# Automatically Reproducing Timing-Dependent Flaky-Test Failures

Shanto Rahman
*University of Texas at Austin*
Austin, United States
shanto.rahman@utexas.edu

Aaron Massey
*Google*
Boulder, United States
aaronemassey@gmail.com

Wing Lam
*George Mason University*
Fairfax, United States
winglam@gmu.edu

August Shi
*University of Texas at Austin*
Austin, United States
august@utexas.edu

Jonathan Bell
*Northeastern University*
Boston, United States
j.bell@northeastern.edu

***Abstract*—When developers run tests after making code changes, they may encounter test failures from flaky tests, which are tests that can non-deterministically pass or fail on the same version of code. Prior work has found "timing dependence" to be a top cause of this non-determinism, i.e., tests may pass or fail depending on the timing of asynchronous callbacks or different thread interleavings that can occur when thread executions run faster or slower relative to others. Similar to how one debugs and fixes normal test failures, developers need to be able to reliably reproduce flaky-test failures. However, many of these failures can be extremely unlikely to occur (e.g., failing only once out of 10,000 runs in prior work), making it costly for developers to reproduce the failures. We present FlakeRake, an automated approach for reproducing timing-dependent (TD) flaky-test failures by inserting well-placed sleep calls, which temporarily pauses one thread or task and allows another to overtake it. When applied to an existing dataset of known flaky-test failures, FlakeRake is able to reproduce the *exact* same failure at least once for 136 failures, whereas simply rerunning each test 10,000 times reproduces only 115 failures or rerunning the entire test suites 10,000 times reproduces only 127 failures. For each failure that can be reproduced, we find that FlakeRake can *reliably reproduce* (>50% of the time) 107 failures, while rerunning just the flaky test or the entire test suite could not reliably reproduce any failure. We also find that if a developer needs to reproduce a failure six or more times, using FlakeRake (including the one-time cost to search for sleep calls) takes *less* time to reproduce that many failures than continually rerunning just the flaky test. Lastly, we inspect the sleep locations that FlakeRake outputs and provide insights for how one should cope with TD flaky tests.**

## I. Introduction

After a developer makes a change to their code, the tests might fail, not because of any fault introduced in that change but because of *flakiness* [1], [2] – the tests are non-deterministic, and they can pass and fail regardless of any change. Flaky tests are a growing interest in research literature, with a wealth of new approaches [3]–[11] recently proposed to detect which tests might be prone to flaky-test failures. Meanwhile, reports from industry via blogs (e.g., Gradle [12], Fitbit [13], Saucelabs [14], and Thoughtworks [15]) and research papers (e.g., Apple [16], Ericsson [17], Facebook [18], [19], Google [20]–[23], Huawei [24], Microsoft [25]–[28], and Mozilla [29], [30]) highlight the difficulties that developers face when dealing with flaky tests. One concern highlighted in these reports is that, despite the increasing interest from the research community to *detect* and *root cause* flaky tests, simply detecting which tests might be flaky and root causing the flakiness category can still be inadequate, i.e., although a developer may know that a test is flaky and the category of the flaky test (e.g., concurrency), the developer still may not be able to recreate the environment (e.g., the particular thread interleaving) to reproduce the flaky-test failure. More specifically, several surveys [31]–[33] have found that the reproduction of flaky-test failures to (1) debug and understand the failures and (2) verify that any patches actually repair or mitigate the flakiness is one of the most difficult challenges related to flaky tests.

Given that modern continuous integration workflows may execute a single test hundreds or thousands of times per day, a flaky-test failure that rarely occurs (e.g., once in 1000 executions) will still fail at least once a day (resulting in a failed build that incorrectly requires a developer's attention) and be a challenge for a developer to reproduce. Alshammari et al. [5] studied flaky tests in 24 open-source projects, executing each test suite 10,000 times and identifying 811 flaky tests. We analyzed this dataset to determine how often each *unique* failure (by stack trace) occurred, finding a total of 1,167 unique failures, of which 737 (63%) occurred in fewer than 10 runs.

