Waffle: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection

Bogdan Alexandru Stoica
University of Chicago
bastoica@uchicago.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Suman Nath
Microsoft Research
suman.nath@microsoft.com

Abstract

Concurrency bugs are difficult to detect, reproduce, and diagnose, as they manifest under rare timing conditions. Recently, active delay injection has proven efficient for exposing one such type of bug — thread-safety violations — with low overhead, high coverage, and minimal code analysis. However, how to efficiently apply active delay injection to broader classes of concurrency bugs is still an open question.

We aim to answer this question by focusing on MemOrder bugs — a type of concurrency bug caused by incorrect timing between a memory access to a particular object and the object's initialization or deallocation. We first show experimentally that the current state-of-the-art delay injection technique leads to high overhead and low detection coverage since MemOrder bugs exhibit particular characteristics that cause high delay density and interference. Based on these insights, we propose Waffle — a delay injection tool that tailors key design points to better match the nature of MemOrder bugs. Evaluating our tool on 11 popular open-source multi-threaded C# applications shows that Waffle can expose more bugs with less overhead than state-of-the-art techniques.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Computer systems organization** → *Reliability*.

Keywords: memory ordering bugs; order violations; concurrency bugs; delay injection; debugging; reliability

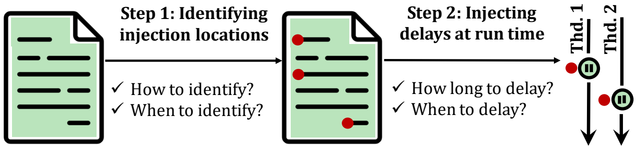


Figure 1. The workflow of active delay injection

ACM Reference Format:
Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2023. Waffle: Exposing Memory Ordering Bugs Efficiently with Active Delay Injection. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 9–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3552326.3567507

1 Introduction

Concurrency bugs are difficult to detect and time-consuming to diagnose. Typically, they only manifest under rare timing conditions [5, 25, 30, 32], many escaping rigorous in-house testing and causing severe production failures [16, 20, 23, 58].

Among the various approaches to detect concurrency bugs before code release, *active delay injection* [12, 26, 32, 46, 53] has an inherent advantage of high detection accuracy. Specifically, this approach reports a bug only *after* it manifests as a *consequence* of the delays injected. However, how to apply active delay injection to a wide variety of concurrency bugs, not only with high accuracy but also with low overhead and high bug coverage, is still an open question.

As Figure 1 illustrates, active delay injection identifies strategic program locations in a target program where concurrency bugs may exist (Step 1). Then, at run time, it injects delays at those particular locations attempting to trigger rare timing conditions and increase the chances of exposing concurrency bugs (Step 2). However, despite its inherent accuracy advantage, active delay injection traditionally suffers from high overhead and potentially low bug coverage.

When identifying injection locations (Step 1), there is a tension between the analysis cost and the quality of the locations identified. On the one hand, pruning program locations already synchronized and hence not needing delay injection, requires costly synchronization analysis [46, 53], such as

EuroSys '23, May 9–12, 2023, Rome, Italy
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3567507

happens-before analysis [27] (typically, 5–10× slowdowns reported by prior studies [14, 32]). On the other hand, performing no synchronization analysis results in too many injection points [12, 26] and hence high injection overhead.

When carrying out delay injection (Step 2), there is a tradeoff between the cost per run and the number of runs needed to expose bugs. Prior work [12, 26] typically samples only a few injection locations in each run, lowering the overhead per run at the expense of needing many runs to expose bugs.

Recently, TSVD [32] effectively adapted active delay injection to detect a particular type of concurrency bug — thread-safety violations — not only with high accuracy but also with low overhead and high coverage. In Step 1, TSVD completely abandons the expensive happens-before analysis and instead relies on easy-to-measure physical time gaps between operations to infer which program locations are likely un-synchronized. In Step 2, TSVD injects delays aggressively in each run to minimize the number of runs needed to expose bugs.

Although effective, TSVD leaves behind an intriguing question: Does its unique approach to active delay injection work for broader classes of concurrency bugs? And if not, are there other design changes to active delay injection that work?

We investigate these questions by focusing on MEMORDER bugs — a type of concurrency bug caused by the lack of synchronization between access to a memory object and its initialization or deallocation (disposal).

It is crucial to expose MEMORDER bugs before software release as they are both common and severe. According to previous studies, MEMORDER bugs are the dominant type of order violations [37, 56]. Moreover, they lead to memory corruption which can cause crashes [19, 22] and serious security vulnerabilities [28, 60].

MEMORDER bugs also present significant research challenges as they have drastically different location and timing properties from thread-safety violations, well representing real-world concurrency bugs beyond those tackled by TSVD:

- *Location-wise*, thread-safety violations can only occur at call sites of thread-unsafe APIs [32]. In contrast, MEMORDER bugs can occur at any memory access to shared heap objects, which are much more common;
- *Timing-wise*, exposing a thread-safety violation requires the execution windows of two thread-unsafe API calls to overlap. Conversely, exposing a MEMORDER bug requires memory accesses to an object to occur before the object's initialization or after its deallocation/disposal (see Figure 2). These represent two fundamentally different concurrency-bug timing conditions: atomicity violations for the former and order violations for the latter [37].

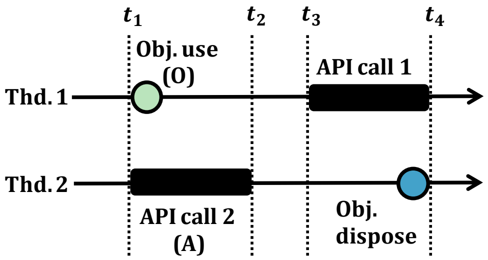


Figure 2. Different timing conditions. To make API calls 1 and 2 execute concurrently, the delay length needs to be within a range: $(T_4 - T_1) > \text{delay}_A > (T_3 - T_2)$. In contrast, to force an object use after it has been deallocated, the delay needs to be sufficiently long: $\text{delay}_O > (T_4 - T_1)$.

1.1 Adapting the state-of-the-art

We began our investigation by adapting the delay injection design of TSVD to detect MEMORDER bugs. Unfortunately, the efficacy of the resulting tool — referred to as WAFFLEBASIC — is limited. Our evaluation reveals that WAFFLEBASIC injects delays at a much higher rate than TSVD, and struggles with greater overhead and lower bug detection coverage.

This experience led us to decompose the workflow of active delay injection into four key design points as illustrated in Figure 1 — to understand why TSVD's approach does *not* work for MEMORDER bugs and propose a new design that would.

How to identify delay candidate locations? Analyzing WAFFLEBASIC reveals that, due to their location properties, there are simply too many program locations where MEMORDER bugs can manifest for TSVD's inference heuristics to prune effectively. In turn, this leads to abundant (dense) and often pointless delay injection.

In particular, we observe that many delays injected by WAFFLEBASIC are needlessly attempting to reverse the execution order between memory accesses across the boundaries of thread forks. This particular type of synchronization can be analyzed accurately with negligible run-time overhead by using a special type of *thread-local storage*, a feature available in modern languages such as Java and C#. Then, the number of delays injected at run time can be reduced considerably with little analysis cost.

When to identify candidate locations? To minimize the number of bug-detection runs, TSVD uses the (buggy) location identification heuristic in the same run where it injects delays. Specifically, after identifying a program location ℓ where a bug may exist, TSVD considers injecting a delay the next time ℓ is executed during the same run.

We found such a strategy not suitable for WAFFLEBASIC. This approach results in abundant (dense) delay injection which severely affects the efficacy of location identification which relies on physical time information. Furthermore, such

Design decisions	RaceFuzzer [53]	CTrigger [46]	RaceMob [26]	DataCollider [12]	Tsvd [32]	Waffle
How to identify candidate location?						
Synchronization analysis?	✓	✓	✓	×	×	✓*
Synchronization inference?	×	×	×	×	✓	✓
When to identify candidate locations?						
During delay injection runs?	×	×	×	×	✓	×
How long is the delay?						
A fixed-length delay	✓	✓	×	✓	✓	×
When to inject at run time?						
Avoid delay interference?	n/a	n/a	n/a	n/a	×	✓
At sampled candidate loc.?	✓	✓	✓	✓	×	×
Probabilistic injection?	×	×	✓	✓	✓	✓

Table 1. Different design decisions for recent active delay injection tools. (“*” indicates partial analysis is done; “n/a” stands for «not applicable» as these tools deal with sparse sets of injection locations.)

a strategy often cannot help expose MemOrder bugs in one run, as many of their candidate locations like object initialization/disposal execute only a small number of times (often once) in each run.

