THE UNIVERSITY OF CHICAGO

EFFICIENTLY EXPOSING MEMORY ORDERING BUGS WITH ACTIVE DELAY INJECTION

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER DEGREE

DEPARTMENT OF COMPUTER SCIENCE

BY
BOGDAN-ALEXANDRU STOICA

CHICAGO, ILLINOIS
FEBRUARY 23, 2022

Copyright © 2022 by Bogdan-Alexandru Stoica

All Rights Reserved

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

ABSTRACT

Concurrency bugs are difficult to detect as they are synchronization problems that only manifest under rare timing conditions. Recent work demonstrates that active delay injection can be used to detect one type of concurrency bugs - thread-safety violations - with low overhead, high coverage, and no program analysis. However, whether such an approach can work with similar efficiency and coverage for other types of concurrency errors remains an open question.

In this paper, we explore whether this state-of-the-art technique can be applied to detecting other types of concurrency bugs. Particularly, we investigate a class of synchronization problems that appears between object usage and its initialization or disposal, referred to as MemOrder bugs. We first show experimentally that the current state-of-the-art delay injection approach leads to high overhead and low coverage for MemOrder bugs, as their unique nature cause high delay density and high delay interference. We then design Wafl, a tool that customizes the injection location finding and injection execution strategies to match the nature of MemOrder bugs. Our evaluation on 11 popular open source C# applications shows that Wafl can expose more bugs with less overhead than the current state-of-the-art.

CHAPTER 1

INTRODUCTION

Concurrency bugs are difficult to detect and time-consuming to diagnose, as they only manifest under rare timing conditions [6,23,28,30]. Many of them escape rigorous in-house testing and cause severe production failures [17,20,22,45]. Among the various approaches to detecting concurrency bugs before code release, *active delay injection* is particularly promising in its high detection accuracy. Recent work [30] demonstrated an active delay injection design that can detect one type of concurrency bugs, thread-safety violations, with low overhead and high coverage. However, whether this approach can work with similar efficiency and coverage for other types of concurrency bugs remains an open question.

As illustrated in Figure 1.1, *active delay injection* [13, 25, 30, 36, 43] identifies strategic locations in a program where concurrency bugs may exist and injects delays at run time to exercise rare timing conditions and increase the chance of exposing concurrency bugs. This approach naturally has high accuracy, as it reports a bug only *after* the bug manifests under the bug-triggering timing. However, it traditionally suffers from high overhead and/or low bug coverage, until recent work TSVD [30] shows some new design philosophies:

To identify injection locations, there is a need to perform happens-before analysis [26] to prevent inserting delays on correctly synchronized locations. This involves a cost trade-off. On the one hand, performing the happens-before analysis to prune delay injection points [36, 43] requires heavyweight analysis either statically or dynamically (i.e., 5x–10x slowdowns [15,30]). On the other hand, performing no happens-before analysis can result in too many injection points [13,25] resulting in high overhead or large number of runs. TSVD

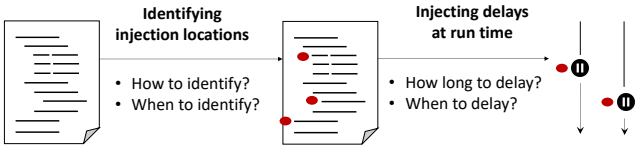


Figure 1.1: The workflow of active delay injection

resolves this trade-off by relying on easy-to-measure physical time to infer which locations are likely un-synchronized.

To carry out the injection, traditionally many bug detection runs are needed in exchange for a high detection coverage. Since previous work worried that too many program locations receiving delays in one run may lead to delay interference and too much overhead, delays are only injected at a small sampled set of candidate locations in each run and many runs are needed to reach good coverage [13, 25, 36, 43]. In contrast, TSVD squeezes many locations' delay injection, as well as the candidate-location identification, all into one run, and empirically shows that this new design can discover many bugs in just one or two runs, much more effectively than traditional detectors.

Although effective, the design of TSVD focuses on one specific type of concurrency bugs, thread-safety violations, and it is unclear whether its design works for other types of concurrency bugs.

This paper designs a lightweight active delay injection tool— WAFL — that effectively exposes an important type of concurrency bugs, refer to as MEMORDER bugs. These are bugs caused by the lack of synchronization between an access to a memory object and the initialization or destroy of the object. MEMORDER bugs are crucial to expose before software release, because they cause use-after-free or use-before-init errors, and potential memory corruptions and software crashes.

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

- *Location wise*, thread-safety violations can only occur at call sites of thread-unsafe APIs [30], while MEMORDER bugs can occur at any memory accesses to shared variables, which are much more common;

- *Timing wise*, exposing a thread-safety violation requires the execution windows of two thread-unsafe API calls to overlap, while exposing a MEMORDER bug requires a

memory access to occur before its corresponding initialization or after its corresponding destroy. They represent the two fundamentally different concurrency-bug timing conditions: atomicity violations for the former, and order violations for the latter [33].

We first designed WaflBasic that applies the Tsvd approach in a straightforward way. Specifically, instead of focusing on thread-unsafe API calls, WaflBasic monitors every read, write or method call related to heap objects. Thus, when formulating bug candidates, WaflBasic searches for use-before-init and use-after-free patterns, rather than thread-unsafe concurrent API calls. Otherwise, the original delay injection philosophy of TSVD remains unchanged.

Unfortunately, the efficacy of WaflBasic is limited. In our evaluation, WaflBasic injects delays at a much denser rate than TSVD, and struggles at both overhead and detection coverage. This experience revealed several design points that needs customization to better suit MemOrder bugs:

How to identify delay candidate locations? Our analysis of WaflBasic shows that the location property of MemOrder bugs presents too many program locations where bugs may exist for the inference algorithm in TSVD to effectively prune, leading to dense and useless delay injections.

Fortunately, while generic synchronization tracking is cumbersome and prohibitively expensive [29], we can complement the Tsvd technique with analyzing one type of synchronization—that caused by thread forks. This relationship can be inexpensive to track using a feature available in modern programming languages such as C# and Java. This is particularly helpful in shrinking the delay candidate set for MemOrder bugs by pruning out a significant fraction of well synchronized object initialization in parent threads. The number of delays injected at run-time can be greatly reduced by preventing needless attempts to reverse the execution order of memory access across fork boundaries.

