a bug about two threads trying to sort a list at the same time in a production run. The sorting result of an unprotected list is undetermined when two threads are doing that concurrently. This undetermined behavior propagated and

Project	LoC	# tests	# run	# TSV	overhead
ApplicationInsights [3]	67.5K	934	2	1	15.31%
DataTimeExtention [12]	3.2K	169	1	3	18.51%
FluentAssertion [20]	78.3K	3076	1	2	8.89%
K8s-client [33]	332.3K	76	2	1	11.79%
Radical [50]	96.9K	965	1	3	1552.13%
Sequolocity [59]	6.6K	209	1	3	2.97%
Stastd [62]	2.5K	34	2	1	9.72%
System.Linq.Dynamic [63]	1.2K	7	1	1	41.39%
Thunderstruck [64]	1.1K	52	1	2	3.33%

Table 4. TSVD results on open source projects.

finally caused the service to go down for several hours. TSVD can reproduce this bug without any prior knowledge and help the developers reduce the debugging effect.

5.7 TSVD on Open Source Projects

To evaluate whether TSVD can be used beyond Microsoft, we applied TSVD to 9 open source C# projects (Table 4). Without any existing C# bug benchmark suite, we searched Github using "race condition" keyword and identified these 9 that contain (1) confirmed bug reports about TSVs on standard library APIs and (2) developer-written test cases.

Using exactly the same parameters as before, TSVD successfully detects and triggers all the TSVs under test in at most 2 runs. For all but 3 projects, TSVD detects these bugs using the *original* test cases released with the buggy software — if TSVD was used, these bugs would have been caught before code release. For Thunderstruck, TSVD detects one TSV that was not part of the original bug report.

We also evaluated the performance of TSVD on *all* the test cases. The overhead is mostly <20%, consistent with earlier performance results. Two projects incur large overhead, as they have many short-running tests (<1 ms). The average slowdown for their tests is actually less than 400ms.

6 Related Work

Data Race Detection Since the tolerance for false error reports is low in our context, we do not focus on static data-race-detection techniques [13, 16, 36, 66]. TSVD is closely related to dynamic active data-race detection techniques [47, 58], as discussed in Section 1 and Figure 2.

Dynamic passive detection techniques [19, 48, 57, 60, 65] perform happens-before analysis, lock-set analysis, or a combination of the two to predict whether a data race could have manifested. Recent work [27, 32, 61] infer more data races by generalizing beyond executions that are happens-before equivalent to the current execution. These techniques pay the runtime cost of monitoring synchronization operations and also suffer from false positives [30, 45, 58]. Several techniques have used sampling to reduce the runtime overhead [5, 31, 42], but like DataCollider [14], they need to repeatedly run the test many times to expose bugs.

Another type of passive detection tools like AVIO [38] and Bugaboo [39] catches concurrency bugs *when* they manifest at run time. Since they do not *predict* or *expose* bugs that have not manifested yet, which TSVD does, they do not need to analyze happens-before relation but are also fundamentally unsuitable for the build and test environment TSVD targets.

Race detection was studied for asynchronous event-driven programs such as Android apps and Web applications [25, 26, 40, 53]. There, the key challenge is to model and analyze the causality between asynchronous requests and responses, which incurs huge run-time slowdowns. TSVD instead automatically infers happens-before relationships. While not the focus of this paper, techniques behind TSVD can work for applications targeted by these works.

Systematic Testing Systematic testing techniques discover concurrency errors by driving the program towards different possible interleavings [6, 7, 21, 22, 24, 35, 43]. These techniques are either guided by the need to cover all interleavings within some bound [21, 22, 35, 43], or a coverage notion [6, 24], or provide a probabilistic guarantee of finding bugs [7]. While these techniques can find general concurrency bugs, they are not designed to discover most bugs in a small number of runs. Instead, TSVD is specifically designed for finding TSVs within the first few runs and without paying the cost of controlling the thread scheduler.

Generating Unit Tests Tools were proposed to synthesize unit tests to help expose concurrency bugs inside library functions [9, 49, 54, 55]. Given multi-threaded libraries that are expected to allow safe concurrent invocations, these tools synthesize concurrent method-call sequences to help expose bugs inside the library implementation. They are orthogonal to TSVD: TSVD focuses on improving the efficiency of exposing thread-safety violation bugs through existing unit tests of software that uses thread-unsafe libraries.