Consequently, separating location identification and delay injection in different runs could fit MemOrder bugs better.

How long is the delay? Tsvd uses a fixed delay length for all candidate locations. Relying on this strategy combined with having to handle a denser set of candidate locations for MemOrder bugs results in an unfortunate trade-off between longer delays required for bug-exposing capabilities and shorter delays for lower run-time overhead.

In comparison, injecting delays with different lengths at different locations is a more suitable strategy for MemOrder bugs: the long delays required for exposing certain bugs will not lead to unnecessary delays at many other program locations.

When to inject delay at run time? To minimize the number of runs, Tsvd injects delays at candidate locations with high probability[1] compared with previous work, while also allowing multiple threads to pause simultaneously. Unfortunately, due to the location property of MemOrder bugs, this strategy leads to severe delay overlapping in Waffle-Basic, with many delays canceling each other. Such delay interference (and ensuing cancellation) occurs almost deterministically for some MemOrder bugs, due to their unique timing properties.

Therefore, MemOrder bugs require more careful coordination during delay injection.

1.2 A new design

Guided by these insights, we propose Waffle — a delay injection tool for exposing MemOrder bugs that explores new trade-offs and heuristics for each design point discussed above. Given a program under test, Waffle runs it once to identify candidate injection locations with lightweight happens-before analysis. In subsequent runs, often just one, Waffle carries out delay injection with a carefully designed delay-or-not decision-making process guided by the information collected from the first run.

Table 1 summarizes Waffle’s design decisions, and shows how our tool compares to Tsvd and other recent delay injection techniques — techniques that rely on expensive program analysis and/or need many runs to expose bugs, as discussed above.

We implemented Waffle for C# and evaluated it on 11 popular open-source multi-threaded C# applications. Our experiments show that Waffle successfully exposes 18 MemOrder bugs, including 12 known and 6 *previously unknown* issues, without any bug-related prior knowledge. Waffle manages to reliably discover and trigger most of these bugs (15 out of 18) in just two runs. The end-to-end slowdown is only 2.5× compared with running the bug-triggering input once without any instrumentation. Our evaluation also shows that Waffle exposes more bugs with much less overhead than various alternative designs.

Starting from an attempt to understand why Tsvd works so effectively for thread-safety violations and whether a simple adaptation suffices for MemOrder bugs, our study ends up with a completely different active delay injection design and a new tool, Waffle. We hope our journey sheds light on how to architect active delay injection tools for broader classes of concurrency bugs. Consequently, we make Waffle available at https://github.com/bastoica/waffle.

[1] Deterministically injecting delays at every candidate location is widely considered unacceptable, due to the huge overhead and delay interference.

2 Background

As mentioned in §1, our first goal is to understand the state-of-the-art for active delay injection. Thus, we begin by presenting a few details about Tsvd [32].

Tsvd is an active delay injection tool that aims to expose thread-safety violations (TSVs), a type of concurrency bug that manifests when the execution windows of two thread-unsafe API calls operating on the same object overlap. Empirically, it reports significantly more TSVs in proprietary software with much less overhead than traditional delay injection techniques [32]. To achieve this, Tsvd works as follows:

How to identify delay candidate locations? Given a program binary, Tsvd first instruments program locations where thread-safety violations are likely to occur, namely call sites of thread-unsafe APIs [32]. At run time, Tsvd collects information at these instrumentation points and leverages two heuristics to maintain a set $\mathcal{S}$ of thread-safety violation *candidates.* Each candidate consists of a pair of program locations $\{\ell_1, \ell_2\}$ where API calls at ℓ_1 and ℓ_2 may contribute to thread-safety violations. Consequently, ℓ_1 and ℓ_2 are candidate locations where Tsvd injects delays and exposes potential thread-safety violations. We refer to $\mathcal{S}$ as the *candidate set.*

The first heuristic — near-miss tracking — *adds* candidate pairs to $\mathcal{S}$ based on constraints related to the 2 corresponding threads, objects accessed, and the physical timestamps of the operations involved. Specifically, if one thread-unsafe API is invoked at location ℓ_1 from thread thd_1 accessing object obj_1 at time τ_1, while another is invoked at location ℓ_2 from thread thd_2 accessing obj_2 at time τ_2, Tsvd adds $\{\ell_1, \ell_2\}$ to $\mathcal{S}$ iff. $obj_1 = obj_2$, $thd_1 \neq thd_2$ and $|\tau_1 - \tau_2| \leq \delta$, for a time gap threshold δ (called the *near-miss window* [32]). The intuition is that two thread-unsafe APIs accessing the same object from different threads are more likely to cause a thread-safety violation if they execute close to each other at run time.

The second heuristic — happens-before inferencing — *removes* candidate pairs from $\mathcal{S}$, namely those unlikely to trigger thread-safety violations, based on delay injection feedback. Assume Tsvd added a candidate pair $\{\ell_1, \ell_2\}$ to $\mathcal{S}$. When a delay is injected before ℓ_1 in thread thd_1, Tsvd checks whether it causes a proportional slowdown before location ℓ_2 in thread thd_2. If true, Tsvd infers that there is a likely happens-before relationship between ℓ_1 and ℓ_2 and consequently removes $\{\ell_1, \ell_2\}$ from $\mathcal{S}$.

When to identify candidate locations? To minimize the number of bug-detection runs, Tsvd uses the above two heuristics to dynamically update $\mathcal{S}$ during the same run it injects delays in. After adding $\{\ell_1, \ell_2\}$ into the candidate set, Tsvd injects a delay before ℓ_1 at the immediate next opportunity, especially if ℓ_1 is exercised again in the same run.

How long is the delay? Tsvd relies on fixed-length delays. Specifically, Tsvd injects a `Thread.Sleep()` operation of δ milliseconds before a candidate location (e.g. ℓ_1). The configuration of δ balances the performance and bug-exposing capabilities: when δ is too short, many bugs are missed; when δ is too long, delay injection overhead becomes prohibitive.

When to inject at run time? For each candidate pair $\{\ell_1, \ell_2\}$, Tsvd injects a delay with 100% probability when ℓ_1 is exercised for the first time. However, each time this action fails to expose a thread-safety violation, the probability of injecting a delay before ℓ_1 in the future drops by a small constant $\lambda > 0$. When this probability reaches 0, all candidate pairs involving ℓ_1 are removed from $\mathcal{S}$. This decay constant is carefully set: If λ is too small, many ineffective delays would contribute to an overhead increase. If too large, only a few candidate locations are delayed, thus many runs are needed to thoroughly search for bugs. We refer to this strategy as *probability decay.*

Additionally, Tsvd injects delays aggressively and allows multiple threads to be blocked in parallel, to reduce the number of runs needed to expose thread-safety violations. Although delays injected simultaneously could overlap creating interference and thus canceling each other's effect, the sparsity of candidate locations related to thread-safety violations, combined with the probability decay scheme avoid such interference in most situations [32].

3 WaffleBasic: Adapting Tsvd

In our first attempt to detect MemOrder bugs using active delay injection, we design WaffleBasic by adapting Tsvd [32] for this type of bug. On the one hand, WaffleBasic departs from Tsvd by operating on different program locations, relevant to MemOrder bugs (§3.1). On the other hand, WaffleBasic preserves Tsvd's core delay injection philosophy (§3.2). Unfortunately, these design choices combined with the location and timing properties of MemOrder bugs limit WaffleBasic's efficacy (§3.3).

3.1 How to identify delay candidate locations?

Instrumentation sites. WaffleBasic first instruments every program location where a MemOrder bug could occur. Specifically, WaffleBasic instruments all operations related to reference-type variables, namely access to member fields and calls to member methods of heap objects. For each operation, WaffleBasic records the corresponding object ID, physical timestamp, the operation type, and the active thread.

At run time, an instrumented operation is categorized into one of three types: object initialization, object disposal, and object use. An operation that changes the object's reference from NULL to non-NULL is considered an *object initialization.* An operation that changes the state of the object's reference from non-NULL to NULL or makes an explicit call to the object's destructor (i.e., Dispose() method) is considered an *object disposal.* Calling one of the object's member methods or accessing one of its member fields is considered an *object use.*

Adding to the candidate set $\mathcal{S}$. WaffleBasic adapts the near-miss heuristic of Tsvd to match the characteristics of MemOrder bugs. Consider an object initialization (object use) at location ℓ_1 executed by Thread 1 on object obj_1 at time τ_1, and another object use (object disposal) at location ℓ_2 executed by Thread 2 on object obj_2 at time τ_2. WaffleBasic adds $\{\ell_1, \ell_2\}$ to $\mathcal{S}$ as a candidate use-before-initialization (use-after-free) MemOrder bug iff. $obj_1 = obj_2$, $thd_1 \neq thd_2$, $\tau_2 - \tau_1 < \delta$, where δ is the size of the near-miss window.