When to identify candidate locations? To minimize the number of bug-detection runs, Tsvd combines its heuristic-based location identification into the same run as delay injection—

Design decisions	RaceFuzzer [43]	CTrigger [36]	RaceMob [25]	DataCollider [13]	TSVD [30]	WAFL
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 1.1: Different design in delay injection tools. (*: partial analysis is done; n/a: not applicable due to sparse delay injection.)

after a program location l is identified, a delay injection is considered the next time when l is executed in this run. We found this strategy not suitable in WaflBasic, as the dense delay injection of WaflBasic severely affects the efficacy of location identification which relies on physical-time information. Furthermore, Tsvd's strategy often cannot help exposing MemOrder bugs in one run, as many of their candidate locations like object initialization/destroy only execute a small number of times, or even once, in each run.

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

How long is the delay? TSVD uses a fixed length delay for all candidate locations. This strategy together with the denser bug-candidate locations necessary for MemOrder bugs results in an unfortunate trade-off between longer delays required for bug-exposing capability and shorter delays for lower runtime overhead.

In comparison, injecting delays with different lengths at different locations is more suitable 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 bug-detection runs, Tsvd injects delays at candidate locations with high probability[1], compared with previous work, and allows multiple threads to pause at the same time. Unfortunately, due to the location property of MemOrder bugs, this strategy leads to much more severe delay overlapping in WaflBasic, with many delays cancelling each other's effect. Such delay interference and cancellation occurs almost deterministically to some MemOrder bugs due to their unique timing properties. Clearly, more careful coordination during delay injection is needed for MemOrder bugs,

Guided by these insights, we build Wafl that applies active delay injection to MemOrder bugs with new designs at all the above key design points. Given a program under

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

test, WAFL runs the program once to identify candidate injection locations with light-weight happens-before analysis. In the following runs (typically just one more run is needed), WAFL carries out delay injection with carefully designed delay length and delay-or-not decision making guided by the information collected from the first run.

We have evaluated WAFL on 11 popular open source applications. Our experiments reveal that WAFL successfully exposes 18 MEMORDER bugs, including 12 previously known bugs and 6 previously unknown bugs. For most of these bugs (15 out of 18) and without any prior knowledge of the bugs, WAFL manages to reliably discover and trigger the bugs, with failure symptoms exposed, 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 WAFL exposes more bugs with much less overhead than various alternative designs.

CHAPTER 2

BACKGROUND

TSVD is an active delay injection tool that aims to expose thread-safety violations. Empirically, it reported significantly more bugs ($1,000+$) with much less overhead (less than 40%) than traditional delay injection techniques [30]. Since our goal is to adapt its delay injection philosophy to MEMORDER bugs, we provide more details about TSVD below.

How to identify delay candidate locations? Given a program binary, TSVD first instruments program locations where thread-safety violations may occur, namely call site of thread-unsafe APIs [30]. Later, at run time, TSVD collects information at these instrumentation points and leverages two heuristics to maintain a set S of thread-safety violation *candidates*. Each candidate is in the form of a pair of program locations $\{\ell_1, \ell_2\}$: API calls at ℓ_1 and ℓ_2 may contribute to thread-safety violations. Consequently, ℓ_1 and ℓ_2 are candidate locations for TSVD to inject delays, in order to expose potential thread-safety violations. We will refer to S as the *candidate set*.

The first heuristic — *near miss tracking* — *adds* candidate pairs to S based on constraints related to the threads involved, objects being accessed, and physical time stamps. 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 S iff $obj_1 = obj_2$, $thd_1 \neq thd_2$ and $|\tau_1 - \tau_2| \leq \delta$, for a time gap threshold δ. 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 — *HB inferencing* — *removes* candidate pairs from S that are unlikely to trigger thread-safety violations based on delay injection feedback. Assume TSVD added a candidate pair $\{\ell_1, \ell_2\}$ to S. When a delay is injected before location ℓ_1 in thread 1, TSVD checks whether it causes a proportional slowdown before location ℓ_2 in thread 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 S.

When to identify candidate locations? To minimize the number of bug-detection runs, TSVD carries out the above two heuristics to dynamically update S in the same run as it injects delays. After adding $\{\ell_1, \ell_2\}$ into the set of candidate pairs, TSVD injects a delay before ℓ_1 immediately at the next opportunity, when ℓ_1 is exercised again in the same run.

How long is the delay? TSVD relies on fixed-length delays. Specifically, whenever a delay is to be injected, TSVD injects a `sleep` of δ milliseconds. The configuration of δ balances the performance and bug-exposing capability: 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 reveal a thread-safety violation, the probability of injecting a delay before ℓ_1 in the future drops by a small constant λ. When its probability reaches 0, all candidate pairs involving ℓ_1 are removed from 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 few candidate-locations are delayed, thus many runs are needed to thoroughly search for bugs.

Finally, TSVD injects delays aggressively and allows multiple threads to be blocked in parallel, in order to reduce the number of runs needed to expose thread-safety violations. Although delays injected simultaneously could overlap and thus cancel each other's effect, the scarcity of candidate-locations related to thread-safety violations, combined with the probability decay scheme avoid interference in most situations [30].

CHAPTER 3

WAFL BASIC: APPLYING THE STATE-OF-THE-ART TO MEMORDER BUGS

We designed WaflBasic strictly following the design of Tsvd [30]. The main difference is that WaflBasic needs to consider different program locations in order to expose MemOrder bugs. However, the delay injection philosophy of Tsvd remains unchanged.

3.1 How to identify delay candidate locations?

Instrumentation sites WaflBasic first instruments every program location where a MemOrder bug may occur. Specifically, WaflBasic instruments all the reads, writes, and method calls from/to heap-allocated objects in a target binary. For each operation, WaflBasic records the corresponding object ID, physical timestamp, the operation type, and the thread conducting the operation at run time.

At run time, an instrumented operation may be categorized into the following three types that are related to MemOrder bugs: object initialization, object disposal, and object use. An operation that changes the state of the underlying object from `NULL` to non-`NULL` is considered an object initialization; an operation that changes the state of the underlying object from non-`NULL` to `NULL` or makes an explicit call to the object's destructor is considered an object disposal; a method call or a field access made through an object, like `obj.foo()` or `obj.field`, is considered an object use.