Reproducing infrequent flaky-test failures is often challenging, yet necessary. In fact, prior work [31] found that 77% of developers often run flaky tests multiple times when debugging a flaky-test failure to reproduce the failure, log different parts of code, and vary the context in which the test is run. To help detect and reproduce flaky-test failures, prior work [3], [4] has proposed tools for detecting and reproducing failures caused by test-order dependency (OD), by controlling the order of test execution. Although OD is a prominent cause of flakiness

This work was supported in part by NSF grant no. CCF-2100037 and CCF-2145774.

and many tools [3], [4], [34]–[36] have been developed to help with these flaky tests, it is not the most prominent cause of flakiness. In fact, when we run a state-of-the-art OD tool [3] on the projects in Alshammari et al.'s dataset [5], we find that only 12% (96) of the projects' 811 flaky tests are OD.

Prior work [1], [31] identified that one of the most prominent causes of flakiness is *timing-dependence* (TD), i.e., tests that depend on execution timing. Yet, there are no publicly available tools to help reproduce TD-test failures.

To address this problem, we present FlakeRake, an automated approach for reproducing TD-test failures. FlakeRake outputs configurations that a developer can use to run tests to more reliably reproduce TD-test failures, debug them, and fix them without needing to repeatedly rerun tests. We implement FlakeRake for Java and evaluate its efficacy in reproducing TD-test failures by applying it to the entire FlakeFlagger dataset, consisting of 811 flaky tests and 1,167 unique test failures. Compared to two state-of-the-practice baselines, we find that FlakeRake is more effective at reproducing the same failures (by matching stack traces) in the FlakeFlagger dataset. We conducted repeated trials with each approach to measure how *reliably* these failures were reproduced, finding that FlakeRake can reproduce most failures in at least 50% of the time. We evaluate FlakeRake's execution time in a use-case (e.g., a debugging session) where a developer needs to reproduce a given flaky-test failure multiple times. While there is a one-time cost to generate the configurations, reproducing the flaky failure is fast afterwards (adding a delay of just a few seconds per-test run). Specifically, we find that FlakeRake is overall faster at reproducing a failure six or more times compared to simply rerunning the test normally (as the test does not reliably fail every time it is run normally).

We inspect the configurations that FlakeRake generates to reproduce TD-test failures and find that certain code characteristics are more likely to be associated with these failures. Our findings can be used to help developers further optimize tools to reproduce TD-test failures. We also study the flaky-test failures that were *never* reproduced in any of our experiments, finding several common root causes and yielding insights for future research. To enable others to use FlakeRake, we make the tool and our list of Java, timing-related APIs publicly available, along with the results of our evaluations [37].

This paper makes the following main contributions:

**Approach.** We present FlakeRake, an automated approach for finding failure-inducing configurations that reproduce TD-test failures. We implement our approach for Java and make our tool publicly available [37].

**Evaluation.** We evaluate FlakeRake on a dataset of flaky tests and find that FlakeRake is able to reproduce more flaky-test failures and reproduce them more reliably and faster compared to two state-of-the-practice techniques.

**Dataset.** We make publicly available a dataset of categorized TD flaky tests and how one can reliably reproduce the failures of these tests. Our dataset includes the failures, how often they occur, and the execution times of FlakeRake and the baseline approaches [37].

```
1  @Test public void testCustomBufferSize() {
2    startSMTPServer(NO_SSL);
3    configure(...);
4    logger.error("hello");
5    waitUntilEmailIsSent();
6    MimeMultipart mp = verifyMultipart("..." + this.
       getClass().getName() + " - " + msg);
7  }
8  MimeMultipart verifyMultipart(...) {
9    waitToReceiveEmails(1);
10   assertEquals(1, server.getMessages().length);
11 }
12 class SenderRunnable implements Runnable {
13   void run() { sendBuffer(...); } // sends e-mail
14 }
```

Fig. 1. Example TD test from `SMTPAppender_GreenTest` class in the `logback` project [38].

## II. Background

Prior work on flaky tests [1], [28], [31] have identified more than 10 causes for flaky tests, including dependency on asynchronous code, test order, thread interleavings, system time, etc. These pieces of work find that dependencies on asynchronous code and specific thread interleavings are the most prominent causes of flaky tests. We refer to all such tests as *timing-dependent* (TD) tests.