Specifically, $\{\ell_1, \ell_2\} \in S$ forms a MemOrder *bug candidate* and ℓ_1 becomes a *candidate location* where WaffleBasic injects delays attempting to expose this potential (candidate) MemOrder bug. In other words, WaffleBasic injects delays before an object initialization hoping to force it to execute after its corresponding object use (as recorded in $\mathcal{S}$), and before an object use hoping to force it to execute after its corresponding object disposal (as recorded in $\mathcal{S}$).

Removing from the candidate set $\mathcal{S}$. WaffleBasic similarly adapts the happens-before inference heuristic of Tsvd. Given a candidate pair $\{\ell_1, \ell_2\}$, WaffleBasic checks whether a delay injected before ℓ_1 is observed to block the progress of the other thread right before ℓ_2. If the delay propagates, ℓ_1 and ℓ_2 are likely ordered by a happens-before relationship and WaffleBasic removes the pair from $\mathcal{S}$.

3.2 What about the other design decisions?

The remaining design decisions of WaffleBasic follow those of Tsvd.

When to identify candidate locations? WaffleBasic injects delays in the same run in which it adds or removes pairs from the candidate set $\mathcal{S}$. This way, WaffleBasic attempts to expose MemOrder bugs in a minimal number of runs.

How long is the delay? Similar to Tsvd, WaffleBasic injects delays of a fixed length δ. This constant is set to 100 milliseconds, exactly as in Tsvd.

When to inject delays at run time? Like Tsvd, WaffleBasic implements a *probability decay* scheme. Similarly, WaffleBasic injects delays at candidate locations in $\mathcal{S}$ with a probability that starts at 100% and gradually decreases towards 0 if no MemOrder bugs could be uncovered there. Likewise, WaffleBasic also allows multiple delays to block multiple threads in parallel.

App	Instrumentation Sites		Injection Sites	
	TSV	MO	TSV	MO
ApplicationIns.	8.7	188.6	0.1	3.5
FluentAssert.	57.3	76.9	0.3	5.9
Kubernetes	5.6	338.5	1.5	3.8
MQTT.Net	23.2	544.1	7.9	156.6
NetMQ	49.2	619.0	13.5	143.4
NSubstitute	1.3	261.4	0.6	10.7
NSwag	2.2	110.4	0.3	70.8
Ssh.Net	56.3	179.0	0.4	13.1

Table 2. The average number of unique static instrumentation and delay-injection sites for thread-safety violations (TSV) and MemOrder bugs (MO) across all test inputs.

3.3 How effective is WaffleBasic?

We evaluated WaffleBasic using 11 open-source applications with 12 previously reported MemOrder bugs (details in Tables 3 and 4). Our experiments found that although WaffleBasic can expose some MemOrder bugs, it fails to expose others even after many runs. Experiments also reveal that WaffleBasic incurs a significant overhead for several applications. In particular, we observed several shortcomings of WaffleBasic that reflect the fundamental difference between MemOrder bugs and thread-safety violations. These shortcomings led us to the design of Waffle (§4).

Too many program locations in play. MemOrder bugs and thread-safety violations have different location properties. Thus, WaffleBasic needs to handle a much more numerous set of instrumentation sites than Tsvd (i.e., program locations that access heap objects versus thread-unsafe API call sites). For 8 out of 11 applications[2] in our benchmark suite, WaffleBasic's instrumentation sites are over 10× more common than Tsvd's, in most cases (see "Instrumentation Sites" columns in Table 2). With a larger base to start with, more program locations tend to pass the near-miss heuristic at run time and get added to the candidate set $\mathcal{S}$. Similarly, the number of delay injection locations identified by WaffleBasic is predominantly an order of magnitude more than those identified by Tsvd (see "Injection Sites" columns in Table 2).

Too much delay overlap. A denser set of instrumentation sites means more delays injected at run time which, in turn, increases delay overlap. To quantify this overlap we run every test suite for the benchmarks in Table 2 and compute the complement of the ratio between the "time projection" of all delays over the total delay value injected. This way, if no delays overlap, the value is 0 while if all delays overlap the ratio is close to 1 (i.e., $\frac{D-1}{D}$, with D representing the total

[2] The public version of Tsvd cannot instrument the other 3 applications in our benchmark suite.

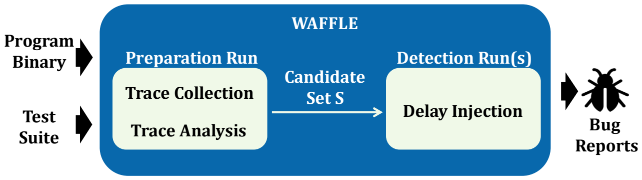


Figure 3. Workflow diagram of WAFFLE.

number of delays injected at run time). For TSVD, the average overlap is less than 1% for 6 out of the 8 applications, with the remaining 2 applications showing an average overlap of 12% and 15%, respectively. In contrast, for WAFFLEBASIC, the average overlap is 2 – 28%, with 3 applications above 25%.

Too few dynamic instances. The main reason TSVD detects many thread-safety violations in just one run is that most thread-unsafe API calls are executed multiple times per run, offering multiple chances for bug manifestation [32]. Unfortunately, this is not true for MEMORDER bugs. In particular, many objection initialization operations naturally execute a small number of times each run. In our evaluation, the median number of dynamic instances for all object initialization operations is 2 across all unit tests for all applications.

4 WAFFLE: A New Design

To tackle the challenges faced by WAFFLEBASIC, we propose WAFFLE. As illustrated in Figure 3, WAFFLE works as follows:

- First, WAFFLE executes the targeted program binary once, without injecting any delay, to record a delay-free execution trace. We refer to this as the *preparation run.*
- Next, WAFFLE analyzes this unperturbed trace to (i) construct the set of candidate location pairs $\mathcal{S}$, (ii) determine the delay length for each candidate location, and (iii) identify which candidate locations might interfere with each other (§4.1—4.4).
- Finally, WAFFLE carries out delay injection in subsequent runs using information collected during the preparation run as well as from the ongoing run itself (§4.4). We refer to these as *detection runs.*

In the rest of this section, we describe WAFFLE's design decisions and how they differ from WAFFLEBASIC.

4.1 How to identify delay candidate locations?

What went wrong in WAFFLEBASIC? To identify candidate locations, WAFFLEBASIC (like TSVD) completely disregards traditional happens-before analysis. Instead, it uses run-time heuristics (near-miss tracking and happens-before inference) to *infer* what operations may be un-synchronized and hence should be part of the candidate set $\mathcal{S}$. Unfortunately, this design did not work well for WAFFLEBASIC, affecting detection overhead and bug coverage, for several reasons.

First, MEMORDER bugs naturally present many more instrumentation sites than thread-safety violations (Table 2) which contributes to an increased number of delays getting injected and higher run-time overhead.

Second, the happens-before inference can be less accurate in WAFFLEBASIC due to delay overlaps. WAFFLEBASIC (like TSVD) infers happens-before relationships between two locations ℓ_1 and ℓ_2, iff. delaying Thread 1 before exercising ℓ_1 causes a proportional slowdown of Thread 2 before exercising ℓ_2. However, if a second delay is injected around the same time in Thread 2, thus overlapping with the first in Thread 1 (see diagram in Figure 5), the happens-before inference heuristic cannot reliably determine whether the slowdown in Thread 2 is caused by a synchronization operation or it is solely the effect of the second delay. Consequently, the more delays overlapping, the less the effective happens-before inference heuristic is.

Finally, even when the happens-before inference is as accurate as in TSVD, it offers little help to those locations that only execute a small number of times each run (e.g. object initializations). By the time WAFFLEBASIC infers the happens-before relationship, the program locations involved no longer get exercised during that same run.

Design of WAFFLE. At first glance, WAFFLE may need to go back to full-blown happens-before analysis, which would require significant manual effort in annotating synchronization operations, in addition to the high overhead incurred by the happens-before analysis itself [31, 32].

Fortunately, we found that a sizable fraction of MEMORDER bug candidates is causally ordered by a specific type of happens-before relationship — that between parent and child threads. Typically, this happens because many objects are allocated in a parent thread before the worker threads are created. Consequently, WAFFLE replaces the happens-before inference in WAFFLEBASIC with a parent-child thread relationship analysis. Pruning these ordered MEMORDER candidates during the preparation run reduces the number of candidate program locations where WAFFLE has to inject delays. As Table 7 shows, failing to remove them decreases the average performance of our tool by 1.17×. However, the impact on memory-intensive applications is much greater (e.g. 1.73× for NpgSQL)

To track parent-child thread relationships, traditionally we would need to instrument every program location where the parent forks a child thread. This is challenging in modern object-oriented languages such as Java and C# which have multiple mechanisms for thread creation. Instead of instrumenting various types of thread fork operations, WAFFLE leverages a special type of thread-local storage (TLS) that automatically gets copied from a parent to all child threads at the moment of thread creation. This language feature is supported by other modern object-oriented programming languages such as Java (`InheritableThreadLocal` [41]) and C++ (`LogicalCallContext` [40]).