Adding to the candidate set S. WaflBasic adapts the near-miss heuristic of Tsvd to match the characteristics of MemOrder bugs. Consider an object initialization (or object use) happens at location ℓ_1, from thread thd_1, on object obj_1, at time τ_1, and another object use (or object disposal) happens at location ℓ_2, from thread thd_2, on object obj_2, at time τ_2. WaflBasic adds $\{\ell_1, \ell_2\}$ to S as a candidate of use-before-init MemOrder bug (or a use-after-disposal MemOrder bug) if these conditions are met: $obj_1 = obj_2$, $thd_1 \neq thd_2$,

$\tau_2 - \tau_1 < \delta$ (δ is the size of the near-miss window).

Here, $\{\ell_1, \ell_2\}$ forms a MemOrder bug *candidate*. Given the timing condition of MemOrder bugs, ℓ_1 then becomes a candidate location for WaflBasic to inject delays in order to expose the potential MemOrder bug. In other words, WaflBasic will inject delays before an object initialization, hoping to make it execute after the corresponding object use; WaflBasic will also inject delays before an object use, hoping to make it execute after the corresponding object disposal.

Removing from the candidate set S WaflBasic similarly adapts the happens-before inference heuristic of Tsvd. Given a potential MemOrder bug between two memory accesses occurring at locations ℓ_1 and ℓ_2, respectively, WaflBasic 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 the pair is removed from S.

3.2 What are other design decisions?

The remaining design of WaflBasic follows that of Tsvd.

When to identify candidate locations? WaflBasic injects delays in the same run as it adds and removes pairs from the candidate set S, hoping to facilitate exposing MemOrder bugs in a minimal number of runs.

How long is the delay? Similar to Tsvd, WaflBasic injects delays of a fixed length δ, which is set to be 100 milliseconds by default as in TSVD.

When to inject delays at run time? Similar to Tsvd, WaflBasic injects a delay at any location in the candidate set with a probability, which starts at 100% and gradually decreases towards 0 if no MemOrder bugs can be uncovered there. Also similar to Tsvd, WaflBasic 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 3.1: 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 well did WaflBasic do?

We evaluated WaflBasic using 11 open-source applications with 12 previously reported MemOrder bugs (the details are in Table 6.1 and 6.2).

Our experiments found that, although WaflBasic can expose some MemOrder bugs, it fails to expose others even after many runs and also incurs large overhead for several applications. Particularly, we observed several properties of WaflBasic that reflect the fundamental difference between MemOrder bugs and thread-unsafe violation bugs. These properties will lead our design of Wafl in Section 4.

Too many program locations in play. Due to the different location properties between MemOrder bugs and thread-safety violations, WaflBasic faces much denser instrumentation sites than TSVD (i.e., heap operation locations versus thread-unsafe API call sites). Indeed, for the 8 applications listed in Table 3.1[1], WaflBasic's instrumentation sites are more than 10 times more common than TSVD's in most cases. With this bigger base to start with more program locations tend to pass the near-miss heuristic at run time and get added to the delay candidate set S. As also shown in Table 3.1, WaflBasic's delay injection locations are more than 10 times more than TSVD's in most cases.

1. The public version of TSVD cannot instrument the other 3 applications in our benchmark suite.

Too much delay overlap. To quantify the overlap we run every test suite for the benchmarks in Table 6.1 and compute the complement of the ratio between the projection of all delays over the total delay value injected. This way, if no delays overlap, the value is is 0 while if all delays overlap the ratio is close to 1 (i.e., $\frac{D-1}{D}$, with D representing the total number of delays injected at run time). For TSVD, the average overlap ratio is less than 1% for 9 out of the 11 applications, with the remaining 2 applications having 12% and 15% average ratio, respectively. In contrast, for WAFLBASIC, the ratio is $2-28\%$, with 3 application above 25%.

Too few dynamic instances. A main reason why TSVD is able to detect many thread-safety violations in just one run is that most thread-unsafe API calls are executed for many times in each run, offering many chances for bug manifestation [30]. Unfortunately, this is not true for MEMORDER bugs. In particular, many objection initialization operations naturally execute a small number of times per run. In our measurements, the median number of dynamic instances of all object initialization operations is 2 across all applications we evaluated.

CHAPTER 4

WAFL: A CUSTOMIZED DELAY INJECTION TOOL FOR MEMORDER BUGS

To tackle the challenges faced by WaflBasic, we designed Wafl. As illustrated in Figure 4.1, Wafl first runs the targeted program binary once, referred to as the *preparation run*, without any delay injection and analyzes the run-time information to identify an initial set of delay candidate locations and to determine the delay length at each location. In subsequent runs, referred to as *detection runs*, Wafl carries out the delay injection using information collected during the first run and the real-time feedback from currently injected delays. In the following, we describe Wafl's design decisions and how they differ from WaflBasic one by one.

4.1 How to identify delay candidate locations?

What went wrong in WaflBasic? To identify candidate locations, Tsvd and hence WaflBasic completely disregards traditional happens-before analysis and instead uses run-time heuristics (near-miss window and happens-before inference) to *infer* what operations may be un-synchronized and hence should be part of the candidate set S. Unfortunately, this design did not work well for WaflBasic, affecting detection overhead and bug coverage, for several reasons.