Figure 1 shows an example of a TD test from the FlakeFlagger dataset [5] that FlakeRake reliably reproduces. The goal of the test is to check whether emails are properly sent when the framework logs an error. `testCustomBufferSize` is flaky because of an assertion failure inside `verifyMultipart` on Line 10. The assertion checks whether the number of emails sent is `1` or not. Depending on the result of an asynchronous call, this test can fail with the message `AssertionError: expected:<1> but was:<0>`. The assertion sometimes fails because Line 4 logs an error and the configured email server from lines 2 and 3 begins to send an email about the error asynchronously in another thread, eventually calling Line 13. Line 9 on the main thread then waits up to five seconds for this other thread to send the email. If Line 13 takes longer than five seconds to finish, then the assertion on Line 10 fails. Based on the complexity of the operations that the test performs, it is possible for the test to take longer than five seconds to send the email. Prior work [5] found that this test failed only once in 10,000 test-suite runs. If a developer tries to debug this flaky test, the developer may need to run this test many times to reproduce the failure. When we run just this test 10,000 times, we find that this test *never* fails. On the other hand, FlakeRake can help reproduce the exact failure (same stack trace) observed in prior work by automatically suggesting the insertion of a sleep (a `Thread.sleep` call and value to use) right before Line 13. This suggestion reproduces the failure reliably (100% of the time in 10,000 runs).

## III. FlakeRake

Figure 2 shows an overview of *FlakeRake*, an approach to automatically reproduce TD flaky-test failures. At a high level, FlakeRake takes as input a known flaky test, and it outputs a *configuration*, which defines a set of locations along

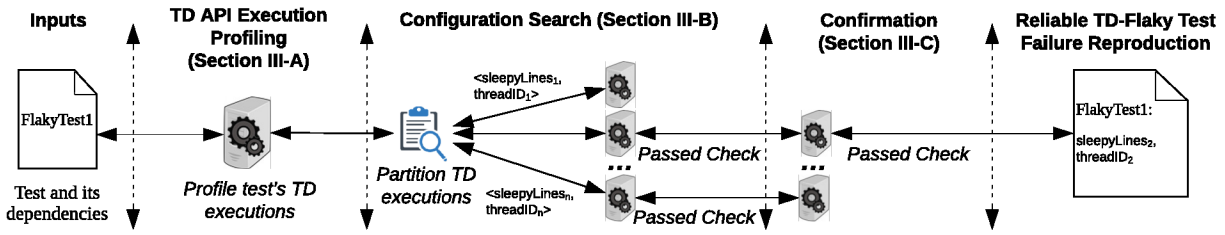


Fig. 2. Overview of FlakeRake's approach to automatically generate scripts to reliably reproduce timing-dependent flaky-test failures. FlakeRake profiles a test's usage of timing-dependent (TD) APIs, and then searches for failure-inducing configurations of *sleepyLines* for that test. These failure-inducing configurations are then confirmed, enabling reliable reproduction of flaky-test failures.

with a particular thread associated with each location where inserting delays (sleeps) at these location/thread pairs should reliably reproduce a specific TD failure. FlakeRake is meant to be used to reproduce a failure *after* a test is detected to be flaky. More details for when and how developers should use FlakeRake is in Section V-A. To generate configurations, FlakeRake has three main steps: Timing Dependent (TD) API Execution Profiling, Configuration Search, and Confirmation.

### A. TD API Execution Profiling

To reproduce TD flaky-test failures, FlakeRake inserts delays around calls to APIs that are timing-dependent. In general, TD APIs are those that implicitly affect timing-related operations, such as methods that read the system clock or acquire a lock (e.g., beginning of a Java synchronized block, which permits only one thread to enter the block at a time).

We construct a list of TD APIs for Java applications by collecting the API methods in the standard Java Development Kit (JDK) that may affect timing of code executions, especially across multiple threads or concurrent operations on data. We first inspected all APIs in the `java.util.concurrent` and `java.nio` packages. The former package contains a collection of classes with APIs for thread concurrency, while the latter contains a collection of classes with APIs for non-blocking, asynchronous IO operations. We also inspected all APIs that have the keywords "thread", "lock", "socket", or "time" in their API documentation from the following packages: `java.io`, `java.lang`, `java.net`, and `javax.net`. Our intuition is that such APIs are likely to affect thread executions or asynchronous services (e.g., network connections via sockets). In the end, we obtain 1955 APIs from `java.util.concurrent`, 565 from `java.nio`, 17 from `java.net` and `javax.net`, and 54 from other packages in the JDK (e.g., `System.currentTimeMillis`). These methods are the TD APIs we use. We further study the relevance of these APIs in our evaluation (Section IV-D). We store the list of TD APIs in a configuration file that developers can change as needed, and make our list publicly available [37]. We use only methods from the JDK as the TD APIs because any other methods from the tests, system-under-test, or third-party library code that perform timing-related operations are likely to eventually use these APIs from the standard JDK.