Note that while Waffle only considers threads, .NET provides a similar mechanism for task-oriented programming — *async-local storge* — which supports state propagation from a parent to a child *task* irrespective of which thread these tasks are scheduled to run on.

Waffle tracks happens-before relationships induced by thread forks by implementing vector clocks on top of the TLS mechanism. More precisely, Waffle creates and stores a tailored thread-local vector clock object in the TLS memory region of each thread. This vector clock is represented by a set of tuples $\{(tid_1, \&rctr_1), (tid_2, \&rctr_2), ...\}$, with each tuple representing a thread ID and a reference (pointer) to the corresponding logical time counter. When a child thread is created, the TLS memory region of the parent thread (and, consequently, the vector clock object with it) gets automatically propagated to the child thread. At this point in the execution, Waffle allocates a vector clock for the child thread using information from the vector clock "cloned" from its parent. Specifically, Waffle implements the constructor of the vector clock object to (1) append a tuple $(tid_k, \&rctr_k = 1)$, with tid_k being the child thread ID, to the vector clock content copied from the parent thread; and (2) increment the logical counter of the parent using the counter reference (pointer) passed through the TLS. Note that while vector clock updates happen as part of the TLS propagation which is triggered by a thread fork operation, the parent's vector clock remains inaccurate until TLS region is completely copied to the child thread—as the value is not incremented right before the fork happens. However, Waffle makes no vector clock comparisons in this time frame.

Waffle leverages these vector clocks during near-miss tracking. Consider a candidate location pair $\{\ell_1, \ell_2\}$ that satisfy all requirements related to timing, active threads, and memory access set forth by the near-miss tracking heuristic (§4.1). Waffle additionally checks whether the corresponding vector clocks of the two active threads at time τ_1 and τ_2, respectively, cannot be partially ordered before deciding to add the pair to $\mathcal{S}$ or not.

4.2 When to identify candidate locations?

What went wrong in WaffleBasic? The design decision of combining candidate locations identification and delay injection into the same run does not benefit detecting MemOrder bugs as much as it benefits detecting thread-safety violations. This happens because program locations involved in many MemOrder bugs have only a few dynamic instances, as mentioned in §3.3. Consequently, starting delay injection in the same run makes little difference for program locations that execute a few times, if at all, after being identified as candidate locations.

Furthermore, our evaluation shows that the injected delays sometimes interfere with candidate location identification, which relies on physical time information. For example, a pair of candidate locations $\{\ell_1, \ell_2\}$ observed during a delay-free run may "disappear" once delays are injected, as delays injected between ℓ_1 and ℓ_2 could prevent them from executing close to each other, failing the $|\tau_1\text{-}\tau_2| < \delta$ requirement. Intuitively, the more delays injected at run time, the more severe this interference is.

Design of Waffle. In contrast, Waffle conducts a preparation run (delay-free) for planning purposes (see Figure 3), before injecting delays in subsequent, detection runs. In this first run, Waffle uses the near-miss heuristic together with the parent-child relationship pruning to construct the candidate locations set $\mathcal{S}$. In subsequent runs, Waffle injects delays at these locations. In addition, Waffle leverages the delay-free environment in the first run to collect timing information to help guide delay injection. We will elaborate more in the next two sub-sections.

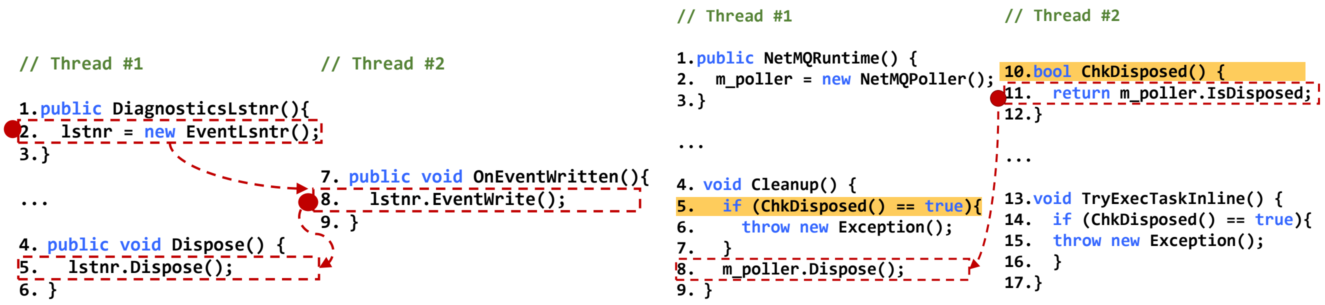
Note that using a delay-free run can potentially increase the cost of Waffle, as at least two runs are now needed to expose a MemOrder bug. We believe the benefits outweigh the extra cost, and we experimentally validate this claim during evaluation (§6).

4.3 How long is the delay?

What went wrong in WaffleBasic? WaffleBasic struggles to find a delay length that can balance performance and bug-exposing capabilities. Compared to Tsvd, WaffleBasic incurs significantly more overhead under the same delay length setting due to starting with a larger candidate set $\mathcal{S}$ (§3.3). For example, using a delay length of 100 milliseconds, Tsvd incurs 15%, 9%, and 11% overhead when running all multi-threaded tests available for ApplicationInsights, FluentAssertion, and Kubernets.Net, respectively [32]. In contrast, WaffleBasic incurs over 100% overhead for the same three applications (Table 5).

Of course, we could lower the overhead by using a much shorter delay. However, the bug-exposing capabilities of WaffleBasic would suffer. For example, decreasing the delay length from 100 to 10 milliseconds would speed up the average performance of WaffleBasic by about 4 times across all multi-threaded unit tests available for NetMQ. Unfortunately, in that case, the known MemOrder bug [42] which could be exposed by WaffleBasic when utilizing delays of 100 milliseconds, cannot be triggered with delays of only 10 milliseconds even after many runs (§6.2-§6.3).

Design of Waffle. To address this challenge, Waffle leverages the observation that different bugs have different time gaps between corresponding operations in bug-free



```
// Thread #1
1.public DiagnosticsLstnr(){
2.  lstnr = new EventLstnr();
3.}
...
4. public void Dispose() {
5.   lstnr.Dispose();
6. }
```

```
// Thread #2
7. public void OnEventWritten(){
8.   lstnr.EventWrite();
9. }
```

(a) ApplicationInsights Issue #1106 [39]: Interfering bugs

```
// Thread #1
1.public NetMQRuntime() {
2.  m_poller = new NetMQPoller();
3.}
...
4. void Cleanup() {
5.   if (ChkDisposed() == true){
6.     throw new Exception();
7.   }
8.   m_poller.Dispose();
9. }
```

```
// Thread #2
10.bool ChkDisposed() {
11.  return m_poller.IsDisposed;
12.}
...
13.void TryExecTaskInline() {
14.  if (ChkDisposed() == true){
15.  throw new Exception();
16.  }
17.}
```

(b) NetMQ Issue #814 [42]: Interfering candidate locations

Figure 4. Examples of delay interference

runs (i.e., the gap between an object initialization and its use, or between an object use and its disposal) and hence opts to inject delays of different lengths at different locations. In particular, for the 12 known bugs in our evaluation (§6.1), measurements reveal that these time gaps range from less than 1 to around 100 milliseconds. Consequently, if we observe the time gap between ℓ_1 and ℓ_2 to be much shorter than ℓ_3 and ℓ_4 during a delay-free run, we can inject much shorter delays at ℓ_1 than at ℓ_3 during detection (i.e., delay injection) runs.

To achieve this, Waffle keeps track of time gaps between each pair of candidate locations in $\mathcal{S}$. For a candidate pair $\{\ell_1, \ell_2\} \in S$, Waffle creates a record with the delay length at that particular candidate location, $len_{\ell_1} = |\tau_1 - \tau_2|$. If ℓ_1 is part of multiple candidate location pairs (e.g. $\{\ell_1, \ell^*\} \in S$), Waffle updates len_{ℓ_1} with the larger gap (i.e., $len_{\ell_1} = \mathtt{MAX}(|\tau_1 - \tau_2|, |\tau_1 - \tau^*|)$.