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

Second, the happens-before inference can be less accurate in WaflBasic due to delay overlaps. Tsvd and WaflBasic infer a happens-before relationship between two locations ℓ_1 and ℓ_2, if injecting a delay on thread 1 before ℓ_1 causes a proportional slowdown on thread 2 before ℓ_2. However, if another delay is injected in thread 2 around the same time as and

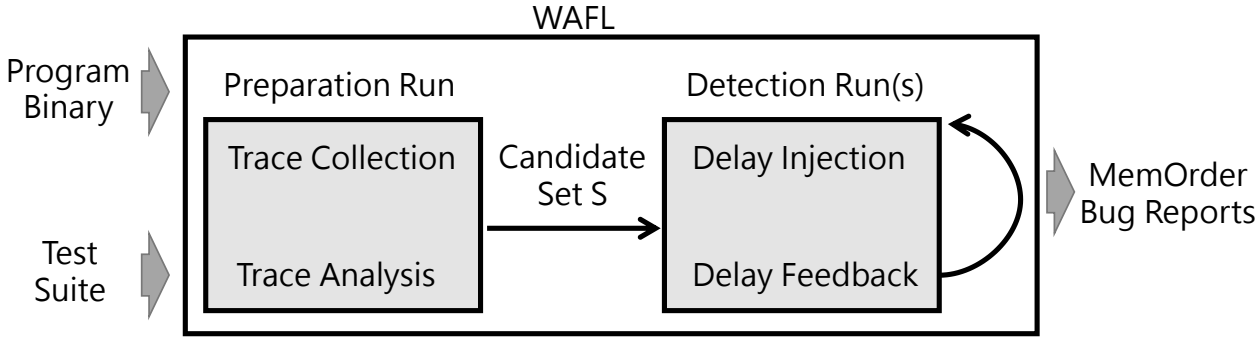


Figure 4.1: Workflow diagram of WAFL.

thus overlapping with the one before ℓ_1, happens-before inference cannot be conducted as we cannot reliably determine whether the slowdown on thread 2 is caused by a synchronization operation or is solely the effect of the second delay. Consequently, the more delay overlap, the less effective happens-before inference is.

Finally, even when the run time happens-before inference is as accurate as for TSVD, it offers little help to those locations that only execute a small number of times per run, such as those object initialization operations. By the time WAFLBASIC infers the happens-before relationship, the related program location no longer gets exercised in that run.

Design of Wafl At the first glance, WAFL 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 a high overhead cost incured by the happens-before analysis itself [29, 30].

Fortunately, we found that a sizable fraction of MEMORDER bug candidates are causally ordered by a specific type of happens-before relationship — that between parent and child threads/tasks. Typically, this happens because many objects are allocated in a parent thread before the worker threads are spawned. Consequently, WAFL supplements WAFLBASIC with parent-child relationship analysis. Pruning these ordered MEMORDER candidates during the preparation run reduces the number of candidate program location where WAFL can inject delays. As Table 6.4 shows, failing to remove them impacts the performance of our tool of

1.17× on average. However, the impact on the more memory-intensive application is much larger (e.g., 1.73× for NpgSQL, §6.4)

To track this parent-child relationship, traditionally we need to instrument every place where the parent forks a child thread or task. This is actually challenging in modern languages such as C# that have many ways for thread/task creation. Consequently, instead of instrumenting various types of thread/task forks, WAFL leverages a special type of thread-local storage that automatically gets copied from a parent to all child threads at the moment of thread creation, called logical call context (LCC)—a language feature supported by many modern languages such as C# and Java [10, 11].

Using the LCC mechanism, WAFL tracks happens-before relationships induced by thread/task forks by implementing vector clocks on top of the logical call context object. In each thread's LCC, WAFL creates a thread-local vector clock that is represented as a set of tuples $\{(tid_1, rcount_1), (tid_2, rcount_2), ...\}$, with each tuple representing a thread ID and a reference to the corresponding logical time counter. When a child thread is created, the LCC (and the vector clock) of the parent is automatically propagated to the child thread, at which point WAFL creates a vector clock of the child thread based on the clock of the parent thread. We design the vector clock object's constructor that (1) appends a tuple $(tid_k, 1)$, with tid_k being the child thread ID, to the vector clock content copied from the parent thread; and (2) increments the logical counter of the parent using the counter reference that got passed through the logical call context, making the parent's vector clock consistent again.

With these thread-local vector clocks in place, before WAFL adds a pair $\{\ell_1, \ell_2\}$ into the candidate set S, it will makes sure not only that $\tau_2 - \tau_1 < \delta$ but also that the vector clock of tid_1 at the moment of τ_1 is concurrent with the vector clock of tid_2 at the moment of τ_2.

4.2 When to identify candidate locations?

What went wrong in WaflBasic? The design decision of combining delay-location identification and delay injection into the same run does not benefit detecting MEMORDER bugs

as much as it benefits detecting thread-safety violations, because program locations involved in many MemOrder bugs only have few dynamic instances, as mentioned in Section 3.3. For these program locations, since they execute for only few times, or not at all, after been identified as delay candidate locations, whether to start delay injection immediately in the same run or to wait until the next run makes little difference.

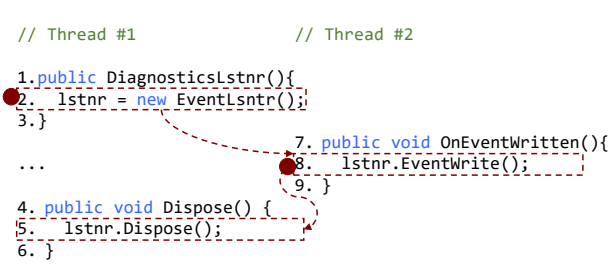
Furthermore, our evaluation has observed that the injected delays sometimes interfere with delay-location identification, which relies on physical time information. For example, a pair of delay-candidate locations $\{\ell_1, \ell_2\}$ observed during a delay-free run may disappear once delays are injected, as delays injected between the execution of ℓ_1 and ℓ_2 may stop 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 Wafl Wafl decides to conduct a delay-free run for planning purpose (i.e., the first run illustrated in Figure 4.1) before delay injections in subsequent runs. In the first run, Wafl uses the near-miss heuristic together with the parent-child relationship based pruning to establish a set of delay candidate locations S. In later runs, delays will be conducted at these locations, while the HB-inference heuristic continues to prune out locations from S that are unlikely to lead to bugs. In addition, Wafl also leverages the delay-free environment during the first run to collect timing-sensitive information that helps guide delay injection, which we will elaborate on in the next two sub-sections.

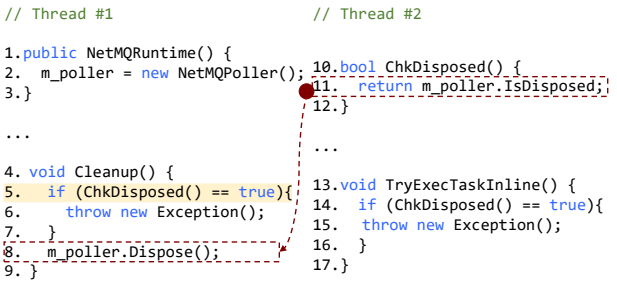
Note that, the use of a delay-free run can potentially increase the cost of Wafl, as at least two runs are needed to expose a MemOrder bug now. We believe the benefit outweighs this extra cost, and we will experimentally validate this in Section 6.