Timing hypothesis Snorlax [29] reproduced and diagnosed in-production concurrency bugs leveraging a coarse interleaving hypothesis. Snorlax experiments showed that the time elapsed between events leading to concurrency bugs ranges between 154 and 3505 micro-seconds, based on which Snorlax could reproduce concurrency bugs without fine granularity monitoring and recording. This coarse-interleaving hypothesis and TSVD's near-miss tracking look at different aspects of concurrency bug's timing window characteristics — Snorlax believes the

timing window is not as small as people used to think, and TSVD believes conflicting accesses in small windows are more likely to lead to real bugs — and leverage the characteristics in different ways.

Causality inference Past work has used run-time trace analysis to infer happens-before relationship among message sending/receiving operations or distributed system tasks in the context of system-performance analysis [1, 10] and network-dependency analysis [8]. Due to the different usage scenarios, the exact inference algorithms differ between TSVD and these previous tools. Previous tools all require a large number of un-perturbed system traces, and statistically infer happens-before relationship based on whether two operations never flip execution order [10] or always execute with a nearly constant physical-time gap in between [8]. Different from previous tools, TSVD works in the unique testing environment where the system trace contains much perturbation introduced by TSVD; TSVD also faces the unique goal of finding bugs in a small number of runs, and hence cannot wait until a large number of traces become available. Consequently, TSVD's happens-before inference is uniquely designed based on observing delays in each testing run.

7 Conclusion

This paper presents a new thread-safety violation detection technique TSVD. Being part of an integrated build and test environment, TSVD provides a push-button no-false-positive bug detection for .NET programs that use complex multi-threaded and asynchronous programming models with a wide variety of synchronization mechanisms. TSVD provides a starting point for exploring the wide design space of active testing and resource-conscious delay injection design. Future research can further explore how to balance bug exposing capability, accuracy, and cost.

Acknowledgments

We would like to thank Lenin Ravindranath Sivalingam, for his help in building the first prototype, Suresh Thummalapenta, for his help with our experiments, our shepherd and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grants funding CNS-1563956 and CNS-1514256.

References

[1] Marcos K Aguilera, Jeffrey C Mogul, Janet L Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance debugging for distributed systems of black boxes. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 74–89.

[2] Aleksandr Mikunov. [n. d.]. Rewrite MSIL Code on the Fly with the .NET Framework Profiling API. https://www.cnblogs.com/WCFGROUP/p/5136703.html.

[3] ApplicationInsights. [n. d.]. Broadcast processor is dropping telemetry due to race condition. https://github.com/Microsoft/ApplicationInsights-dotnet/issues/994.

[4] Bazel. [n. d.]. Bazel: Build and test software of any size, quickly and reliably. https://bazel.build/.

[5] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada*.

[6] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. 2005. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP, Chicago, IL, USA*.

[7] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. 167–178.

[8] Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. 2008. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions.. In *OSDI*, Vol. 8. 117–130.

[9] Ankit Choudhary, Shan Lu, and Michael Pradel. 2017. Efficient Detection of Thread Safety Violations via Coverage-guided Generation of Concurrent Tests. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*.

[10] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale internet services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 217–231.

[11] Csharpmm. [n. d.]. Standard ECMA-334 C# Language Specification. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[12] DateTimeExtensions. [n. d.]. Resolve a random race condition. https://github.com/joaomatossilva/DateTimeExtensions/pull/86.

[13] Dawson R. Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles , SOSP , Bolton Landing, NY, USA*.

[14] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel.. In *OSDI*, Vol. 10. 1–16.

[15] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's Distributed and Caching Build Service. In *SEIP* (seip ed.). IEEE - Institute of Electrical and Electronics Engineers. https://www.microsoft.com/en-us/research/publication/cloudbuild-microsofts-distributed-and-caching-build-service/

[16] Facebook. [n. d.]. A tool to detect bugs in Java and C/C++/Objective-C code before it ships. https://fbinfer.com/.

[17] Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/258492.258493

[18] Colin J Fidge. 1987. Timestamps in message-passing systems that preserve the partial ordering. (1987).

[19] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[20] FluentAssertion. [n. d.]. Race condition in SelfReferenceEquivalencyAssertionOptions.GetEqualityStrategy. https://github.com/fluentassertions/fluentassertions/issues/862.