During a detection run, Waffle injects delays proportional to the gap measured above. For instance, given a location ℓ_1, Waffle injects a delay of $\alpha \cdot len_{\ell_1}$ milliseconds, for a small constant $\alpha \geq 1$. In our experiments, $\alpha = 1.15$.

4.4 When to inject at run time?

What went wrong in WaffleBasic? As discussed in §3.3, WaffleBasic experiences much more delay overlap than Tsvd. These overlapping delays interfere with each other and cause WaffleBasic to miss some MemOrder bugs with significant probability. Most often, this occurs in two scenarios:

1. Interfering bugs: Sometimes, the manifestation of two bug candidates interfere with each other — one requires Thread 1 to execute faster than Thread 2, and the other requires Thread 2 to execute faster than Thread 1. When attempting to trigger them both, the injected delays cancel each other. Unfortunately, these cases are particularly common when, for example, attempting to trigger a use-before-initialization and use-after-free MemOrder bug on the same object instance (in the same run).

Figure 4a illustrates such an example from ApplicationInsights [39]. The bug manifests when the constructor takes longer than expected to allocate `lstnr` before a `WRITE` event occurs, which triggers the `OnEventWritten()` handler. WaffleBasic consistently misses this bug because it injects delays both before the allocation at line 2 in Thread 1, aiming to push the allocation after the object use (line 8), and before the object use at line 8 in Thread 2, aiming to push the use after the dispose operation (line 8). Therefore, WaffleBasic blocks both threads in parallel for the same duration and cannot trigger the bug even after 50 runs.

2. Interfering dynamic instances: Sometimes, WaffleBasic injects a delay at a location ℓ_1, hoping to make it execute after ℓ_2. Unfortunately, this delay repeatedly gets canceled out by a delay at another dynamic instance of ℓ_1, which is executed by the same thread right before exercising ℓ_2.

Figure 4b illustrates such an example from NetMQ [42]. The failure happens when a connection is abruptly terminated, causing several shared objects to be disposed (e.g. `m_poller` on line 8, Thread 1) while other threads are still processing network packages (e.g. `m_poller` on line 11, Thread 2). To trigger this bug, WaffleBasic injects a delay right before line 11, aiming to push the use of `m_poller` in Thread 2 to execute after the dispose operation in Thread 1. Unfortunately, since line 11 is also executed (in a different execution context) by Thread 1 right before the dispose call, both threads get delayed at around the same time and for the same duration. This, in turn, prevents WaffleBasic from exposing the bug even after 50 runs.

Design of Waffle. Similar to WaffleBasic and Tsvd, Waffle uses the *probability decay* strategy to inject delays at candidate locations with decreasing probabilities. Unfortunately, this alone is not enough to mitigate delay interference. Thus, Waffle proposes an additional, new heuristic to reduce delay interference.

A naïve solution to the delay interference problem is to modify WaffleBasic to inject only one delay per run, similar to prior work [46, 53]. However, this would require too many runs to expose the bug, as Waffle routinely observes tens, or even hundreds of location candidates after the preparation run, for just one input. A better solution might be to change

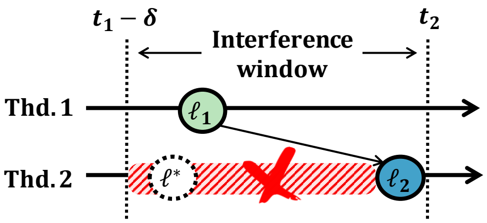


Figure 5. Illustration of delay interference. Highlighted, the interference window — when a concurrent delay injected in Thread 2 can cancel the effect of the delay injected before ℓ_1.

WAFFLEBASIC to avoid any parallel (i.e., overlapping) delays. However, completely avoiding parallel delays may cause some bugs to take many runs to trigger [32]. This could be even worse for MEMORDER bugs: If the candidate location ℓ of a bug executes only a small number of times each run, WAFFLEBASIC may never trigger the bug if another candidate location, from a different thread, keeps getting exercised right before ℓ does.

Therefore, WAFFLE proposes a new heuristic that aims to strike a balance between enabling parallel delay injection and reducing delay overlaps. The high-level idea is to enhance WAFFLEBASIC so that a delay planned to be injected before ℓ_1 is skipped when an *interfering* delay is ongoing. As illustrated in Figure 5, we consider a delay planned for ℓ^* in thread Thd$_2$ to interfere with another delay planned for ℓ_1 in a different thread Thd$_1$ if two conditions are met: (1) First, ℓ^* executes before ℓ_2 on the same thread Thd$_2$—if $\{\ell_1, \ell_2\}$ is a bug candidate that WAFFLE aims to expose by injecting a delay before ℓ_1, allowing another delay before ℓ^* will block Thd$_2$, essentially canceling out the original delay and preventing ℓ_1 execute after ℓ_2. (2) Second, ℓ^* needs to either execute shortly ahead of ℓ_1 or between ℓ_1 and ℓ_2—otherwise, the interference is negligible.

A challenge in implementing the above strategy is that we cannot predict whether ℓ_2 will be executed by thread Thd_2 (i.e., the thread of ℓ^*) at the time when execution reaches ℓ_1. Consequently, we cannot predict which delays might interfere with ℓ_1 when it gets exercised. To address this challenge, we leverage the preparation (i.e., delay-free) run. Specifically, we augment the near-miss heuristic as follows: when the execution reaches ℓ_2 and WAFFLE identifies $\{\ell_1, \ell_2\}$ as a candidate pair, it further checks if any other candidate location, say ℓ^*, was exercised by the same thread that reaches ℓ_2 at a time between $t_1 - \delta$ or and t_2. If so, the pair (ℓ_1, ℓ^*) is added to a global set $\mathcal{I}$ of locations that could interfere with each other if delayed simultaneously. $\mathcal{I}$ (along with $\mathcal{S}$ and the per-location delay-length values) is saved after analyzing the execution traces recorded during the preparation run and used to bootstrap future detection runs. This way, in future detection runs no delay gets injected at ℓ^* as long as there is another delay concurrently injected at a location interfering with ℓ^* (e.g., ℓ_1). Conversely, no delay gets injected at ℓ_1 as long as there is another delay concurrently injected at a location interfering with ℓ_1 (e.g. ℓ^*).

5 Implementation

WAFFLE. We implemented WAFFLE to find MEMORDER bugs in C# applications. WAFFLE measures 6, 378 lines of C# code and an additional 433 lines of Python and PowerShell scripts. WAFFLE is divided into three key components: (1) the instrumenter which statically instruments the target binary; (2) the trace analyzer which constructs the candidate set $\mathcal{S}$, the interference set $\mathcal{I}$, and determines appropriate delay lengths; and (3) the runtime which implements the delay injection algorithm.

1. WAFFLE's instrumenter. This component takes an application binary as input and wraps every access to object member fields or calls to member methods in a proxy function. The proxy function transfers control to WAFFLE's runtime library (see below) which implements the delay injection strategy. To instrument the binary, WAFFLE relies on a .NET instrumentation framework called Mono.Cecil [13].

WAFFLE executes the instrumented application at least twice. WAFFLE runs the instrumented application once to collect an unperturbed execution trace containing every access to heap objects (preparation). Note that no delay is injected in this preparation phase. WAFFLE runs the instrumented application at least one more time, to inject delays and expose MEMORDER bugs (detection).

2. WAFFLE's trace analyzer. This component analyzes a run time trace to identify pairs of memory accesses likely to cause MEMORDER bugs. At this stage WAFFLE constructs the candidate set $\mathcal{S}$, discarding those pairs ordered by happens-before relationships between parent and child threads, along with those with a physical time gap larger than the near-miss threshold. Next, it computes the appropriate delay length to inject for each candidate pair in $\mathcal{S}$. Finally, it identifies which candidate locations interfere with each other if delays are to be injected concurrently, constructing set $\mathcal{I}$.

3. WAFFLE's runtime. This component implements the core delay injection algorithm.

Recall that during the preparation run, the library logs all accesses to reference-type variables (heap objects) along with metadata such as timestamps, accessed object id, and access types (i.e., object initialization, object dispose, access to object fields, or call to object methods).

During the detection run, the library injects delays according to the delay planning done in the preparation run and updates delay probabilities at candidate locations based on the probability decay heuristic. After each detection run, the new delay probabilities are saved on disk and used to bootstrap the next detection run.

Finally, Waffle reports a bug only when the target binary raises a `NULL` reference exception as a consequence of the delay injection performed. At that time, the relevant runtime context (i.e., faulty input, candidate locations involved, stack traces for all threads, and delay value information) is recorded as part of the bug report.