4.3 How long is the delay?

What went wrong in WaflBasic? WaflBasic struggles at finding a delay length that can balance performance and bug-exposing capability. Comparing with Tsvd, WaflBasic incurs much larger overhead under the same delay-length setting due to the much larger



(a) ApplicationInsights Issue #1106 [3]: Interfering bugs example



(b) NetMQ Issue #814 [1]: Interfereing candidate-locations example

Figure 4.2: Examples of delay interference

delay candidate set as discussed in Section 3.3. For example, under the default 100 milliseconds delay-length, TSVD only incurs 15%, 9%, and 11% overhead while running all the test suites of ApplicationInsights, FluentAssertion, and Kubernets.Net [30]. In comparison, WAFLBASIC incurs over 100% overhead for the set test suites of these three applications (Table 6.3). We could lower this overhead by using a much shorter delay length, but the bug exposing capability would drop. For example, decreasing the delay length from 100 down to 10 milliseconds would speed up the average performance of WAFLBASIC by about 4 times across all multi-threaded test inputs of the application NetMQ. Unfortunately, the known MEMORDER bug in NetMQ, which could be exposed by WAFLBASIC under 100 milliseconds setting, could no longer be exposed under the 10 milliseconds setting even after many runs.

Design of Wafl To address this challenge, WAFL leverages the observation that different bugs have different time gaps between corresponding operations in bug-free runs (i.e., the time gap between an object initialization and its use; or between an object use and its destroy) and hence injects delays of different lengths at different locations. Specifically, for the 10 known bugs, our measurement shows their time gaps to range from less than $1ms$ to around $100ms$. 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 that at ℓ_3 during delay-injection runs.

During the preparation run, when WAFL observes a program location ℓ_1, such as an object use, to execute shortly after its conflicting operation ℓ_2, such as an object init, WAFL

not only adds ℓ_1 into the candidate location set S, but also adds the time gap $\tau_1 - \tau_2$ as part of the record of ℓ_1, denoted as len_{ℓ_1} into S. Later on, when ℓ_1 is executed again, WAFL again measures the time gap between ℓ_1 and its most recent conflicting operation ℓ^* (ℓ^* may or may not be the same as ℓ_2), if the time gap is smaller than the near-miss threshold and yet larger than the time gap recorded in S, the record len_{ℓ_1} is updated with this longer gap.

Later on, during delay injection runs, the length of the delay injected at a location ℓ^* is proportional to the time-gap record of it in S: $\alpha \cdot len_{\ell}$, $\alpha \geq 1$ (by default, WAFL sets α to be 1.25). In our experiments, the length of the injected delays ranges from 1 to 100 milliseconds.

4.4 When to inject at run time?

What went wrong in WaflBasic? As discussed in Section 3.3, WAFLBASIC experiences much more delay overlap than TSVD. These overlapped delays significantly interfere with each other and cause WAFLBASIC to miss some true bugs with almost 100% probability, as in these two scenarios:

Interfering bugs. Sometimes, two bug candidates' manifestation interferes 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 both bug candidates, delay injection cancels each other. Unfortunately, these cases are particularly common when exposing MEMORDER bugs, as the manifestation conditions for use-before-init and use-after-destroy often interfere with each other.

Figure 4.2a illustrates a MEMORDER bug in ApplicationInsights [3]. The bug manifests when the constructor fails to allocate `lstnr` before a `WRITE` event invokes the `OnEventWritten()` handler. WAFLBASIC consistently misses this bug because it injects delays both before the allocation at line 2 in thread 1, which aims to push the allocation after the object use (line 8), and before the object use at line 8 in thread 2, which aims to push the use after the de-allocation (line 8). WAFLBASIC blocks both threads in parallel for the same duration,

and cannot trigger the bug even after 50 runs.

Interfering dynamic instances. Sometimes, WaflBasic injects a delay at a location ℓ_1, hoping to make it execute after location ℓ_2. Unfortunately, this delay repeatedly gets canceled out by a delay at another dynamic instance of ℓ_1, which unfortunately is executed right before ℓ_2.

Figure 4.2b illustrates such a MemOrder bug in NetMQ [1]. The failure happens when a connection is abruptly terminated causing several shared objects to be disposed (e.g., `m_poller` on line 8 of thread 1) while other threads are still processing network packages (e.g., `m_poller` on line 11 of thread 2). To trigger this bug, WaflBasic injects a delay right before line 11, aiming to push the use of `m_poller` there in thread 2 to execute after the dispose in thread 1. Unfortunately, since line 11 is also executed right before the dispose in thread 1 under a different call context, both threads always get delayed at around the same time for the same amount of time which prevents WaflBasic from exposing the bug even after 50 runs.

Design of Wafl A naive solution to this delay-interference problem is injecting only one delay in every test run, like previous work [36, 43]. However, this would require too many testing runs, as Wafl routinely observes tens of location-candidates after the analysis run of just one input. A better solution might be to change WaflBasic to avoid any parallel (i.e., overlapping) delays. However, according to Tsvd, completely avoiding parallel delays may cause some bugs to take many runs to expose. This could be even worse for MemOrder bugs: if the delay location of a true bug only executes for once or twice in a run, it may never get exposed if another delay location happens to execute right before it from another thread.