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

[22] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 174–186.

[23] Patrice Godefroid. 2005. The Soundness of Bugs is What Matters. In *PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools (BUGS'2005)*.

[24] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. 2012. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*.

[25] Chun-Hung Hsiao, Satish Narayanasamy, Essam Muhammad Idris Khan, Cristiano L. Pereira, and Gilles A. Pokam. 2017. AsyncClock: Scalable Inference of Asynchronous Event Causality. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, Xi'an, China*.

[26] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race Detection for Event-Driven Mobile Applications. In *PLDI*.

[27] 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 (PLDI)*.

[28] Jb Evain. [n. d.]. Mono.Cecil. https://github.com/jbevain/cecil.

[29] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[30] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS , London, UK*.

[31] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: crowdsourced data race detection. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, Farmington, PA, USA*.

[32] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[33] kubernetes-client. [n. d.]. fix a race condition. https://github.com/kubernetes-client/csharp/pull/212.

[34] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[35] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 399–414.

[36] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and*

Implementation (PLDI).

[37] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 155–162.

[38] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. 2006. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, San Jose, CA, USA*.

[39] Brandon Lucia and Luis Ceze. 2009. Finding Concurrency Bugs with Context-aware Communication Graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[40] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race Detection for Android Applications. In *PLDI*.

[41] J. Manson, W. Pugh, and S. Adve. 2005. The Java memory model. In *Proceedings of POPL*. ACM, 378–391.

[42] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. 2009. LiteRace: effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland*.

[43] Madan Musuvathi and Shaz Qadeer. 2006. CHESS: Systematic stress testing of concurrent software. In *International Symposium on Logic-based Program Synthesis and Transformation*. Springer, 15–16.

[44] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. 2007. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis.

[45] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. 2007. Automatically classifying benign and harmful data racesallusing replay analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA*.

[46] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid Dynamic Data Race Detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[47] Soyeon Park, Shan Lu, and Yuanyuan Zhou. 2009. CTrigger: exposing atomicity violation bugs from their hiding places. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 25–36.

[48] Eli Pozniansky and Assaf Schuster. 2007. MultiRace: Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs: Research Articles. *Concurr. Comput. : Pract. Exper.* 19, 3 (March 2007), 327–340. https://doi.org/10.1002/cpe.v19:3

[49] Michael Pradel and Thomas R. Gross. 2012. Fully Automatic and Precise Detection of Thread Safety Violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 521–530. https://doi.org/10.1145/2254064.2254126

[50] Radical. [n. d.]. MessageBroker internal subscription(s) list is not thread safe. https://github.com/RadicalFx/Radical/issues/108.

[51] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-finish Parallelism. In *Proceedings of the First International Conference on Runtime Verification (RV'10)*. Springer-Verlag, Berlin, Heidelberg, 368–383. http://dl.acm.org/citation.cfm?id=1939399.1939430

[52] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 531–542. https://doi.org/10.1145/2254064.2254127

[53] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective Race Detection for Event-Driven Programs. In *OOPSLA*.

[54] Malavika Samak and Murali Krishna Ramanathan. 2014. Multithreaded Test Synthesis for Deadlock Detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*.

[55] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. 2015. Synthesizing Racy Tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[56] Anirudh Santhiar and Aditya Kanade. 2017. Static Deadlock Detection for Asynchronous C# Programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[57] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[58] Koushik Sen. 2008. Race directed random testing of concurrent programs. *ACM Sigplan Notices* 43, 6 (2008), 11–21.

[59] Sequelocity.NET. [n. d.]. Race condition on TypeCacher. https://github.com/AmbitEnergyLabs/Sequelocity.NET/pull/23.

[60] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. ACM, 62–71.

[61] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[62] statsd.net. [n. d.]. Race conditions when updating guage value. https://github.com/lukevenediger/statsd.net/issues/29.

[63] System.Linq.Dynamic. [n. d.]. Fix the multithreading issue at ClassFactory.GetDynamicClass. https://github.com/kahanu/System.Linq.Dynamic/pull/48.

[64] Thunderstruck. [n. d.]. Race condition in ConnectionStringBuffer singleton. https://github.com/19WAS85/Thunderstruck/issues/3.

[65] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, Vol. 39. ACM, 221–234.

[66] Sheng Zhan and Jeff Huang. 2016. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA*.