Extending Waffle. To extend Waffle for applications written in other object-oriented languages, like Java and C++, we mainly need to change the underlying instrumentation framework. We would similarly instrument method calls and field accesses related to heap objects, and implement the same delay injection algorithms which rely primarily on language-independent time-based heuristics. One language-dependent feature used by Waffle is C#'s thread-local storage (TLS) mechanism which transfers state from a parent to a child thread and facilitates happens-before analysis (§4.1). However, similar language features are available for Java [41] and C++ [40].

WaffleBasic. Like Waffle, WaffleBasic uses the same .NET instrumentation framework (i.e., Mono.Cecil [13]). WaffleBasic instruments applications at similar locations as Waffle, although different delay injection algorithms are conducted at run time. Furthermore, WaffleBasic does not instrument applications to produce traces and does not contain a trace analyzer. The delay injection policy implemented by WaffleBasic measures 447 lines of C# code.

6 Evaluation

Benchmarks. We evaluate Waffle on 11 popular open-source multi-threaded C# applications (Table 3). We made this selection by searching Github for C# applications that (1) are popular, measured by the number of Github stars; (2) have well-maintained test suites which would help us conduct a systematic evaluation; and (3) contain confirmed and clearly described MemOrder bugs in their issue trackers.

To find MemOrder bugs, we first searched for issue reports containing keywords such as "data race" or "race condition". We further narrowed down the results using keywords like "exception" or "crash". Finally, we manually inspected the remaining reports to confirm they are describing MemOrder bugs and include bug-triggering inputs.

6.1 Methodology

Following instructions in these MemOrder bug reports, we were able to manually reproduce 12 previously known MemOrder bugs in 9 applications (the top 12 bugs in Table 3). These helped us evaluate the bug-detection capabilities of Waffle and WaffleBasic. Although we were unable to reproduce the MemOrder bugs reported in SignalR and MQTT.Net (we suspect the reported bug-triggering inputs or bug code versions are inaccurate), we keep these two applications in our benchmark suite as they both contain a large set of multi-threaded test cases.

Application	LoC	# Multi-thread tests	# Stars
ApplicationInsights	151.2K	156	0.5K
FluentAssertions	47.7K	41	2.5K
Kubernetes.Net	173.2K	21	0.7K
LiteDB	18.3K	7	6.2K
MQTT.Net	27.1K	126	2.2K
NetMQ	20.7K	101	2.3K
NpqSQL	51.9K	283	2.4K
NSubstitute	17.9K	13	1.7K
NSwag	101.5K	18	4.9K
SignalR	51.8K	52	8.5K
SSH.Net	84.4K	117	2.8K

Table 3. Details about the set of open-source C# applications used to evaluate Waffle.

Note that, although we manually reproduced all 12 known bugs, we did **not** use this knowledge when evaluating Waffle or WaffleBasic. Specifically, we ran both tools using *every* multi-threaded test case in the test suites of each application and recorded the number of bugs exposed as well as how many runs it took to trigger them, statistics about delays injected, and overhead measurements.

Experiments Setup. We run each benchmark on a Windows 10 desktop machine with an Intel Core i7-8700 3.2GHz CPU, 16GB of RAM, and 1TB of disk space.

We use a near-miss window δ of 100 milliseconds for both Waffle and WaffleBasic — the default setting in Tsvd [32]. Similar to Tsvd, we also use 100 milliseconds as WaffleBasic's fixed-length delay value.

Finally, we repeat each experiment 15 times, to reduce measurement variations that could arise due to the probabilistic nature of our tools.

6.2 Bug-detection coverage

Waffle can trigger all 12 previously known MemOrder bugs, as well as 6 previously *unknown (i.e., unreported)* MemOrder bugs (18 bugs in total), using only inputs from the applications' test suites.

Note that none of these 18 bugs can manifest themselves without delay injection, even when we execute the corresponding bug-triggering inputs repeatedly 50 times. Moreover, some of the previously unknown bugs discovered by Waffle remained undetected for many months or even years (e.g. Bug-14). Additionally, Waffle can trigger 3 of the known bugs using a test case that was already available in the test suite before the issues were reported, indicating that

No	Application	Issue	Previously	Exec. time (ms) w/o	# of detection runs		Detection slowdown (×)	
		ID	known?	instrumentation	WaffleBasic	Waffle	WaffleBasic	Waffle
Bug-1	SSH.Net	80	Yes	2,464	2	2	1.4×	1.2×
Bug-2	SSH.Net	453	Yes	1,042	2	2	1.7×	1.6×
Bug-3	NSubstitute	205	Yes	437	1	2	3.3×	5.1×
Bug-4	NSubstitute	573	Yes	316	2	2	9.0×	4.4×
Bug-5	NSwag	3015	Yes	887	2	2	2.1×	1.8×
Bug-6	FluentAssertions	664	Yes	782	1	2	1.4×	2.7×
Bug-7	FluentAssertions	862	Yes	831	2	2	1.2×	2.5×
Bug-8	LiteDB	1028	Yes	495	-	2	-	4.9×
Bug-9	Kubernetes.Net	360	Yes	1,955	1	2	1.3×	2.0×
Bug-10	ApplicationInsights	1106	Yes	143	-	2	-	4.9×
Bug-11	NetMQ	814	Yes	18,503	5	2	5.1×	2.2×
Bug-12	NpqSQL	3247	Yes	1,097	-	4	-	6.9×
Bug-13	SignalR	n/a	No	952	-	2	-	1.3×
Bug-14	ApplicationInsights	2261	No	1,349	2	2	1.5×	1.3×
Bug-15	NetMQ	975	No	593	-	3	-	12.2×
Bug-16	MQTT.Net	1187	No	1,207	-	4	-	5.4×
Bug-17	MQTT.Net	1188	No	13,722	-	3	-	6.2×
Bug-18	Kubernetes.Net	n/a	No	1,494	2	2	2.5×	2.0×

Table 4. Detection results from Waffle and WaffleBasic (Basic). Waffle discovered 6 previously unreported bugs (the bottom 6). Four of these bugs manifest in the latest available major release version (March 30th, 2022) and are reported by us. The other two (Bug-13, 18) no longer surface in the latest builds. The slowdowns are based on the execution time of the bug-triggering input without any instrumentation. "-" indicates that WaffleBasic fails to expose the bug in 50 runs.

our tool is useful for finding hard-to-detect bugs in mature software.

In contrast, WaffleBasic exposes only 11 out of the 18 bugs. WaffleBasic cannot expose any of the other 7 bugs even after a significant number of detection runs (50 in our evaluation). This happens because WaffleBasic injects many more delays and allows much more delay interference than Waffle, as discussed in §4.

In theory, the number of runs required to expose a MemOrder bug could vary in different attempts due to the probabilistic nature of concurrency bugs. Therefore, we repeated our experiment 15 times. When we report that a bug can be detected in 1 or 2 runs, we make sure that this is the case in the majority of attempts (i.e., at least 10 out of the 15 attempts). Bugs that require more runs to expose tend to behave more non-deterministically. For those bugs, we report the median number of runs required to expose them (Table 4).

Previously unknown bugs. Among the 6 previously unknown MemOrder bugs exposed by Waffle, 4 have since been fixed by developers in the most recent releases.

Two out of these six bugs lead to use-before-initialization problems (Bug-13 and Bug-14). For example, Bug-14 in ApplicationInsights [39] happens because the constructor only manages to initialize the event handler field of the object (i.e., `this.buffer.OnFull`) before the "buffer event" occurs. At that point control is transferred to the event handler, which attempts to access one of the still uninitialized fields of the object, triggering a `NULL` reference exception.

The other four previously unknown bugs lead to use-after-free problems. For example, Bug-15 in NetMQ [43] happens because the underlying message queue gets disposed while the queue still stores messages that are currently being processed. Later on, when a worker thread attempts to dequeue a message that just finished processing, it triggers a `NULL` reference exception.