The high level idea of Wafl's solution is to enhance WaflBasic so that a delay planned to be injected before ℓ_1 is skipped when an *interfering* delay is ongoing. Here, we consider a delay planned for ℓ^* in thread t_2 to interfere with another delay planned for ℓ_1 in a different thread t_1, if two conditions are met, as illustrated in Figure 4.3: (1) ℓ^* executes before ℓ_2 in

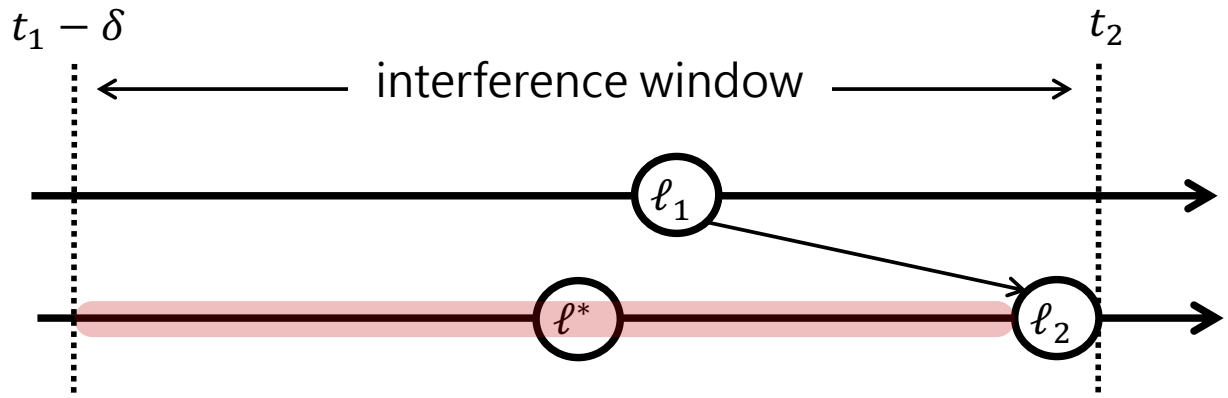


Figure 4.3: Illustration of delay interference. The interference window— when a concurrent delay injected in thread 2 can cancel the effect of that injected before ℓ_1— is highlighted.

the same thread, so that $\{\ell_1, \ell_2\}$ happens to be a bug candidate that we aim to expose by the delay at ℓ_1—the delay at ℓ^* blocks the thread of ℓ_2, essentially canceling out the attempt of making ℓ_1 execute after ℓ_2; (2) ℓ^* executes shortly ahead of ℓ_1 or right between ℓ_1 and ℓ_2—otherwise, the interference is negligible.

A challenge in realizing the idea above is that we cannot predict whether ℓ_2 will be executed in the thread of ℓ^* at the moment when ℓ_1 is to be executed and hence we cannot predict for sure which delays might interfere. To address this challenge, we leverage the delay-free analysis run. During the delay-free analysis run, as part of the near-miss heuristic checking, when the execution reaches ℓ_2 and identifies ℓ_2's conflicting operation ℓ_1 inside the near-miss window, WAFL further checks if any other delay candidate location (e.g., ℓ^* in Figure 4.3) has been executed in the same thread as ℓ_2 not too long ago (i.e., after $t_{\ell_1} - \delta$). If so, that location ℓ^* is added to the interference location set of ℓ_1 as part of ℓ_1's record in the delay candidate location set S. Later on, during delay injections, a delay at ℓ_1 is skipped if one of the interference locations of ℓ_1 has an ongoing delay.

CHAPTER 5

IMPLEMENTATION

We have implemented WAFL to find MEMORDER bugs in .NET applications (e.g., C#). WAFL has three key components: (1) the instrumenter which statically instruments the target binary, (2) the trace analyser which constructs the candidate-location set S, and (3) the runtime which implements the delay injection algorithm. Although our implementation is .NET-specific, we believe the algorithms are generic enough to be implemented for different programming languages or test/build frameworks.

WAFL *instrumenter*. This takes an application binary as input and wraps every heap memory access (i.e., read/writes to object fields, object method calls) in a proxy call. The proxy call transfers control to WAFL's runtime library that implements our logging and delay injection scheme (see below). To instrument the binary, we rely on a .NET instrumentation framework called Mono.Cecil [14].

WAFL executes the instrumented application in two phases. In the *preparation phase*, it runs the instrumented application to collect a runtime trace containing all heap memory accesses. No delay is injected in this preparation phase. In the *delay injection phase*, WAFL carefully injects delay to expose MEMORDER bugs.

WAFL*'s trace analyzer*. This analyzes a run time trace to identify pairs of memory accesses likely to cause MEMORDER bugs. At this stage WAFL constructs the candidate set 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 S. Finally, it identifies which candidate locations interfere with each other if delays are to be injected concurrently.

WAFL*'s runtime*. This implements the core of our logging and delay injection algorithm.

In the preparation phase, the runtime logs all heap memory accesses along with metadata

such as timestamps, accessed object id, access types (creation/read/write/dispose), etc.

In the delay injection phase, WAFL's runtime injects delays according to the delay planning done in the preparation. It also conducts heuristic-based happens-before inference and delay-probability decay to remove locations from the candidate set S that are unlikely to be buggy. This updated candidate set S, as well as the updated delay probability, are propagated from one detection run to the next.

Finally, WAFL reports a bug only when a `NULL` reference exception related to its delay injection occurs. 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.

CHAPTER 6

EVALUATION

Application	LoC	#M-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 6.1: Benchmark applications.

6.1 Methodology

Benchmarks. We evaluate WAFL on 11 popular open source C# applications from GitHub (Table 6.1). We picked these applications by searching in Github for C# applications that (1) are popular, measured by the number of Github stars; (2) contain well-maintained test suites, which will help us conduct systematic evaluation; and (3) contain confirmed and clearly described MEMORDER bugs in their issue tracking systems. For the last item, we first searched for keywords such as "data race" or "race condition" in the issue tracking systems; in the resulting issue reports, we then searched for keywords such as "exception" or "crash"; finally, we manually read the issue reports to see if they are about MEMORDER bugs, with bug-triggering inputs provided.

Following these MEMORDER bug reports, we were able to manually reproduce 12 previously known MEMORDER bugs in 9 applications (the top 12 bugs in Table 6.1). They will help us evaluate the bug-exposing coverage of WAFL and other alternative designs. 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 still keep these two applications in our benchmark suite, as they both contain a large set of multi-threaded test cases.

Note that, although we have manually reproduced all the 12 known bugs, we do **not** apply our knowledge about these bugs when we evaluate WAFL and other tools. Specifically, we will apply WAFL and alternative designs of WAFL to *every* multi-threaded test case in each application's test suite, and record how many bugs are exposed, how much slowdowns are incurred, and so on.

Experiment Setting. We run the benchmarks on a Windows 10 desktop machine with Intel Core i7-8700 3.2GHz CPU, 16GB of memory and 1TB SSD.

We repeat each performance measurement 5 times, and report the average. Moreover, WAFL and WAFLBASIC use a near-miss window δ of 100 milliseconds, the default setting in TSVD [30]; WAFLBASIC uses 100 milliseconds as its fixed delay length, as in TSVD [30].

6.2 Bug-detection coverage

WAFL uncovers all the 12 previously known MEMORDER bugs, as well as 6 previously unknown MEMORDER bugs from the 11 applications (18 bugs in total), using inputs from the applications' test suite.

Note that, none of these 18 bugs can manifest themselves without delay injection, even when we execute the corresponding bug-triggering inputs for 50 times. Some of the previously unknown bugs discovered by WAFL have remained undetected for many months or even years (e.g., Bug-14). Additionally, WAFL can trigger 3 of the known bugs using a test case which was already available in the test suite before the issues were reported, indicating that WAFL is useful in finding hard-to-detect bugs in mature software.

In contrast, WAFLBASIC exposes only 11 out of the 18 bugs. WAFLBASIC cannot expose any of the other 7 bugs even after many delay injection runs (50 in our evaluation). This happens because WAFLBASIC injects many more delays and allows much more delay

interference than WAFL, as discussed in Section 4.3 and 4.4.

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 for 15 times. When we report that a bug can be detected in 1 or 2 runs, we make sure that is the case in the majority of attempts (i.e., in 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 in Table 6.2.

6.3 Bug-detection efficiency

For 14 out of these 18 bugs, WAFL reliably exposes them by running the corresponding test case twice. That is, the bug is reliably exposed in WAFL's first delay injection run after a preparation run. The remaining 4 bugs took WAFL 3 or 4 runs to expose. This happens because NpqSQL, MQTT.Net, and NetMQ perform significantly more heap memory accesses, which in turn presents many more delay candidate locations for WAFL to search through.

For these 14 bugs, WAFL's bug detection imposes 1.2X–5.1X slowdown (median: 2.1X), comparing with running the bug-triggering input without any instrumentation, as shown in Table 6.2. For 7 of them, the slowdowns are 2.0X or lower: since the bug's manifestation ends the delay-injection run prematurely, the end-to-end time in these cases are similar or much shorter than running the original test input twice without any instrumentation. The remaining 4 bugs take a longer time to get exposed (5.4×–12.2×), as they require more than one delay-injection run to manifest. This is because more delays get injected in each run due to the denser heap memory accesses in these applications—a similar trend across all their test inputs, as shown in Table 6.3.

In comparison, WAFLBASIC takes the same number of runs or, in one case more (Bug-11), to expose only 11 bugs. This is actually surprising, as WAFLBASIC starts delay injection from the first run, unlike WAFL that spends its first run for preparation without any delay

injection. For only 3 bugs (Bug-3, Bug-6, and Bug-9), WaflBasic was able to expose them in fewer runs than Wafl (i.e., in its first run); for the other 8 bugs, WaflBasic requires more delay injection runs than Wafl does. As a result, WaflBasic incurs longer end-to-end bug-detection slowdowns than Wafl for 7 out of these 11 bugs, justifying Wafl's decision of dedicating the first run for preparation without delay injection (Section 4.2).

Finally, Bug-7 is the only case where Wafl incurs more slowdowns than WaflBasic with the same number of detection runs. The reason is that WaflBasic exposes this bug at the very beginning of the second run, while Wafl does not expose the bug until the end of its second run (i.e., its first delay-injection run).

6.4 Detailed results

Overhead. In Table 6.3, we report the average overhead of Wafl on *every* multi-threaded test case in each application's test suite, for both its preparation runs and delay-injection run. We skip LiteDB's results from Table 6.3, as it contains fewer multi-threaded test cases as shown in Table 6.1.

Wafl incurs much less overhead than WaflBasic in 8 applications, and similar overhead as WaflBasic in the remaining 2 applications (Nswag and Kubernetes.Net). Particularly, for NSubstitute, NpgSQL, and ApplicationInsights, Wafl's delay-injection runs (i.e., R#2) are more than twice as fast as WaflBasic's delay-injection runs. Furthermore, for MQTT.Net, a protocol communication application, WaflBasic incurs so much overhead that most of the test cases timed out. The performance benefit of Wafl comes from its decision of analyzing parent-thread causal relationship (Sec. 4.1) and using varied-length delays (Sec. 4.3).

Wafl achieves reasonable performance for in-house testing. For the preparation run (R#1), Wafl incurs 9–34% average overhead across all applications except for NpgSQL; for the first delay injection run (column R#2), Wafl incurs 20–81% average overhead for all applications except for NpgSQL, NetMQ, and MQTT.Net. These three applications create

a large number of objects at run time. Even though WAFL achieves significant improvement over WAFLBASIC, dense delay injections are still conducted for these 3 applications.

Benefit of every design point. Table 6.4 shows how different design points help WAFL's bug detection and performance results. We measure the impact on the number of bugs exposed and overhead incured, averaged across all test inputs in all the applications, when disregarding one of the four key designs discussed in Section 4 (i.e., w/o parent-child analysis means WAFL does not prune out parent-child thread causal relationships).

As we can see, every design point has its benefit. Among the 4, the decision of having a dedicated preparation run without delay injection (Sec. 4.2) and the decision of coordinating delays to avoid interference (Sec. 4.4) offer the biggest benefit in both bug coverage and performance. The other two designs are also helpful. For example, for NpgSQL, skipping parent-child causal analysis would slow down WAFL's delay injection runs by 1.73× on average across all test inputs.

False positives. WAFL has no false positives, as WAFL only reports a bug after it triggers the order violation *and* observes a resulting null-dereference not handled by the application.

False negatives. Although WAFL successfully detected all the 12 previously known bugs in our benchmark suite, as well as a few previously unknown bugs, it could definitely miss MEMORDER bugs for several reasons. First, like all dynamic detection tools, WAFL's bug detection capability relies on test inputs. If a buggy code region is not exercised by the test suite, WAFL cannot detect the bug. Second, like TSVD, WAFL uses several algorithms that rely on physical time information, such as its delay interference analysis, delay length analysis, near-miss windows, and so on. Consequently, WAFL could non-deterministically miss some bugs in the first few delay injection runs.