6.3 Bug-detection efficiency

For 14 out of the 18 bugs, Waffle reliably exposes them by running the corresponding test case twice. Specifically, the bug is reliably exposed in Waffle's first detection run after a preparation run. The remaining 4 bugs took Waffle 3 or 4 runs to trigger. This happens because NpqSQL, MQTT.Net, and NetMQ perform significantly more heap object accesses which, in turn, are the source of many more delay candidate locations for Waffle to sift through. Moreover, as shown in Table 4, Waffle incurs a 1.2×–5.1× slowdown (median, 2.1×) for these 14 bugs when compared with running the bug-triggering input without instrumentation. For 7 of those, the overhead is 2.0× or lower. This happens because the bug manifestation halts the detection run prematurely, thus the end-to-end time in these cases is similar to or much

App.	Base	WaffleBasic (%)		Waffle (%)	
	(ms)	Run#1	Run#2	R#1	R#2
Applic.	227	122	357	19	38
Fluent.	776	48	48	24	27
Kubernet.	2051	14	37	9	41
MQTT.Net	1768	TimeOut	TimeOut	13	332
NetMQ	1657	167	375	34	288
NpgSQL	1118	2818	2509	266	968
NSubst.	344	72	294	26	78
NSwag	995	12	56	14	51
SignalR	267	58	144	13	81
Ssh.Net	702	68	96	16	20

Table 5. Average overhead on all test inputs. (Base: the average run time of a test input without any instrumentation)

Application	WaffleBasic		Waffle	
	# Delays	Duration (ms)	# Delays	Duration (ms)
Applic.	2,475	247,500	475	7,212
Fluent.	448	44,800	43	167
Kubernet.	177	17,700	197	5,904
MQTT.Net	TimeOut	TimeOut	3,243	141,699
NetMQ	11,767	1,176,700	11,271	520,037
NpqSQL	246,477	24,647,700	123,166	4,535,586
NSubst.	575	57,500	78	1,083
NSwag	343	34,300	349	11,806
SignalR	861	86,100	513	11,342
Ssh.Net	829	82,900	506	10,126

Table 6. Cumulative number and duration of delays injected across all test inputs (one detection run for each input). TimeOut: most tests timed out due to excessive delay.

shorter than running the original test input twice without instrumentation.

The remaining 4 bugs take a longer time to get exposed (5.4×–12.2×), as they require more than one detection run to manifest. This happens because more delays get injected in each run due to the denser, much more frequent heap object accesses—a similar trend across all test inputs for these particular applications (Table 5). Note, however, that traditional race detection techniques routinely incur several times the slowdown (e.g. 5-10× [14, 32]).

In comparison, WaffleBasic takes the same number of runs or, in one case, more (Bug-11), while managing to expose only 11 bugs. This is actually surprising, as WaffleBasic starts delay injection from the first run, unlike Waffle which spends its first run for preparation, without injecting any delays. WaffleBasic was able to expose only 3 bugs (Bug-3, Bug-6, and Bug-9) in fewer runs than Waffle (i.e., in its first run). WaffleBasic requires more detection runs than Waffle for the remaining 8. Moreover, WaffleBasic incurs longer end-to-end bug-detection overhead than Waffle for 7 out of these 11 bugs. Overall, these findings justify our decision to dedicate Waffle's first run for preparation without any delay injection (§4.2).

Finally, Bug-7 is the only case where Waffle incurs more overhead than WaffleBasic, although both tools need the same number of detection runs to trigger it. This happens because WaffleBasic exposes Bug-7 close to the beginning of its second run, while Waffle only does so towards the end of its second run (i.e., its first detection run).

6.4 Detailed results

Overhead. Table 5 reports the average overhead that Waffle incurs on *every* multi-threaded test case in each application's test suite, for both its preparation and detection runs. We exclude LiteDB since it contains only a few multi-threaded test cases, as noted in Table 3.

Waffle incurs much less overhead than WaffleBasic for 8 applications, and similar overhead for the remaining 2 applications (Kubernetes.Net and NSwag). Particularly, for NSubstitute, NpgSQL, and ApplicationInsights, Waffle's detection runs (R#2) are more than twice as fast as WaffleBasic's. Furthermore, for MQTT.Net, a protocol communication application, WaffleBasic incurs so much overhead that most of the test cases timed out, which does not happen for Waffle.

The performance benefit of Waffle comes from its decision to analyze parent-thread causal relationships (§4.1) and its reliance on variable-length delays (§4.3). This is reflected by the significantly fewer delays injected and the much shorter cumulative (total) delay duration introduced by Waffle, as illustrated in Table 6.

For 7 out of 10 applications (i.e., except for Kubernets.Net, NetMQ, and NSwag), Waffle injects only one-tenth to about half the number of delays across all test inputs, compared to WaffleBasic. Since Waffle uses variable-length delays instead of a fixed 100-millisecond value like WaffleBasic, the cumulative delay duration Waffle injects is 5× less than WaffleBasic. Note that for multi-threaded programs the cumulative delay duration only indirectly affects the total execution time. Thus, although Waffle injects less cumulative delay than WaffleBasic for Kubernets.Net and NSwag, the overhead incurred by Waffle and WaffleBasic is similar. In all other cases, Waffle incurs much less overhead than WaffleBasic.

Overall, Waffle achieves reasonable performance for in-house testing. For the preparation run (R#1), Waffle incurs 9–34% average overhead across all applications except for NpgSQL; for the first detection run (R#2), Waffle incurs 20–81% average overhead for all applications except for NpgSQL, NetMQ, and MQTT.Net. These three applications allocate a large number of objects at run time. Even though Waffle achieves significant improvements over WaffleBasic, the delay density is still moderately high for these 3 applications.

	# bugs missed	slowdown over Waffle
no parent-child analysis (§4.1)	0	1.17x
no preparation run (§4.2)	4	1.84x
no custom delay length (§4.3)	1	1.03x
no interference control (§4.4)	6	1.41x

Table 7. Alternative designs detect fewer bugs with slower detection runs. (Baseline # of bugs and performance are from Waffle across all applications).

Benefit of every design point. Table 7 shows how different design points help Waffle's bug detection capabilities and performance. We measure the impact on the number of bugs exposed and overhead incurred, averaged across all test inputs for all applications when disregarding one of the four key design decisions discussed in §4 (e.g. w/o parent-child analysis means Waffle does not prune out parent-child thread causal relationships).

This experiment shows that every design point has its benefit. Among the 4, the decision of having a dedicated preparation run without delay injection (§4.2) and the decision of coordinating delays to avoid interference (§4.4) offer the biggest benefit in both bug coverage and performance. The other two are also helpful. For example, excluding parent-child causal analysis for NpgSQL slows down Waffle's detection runs by 1.73×, on average, across all test inputs.

False positives. Waffle has no false positives, as our tool only reports a bug after triggering it *and* once it observes the NULL pointer exception not handled by the target application.

False negatives. Although Waffle successfully detected all 12 previously known bugs as well as 6 previously unknown bugs in our benchmarks, it could still miss MemOrder bugs for several reasons. First, like all dynamic detection tools, Waffle's bug detection capabilities rely on test inputs. If a buggy code region is not exercised by the test set, Waffle cannot detect the bug. Second, similar to Tsvd, Waffle uses several algorithms that rely on physical time information, such as delay interference analysis, delay length analysis, near-miss tracking, and so on. Consequently, Waffle could non-deterministically miss some MemOrder bugs in the first few detection runs.

7 Related Work

Concurrency-Bug Detection. Some techniques aim to *predict* concurrency bugs that might occur in the future by analyzing memory accesses and synchronization operations executed in one monitored run [14, 45, 47, 52, 55]. Typically, they do not aim to report bugs without false positives. This is difficult to guarantee without actually observing how the software behaves under buggy timing conditions, particularly in large software systems.

Some prior work [7, 22, 35, 53, 64, 65] guarantees no false positives in its reporting of concurrency bugs, including of MemOrder bugs [7, 22, 65]. Many first identify bug candidates through a predictive bug-detection run [7, 35, 64, 65] or through static analysis [53] and then use one or more delay-injection runs to validate *each* bug candidate. These tools naturally require many more runs than Waffle. UFO [22] eliminates false positives from its predictive bug detection through sophisticated trace analysis and constraint solving, instead of delay injection. Unfortunately, the complexity of its analysis prevents it from analyzing long traces. Furthermore, all these techniques incur about 10× or even 100× slowdown during the predictive bug detection run and require precise knowledge about synchronization operations present in the target application.

A significant amount of research is dedicated to static data race detection [3, 11, 33, 49, 61, 63], including uncovering use-after-free bugs [2]. These tools require careful annotation of all synchronization operations and inevitably incur more false positives than those using dynamic program analysis. Nevertheless, they are orthogonal to the latter and thus complement tools similar to Waffle.

Systematic testing. Extensive research has been conducted on systematic testing. These techniques steer the program execution towards potential buggy interleavings within some bound [15, 17, 18, 29, 38], relying on various coverage metrics [4, 21], or offering certain probabilistic guarantees [5]. Despite finding concurrency bugs, they are not designed to minimize the number of runs necessary to expose bugs and bear the cost of controlling the thread scheduler. In contrast, Waffle is explicitly designed to find concurrency bugs in a small number of runs, instrumenting only the target binary.