No	Application	Issue ID	Previously known?	Exec. time (ms) w/o instrumentation	# of detection runs		Detection slowdown (×)	
					WaflBasic	Wafl	WaflBasic	Wafl
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 6.2: Detection results from Wafl and WaflBasic (Basic). Wafl discovered 6 previously unreported bugs (the bottom 6). Four of these bugs manifest in the latest available major release version (Dec 27th, 2021) and are reported by us. The other two (Bug-13, 18) no longer surface in latest builds. The slowdowns are based on the execution time of the bug-triggering input without any instrumentation. - indicates that WaflBasic fails to expose the bug in 50 runs.

App.	Base	WaflBasic (%)		Wafl (%)	
	(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 6.3: Average overhead on all test inputs. (Base: the average run time of a test input without any instrumentation)

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

Table 6.4: Alternative designs detect fewer bugs with slower delay-injection runs. (Baseline # of bugs and performance are from Wafl across all applications).

CHAPTER 7

DISCUSSION

7.1 Limitations

Our evaluation shows the WAFL's potential for detecting a broader class of concurrency bugs. Yet, the current prototype has a few limitations.

First, WAFL's efficiency is ultimately linked to the number of heap memory accesses performed by the target application. More memory accesses mean more potential candidate-locations, thus more delays get injected which, in turn, translates to more slowdown, as discussed in 6.3.

Second, WAFL's delay interference detection algorithm is neither sound, nor complete. WAFL relies on a greedy approach, namely skip injecting a delay at location ℓ if an overlapping delay at another location ℓ' is ongoing. This strategy is not designed to capture *all* sources of interference (e.g., accumulated delays) and could potentially increase the number of runs needed to expose bugs if some critical delays are withheld for the first few injection opportunities. However, we did not observe the latter scenario in our evaluation and it is likely a rare occurrence as delays are injected probabilistically due to the probability decay heuristic.

Finally, WAFL relies on C# runtime support to identify parent-child thread relationships at low cost, by essentially infering synchronizations determined by thread fork operations without having to explicitly monitor fork operations. To the best of our knowledge, no equivalent inexpensive runtime support exists for thread join operations. This creates an imbalance where more candidate-locations pertaining to object allocations get pruned during the (less expensive) analysis run, while more candidate-locations pertaining to object uses get pruned in the (more expensive) delay injection run(s).

7.2 Threats to Validity

Internal threats to validity. As described in §6.1, WAFL leverages existing tests suites shipped with the target application to expose MEMORDER bugs. Thus, our tool could potentially miss bugs that are not exercised by these inputs. Additionally, not all tests provided by developers exercise multi-threaded code, further narrowing bug coverage. Finally, WAFL incurs false negatives related to sources of delay interference and happens-before inferencing as discussed above.

External threats to validity. The 11 applications and 18 bugs in our evaluation (chapter 6) may not be representative of real-world applications. Overall, we made a best-effort attempt to select non-trivial open-source C# applications that are both popular and broadly used (Table 6.1).

CHAPTER 8

RELATED WORK

Concurrency-Bug Detection. Active delay injection is definitely not the only approach to detecting concurrency bugs.

Some techniques aim to *predict* what concurrency bugs might occur in the future by analyzing memory accesses and synchronization operations executed in one monitored run [15, 32, 35, 37, 42, 44, 47]. Since they have different goals from WAFL, they also have very different designs and characteristics. Specifically, they do not aim to report bugs without false positives, which is impractical to guarantee without actually observing how the software behaves under the buggy timing. Some of them [32, 47] appends the bug-detection run with bug-validation runs, where one or multiple delay injection runs are used to confirm *each* bug candidate report, which is much more expensive than WAFL. They require knowledge about what are the synchronization operations in the software, and typically incur 10X or more slowdowns in each bug-detection run in order to conduct synchronization analysis and make bug predictions.

There are also tools that aim to catch concurrency bugs at run time as they manifest [23, 24]. These techniques are orthogonal to the goals of WAFL, which aims to uncover bugs in build and testing environments, prior to software deployment.

A large amount of research is dedicated to static data race detection [4, 12, 31, 39, 46]. They requires careful annotation about what are synchronization operations and inevitably incur more false positives than dynamic tools, which is not suitable for the context of WAFL.

Test Generation. A large number of tools have been proposed to synthesize inputs to expose bugs inside concurrency libraries [8, 38, 40, 41]. Typically, they rely on generating sequences of concurrent method calls to help find those that harbor bugs. This is orthogonal to WAFL: our tool is designed to re-purpose existing tests to uncover memory order bugs.

Systematic testing. Extensive research has been conducted on systematic testing. These techniques steer the program towards potential buggy interleavings within some bound [16,

18, 19, 27, 34], relying on various coverage measures [5, 21], or offering certain probabilistic guarantees [6]. Despite finding concurrency bugs, these techniques are not designed to minimize the number of runs need for bug exposure and bear the cost of controlling the thread scheduler. In contrast, WAFL is explicitly designed to find concurrency errors in a small number of runs instrumenting only the target binary.

Causality Inference. Several works explore how to automatically infer happens-before causality between send/receive messages for system performance [2,9], or for network dependency analysis [7], or for concurrency bug detection [29]. These frameworks require observing a large number of runs to draw a robust inference, and hence cannot directly help WAFL in its context of exposing bugs with few runs.

CHAPTER 9

CONCLUSION

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

REFERENCES

[1] NetMQ Issue # 814. https://github.com/zeromq/netmq/issues/814.

[2] 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.

[3] ApplicationInsights. https://github.com/microsoft/ApplicationInsights-dotnet/issues/1106.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] CallContext Class. `https://docs.microsoft.com/en-us/dotnet/api/system.runtime.remoting.messaging.callcontext`.

[11] InheritableThreadLocal Class. `https://docs.oracle.com/javase/8/docs/api/java/lang/InheritableThreadLocal.html`.

[12] 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.

[13] 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.

[14] Jb Evain. Mono.cecil. `https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil`.

[15] 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.

[16] 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.

[17] 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/`.

[18] 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.

[19] 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.

[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] 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`.

[23] 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.

[24] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 344–360, New York, NY, USA, 2015. Association for Computing Machinery.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

[46] 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.

[47] 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.