Test Generation. A large number of tools have been proposed to synthesize inputs to expose concurrency bugs, mainly inside libraries [9, 48, 50, 51]. Typically, they rely on generating sequences of concurrent method calls to help find combinations that harbor bugs. This is orthogonal to Waffle, which re-purposes existing tests to uncover MemOrder bugs.

Recently, fuzzing techniques have also been applied to tackle this problem [24, 34]. Razzer [24] leverages fuzzing to effectively generate system-call sequences whose concurrent execution can help expose concurrency bugs. ConAFL [34] combines traditional timing perturbation with input fuzzing. Specifically, it inserts code snippets that adjust thread-schedule priority right after thread creation and around potential bug locations[3], similar to previous concurrency-bug detection

[3] To identify these locations, ConAFL uses static analysis that cannot scale beyond ten thousand lines of code.

tools [53, 65]. It then applies AFL [62] on the modified application to increase the fuzzer's capability of exposing concurrency bugs.

Causality Inference. Past research has explored how to automatically infer happens-before causality between send/receive messages for system performance [1, 10], network dependency analysis [8], or concurrency bug detection [31]. These tools need to observe a large number of runs to make robust inferences and hence cannot directly help Waffle to expose bugs after a small number of runs.

Memory Bugs. Different from MemOrder bugs, there are also *deterministic* use-after-free and use-before-initialization bugs that occur within one thread and do not require concurrent accesses or rare timings to manifest. These belong to the larger family of memory bugs [57] and can be detected by generic memory bug detection tools like AddressSanitizer [54] and Valgrind [44], as well as dedicated use-after-free bugs detectors [6, 19, 28, 36, 59]. Naturally, analyzing synchronization or exploring different thread interleaving is out of the scope of these tools. They cannot detect concurrency bugs such as MemOrder bugs unless the program is executed under the bug-triggering input many times and the bug-triggering timing occurs spontaneously.

8 Conclusion

This paper explores a new design point in the active delay injection space—a design point aimed at exposing MemOrder bugs efficiently and effectively. We start from the existing state-of-the-art and gradually move towards a novel approach that balances bug-exposing capabilities, cost, and practicality. We hope future research can rely on our experience to design other resource-conscious active delay injection tools for detecting concurrency bugs.

9 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments on the paper and the artifact, and Jia-Ju Bai for shepherding this work. We would also like to thank Guangpu Li for his invaluable help with sketching out an initial prototype. The authors' research is supported by NSF (grants CCF-2119184, CCF-2028427, CNS-1956180, CCF-1837120, CNS-1764039), the CERES Center for Unstoppable Computing, an Eckhardt Fellowship, and gifts from Microsoft and Meta.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.

[2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 255–268, USA, 2019. USENIX Association.

[3] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: Compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[4] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, page 206–212, New York, NY, USA, 2005. Association for Computing Machinery.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3):167–178, March 2010.

[6] Juan Caballero, Gustavo Grieco , Mark Marron, and Antonio Nappa . Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA 2012 Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 133–143. Association for Computing Machinery, July 2012.

[7] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 706–717, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 117–130, USA, 2008. USENIX Association.

[9] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 266–277. IEEE Press, 2017.

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

[11] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 237–252, New York, NY, USA, 2003. Association for Computing Machinery.

[12] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 151–162, USA, 2010. USENIX Association.

[13] Jb Evain. Mono.cecil. Retrieved October 4, 2022. https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil.

[14] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. volume 44, page 121–133, New York, NY, USA, jun 2009. Association for Computing Machinery.

[15] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 415–431, USA, 2014. USENIX Association.

[16] Sean Gallagher. Wcry ransomware worm's bitcoin take tops $70k as its spread continues, May 2017. https://arstechnica.com/information-technology/2017/05/wcry-ransomware-worms-bitcoin-take-tops-70k-as-its-spread-continues/.

[17] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 174–186, New York, NY, USA, 1997. Association for Computing Machinery.

[18] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

[19] Binfa Gui, Wei Song, and Jeff Huang. Uafsan: An object-identifier-based dynamic approach for detecting use-after-free vulnerabilities. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 309–321, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Egor Homakov. Hacking starbucks for unlimited coffee. May 2015. https://www.dailydot.com/unclick/starbucks-hack-unlimited-coffee-2015/.

[21] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 210–220, New York, NY, USA, 2012. Association for Computing Machinery.

[22] Jeff Huang. Ufo: Predictive concurrency use-after-free detection. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 609–619, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Joab Jackson. Nasdaq's facebook glitch came from 'race conditions', May 2012. https://www.computerworld.com/article/2504676/nasdaq-s-facebook-glitch-came-from--race-conditions-.html.

[24] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, 2019.

[25] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 582–598, New York, NY, USA, 2017. Association for Computing Machinery.

[26] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 406–422, New York, NY, USA, 2013. Association for Computing Machinery.

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

[28] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium*, NDSS '15, pages 101–115. USENIX Association, 02 2015.

[29] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. Samc: A fast model checker for finding heisenbugs in distributed systems (demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 423–427, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, pages 517–530. ACM, 2016.

[31] Guangpu Li, Dongjie Chen, Shan Lu, Madanlal Musuvathi, and Suman Nath. Sherlock: Unsupervised synchronization-operation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 314–328, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162–180, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Bozhen Liu and Jeff Huang. D4: Fast concurrency debugging with parallel differential analysis. page 359–373, 2018.

[34] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. A heuristic framework to detect concurrency vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 529–541, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS*, 2017.

[36] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Doubletake: Fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 911–922, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[38] Madan Musuvathi and Shaz Qadeer. Chess: Systematic stress testing of concurrent software. *Lecture Notes in Computer Science: Logic-Based Program Synthesis and Transformation*, 4407:18–41, July 2007.

[39] [n.d.]. Applicationinsights.net issue # 1106. Retrieved October 4, 2022. https://github.com/microsoft/ApplicationInsights-dotnet/issues/1106.

[40] [n.d.]. Class `CallContext`. Retrieved October 4, 2022. https://docs.microsoft.com/en-us/dotnet/api/system.runtime.remoting.messaging.callcontext.

[41] [n.d.]. Class `InheritableThreadLocal`. Retrieved October 4, 2022. https://docs.oracle.com/javase/8/docs/api/java/lang/InheritableThreadLocal.html.

[42] [n.d.]. Netmq issue # 814. Retrieved October 4, 2022. https://github.com/zeromq/netmq/issues/814.

[43] [n.d.]. Netmq issue # 975. Retrieved October 4, 2022. https://github.com/zeromq/netmq/issues/975.

[44] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

[45] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *PPOPP*, 1991.

[46] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. *SIGARCH Comput. Archit. News*, 37(1):25–36, March 2009.

[47] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[48] Michael Pradel and Thomas R. Gross. 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, page 521–530, New York, NY, USA, 2012. Association for Computing Machinery.

[49] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Trans. Program. Lang. Syst.*, 33(1), jan 2011.

[50] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 473–489, New York, NY, USA, 2014. Association for Computing Machinery.

[51] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 175–185, New York, NY, USA, 2015. Association for Computing Machinery.

[52] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, 1997.

[53] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 11–21, New York, NY, USA, 2008. Association for Computing Machinery.

[54] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[55] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[56] Yao Shi, Soyeon Park, Zuoning Yin, Shan Lu, Yuanyuan Zhou, Wenguang Chen, and Weimin Zheng. Do i use the wrong definition? defuse: Definition-use invariants for detecting concurrency and sequential bugs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 160–174, New York, NY, USA, 2010. Association for Computing Machinery.

[57] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.

[58] The ImmunEFI Tool. Bitswift race condition bug fix postmortem. September 2021. https://medium.com/immunefi/bitswift-race-condition-bug-fix-postmortem-588184b8b43e.

[59] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 405–419, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Brian Wickman, Hong Hu, Insu Yun, DaeHee Jang, JungWon Lim, Sanidhya Kashyap, and Taesoo Kim. Preventing Use-After-Free attacks with fast forward allocation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2453–2470. USENIX Association, August 2021.

[61] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. ICSE '18, page 327–337, New York, NY, USA, 2018. Association for Computing Machinery.

[62] Michał Zalewski. American fuzzy loop. https://lcamtuf.coredump.cx/afl/. Retrieved October 4, 2022.

[63] Sheng Zhan and Jeff Huang. Echo: Instantaneous in situ race detection in the ide. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 775–786, New York, NY, USA, 2016. Association for Computing Machinery.

[64] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas W. Reps. Conseq: detecting concurrency bugs through sequential errors. In *ASPLOS*, 2011.

[65] Wei Zhang, Chong Sun, and Shan Lu. Conmem: Detecting severe concurrency bugs through an effect-oriented approach. volume 45, page 179–192, New York, NY, USA, mar 2010. Association for Computing Machinery.