In its first phase, FlakeRake dynamically profiles the test's execution to find where it makes calls to any of the TD APIs. FlakeRake dynamically instruments the code to track calls to TD APIs, regardless of whether the calls are made directly or indirectly (e.g., by an external dependency) by the test. This step executes the test multiple times to record which APIs get called. The instrumented executions produce a list of *sleepyLine*s – the list of candidate lines to sleep at. As each TD API might be invoked in multiple threads, FlakeRake also records a threadID for each sleepyLine. We compute a stable identifier for the thread by generating a hash of the stack trace at the location that spawned the thread, the number of times that location has spawned a thread, and the threadID that spawned the thread. FlakeRake gives the initial, "main" thread the consistent identifier 0.

There may not always be the same consistent TD APIs called during a test execution due to inherent non-determinism of the execution. To mitigate the issues of non-determinism, FlakeRake executes this profiling step multiple times and aggregates the set of TD API calls made across all executions. When we perform this profiling step 10 times on the flaky tests in the FlakeFlagger dataset [5], we find that the first profiling run already contributes 84% of the TD API calls that can be found in 10 runs. By the fifth run, no TD APIs found by our profiling were needed to reproduce the flaky-test failures in the FlakeFlagger dataset. Therefore, we set and recommend the default number of executions for this step to be five.

### B. Configuration Search

While there are many possible TD API lines to sleep at to reproduce a TD failure (sleepyLines), only a few of them may be needed. FlakeRake explores these lines by repeatedly running the test under different configurations involving subsets of the sleepyLines. Although FlakeRake's process to repeatedly run the test can have a high cost, this cost is per test and not per failure, i.e., once FlakeRake produces a configuration, a test's TD failure can be trivially reproduced. Details on how the efficiency of FlakeRake compares to other approaches is in Section IV-C. If a test is found to fail under the initial configuration consisting of all sleepyLines, FlakeRake notes the stack trace of that failure, and it then attempts to minimize the set of sleepyLines, with the goal of finding a simpler configuration to reproduce that same failure. We implement and evaluate two heuristic-based algorithms to partition the set of sleepyLines into smaller sets:

***One-by-one (OBO)***: For each identified sleepyLine, OBO produces a configuration for each line. For example, given sleepyLines and corresponding threadIDs $(s1, t1), (s2, t1), (s3, t2)$, OBO produces three configurations: $< (s1, t1) >$, $< (s2, t1) >$, and $< (s3, t2) >$.

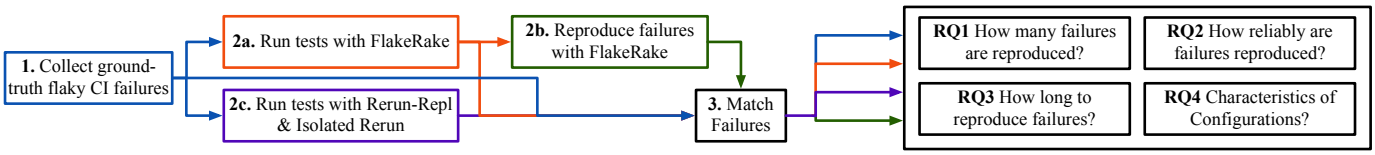


Fig. 3. Overview of evaluation methodology and research questions. Using a ground-truth list of flaky-test failures from the FlakeFlagger dataset, we use FlakeRake to identify failure-inducing configurations, and confirm that those configurations reproduce each failure 100 times. As baselines, we run each test with Isolated Rerun, and each test suite with Rerun-Repl. Each test failure is matched by stack trace, and we study the characteristics of each test and failure.

***Bisection***: Inspired by the bisection algorithm commonly used in mathematics [39] and similar to binary search (except the values need not be sorted), this algorithm identifies combinations of sleepyLines that could induce test failures by repeatedly bisecting the sleepyLine values and then selecting values under which the test outcome changes (i.e., changing from non-failure-inducing to failure-inducing). For example, given sleepyLines $(s1, t1), (s2, t1), (s3, t2)$ in order of appearance during execution, Bisection produces four configurations: $< (s1, t1), (s2, t1) >$, $< (s1, t1) >$, $< (s2, t1) >$, and $< (s3, t2) >$. FlakeRake's bisection will not explore subsets of configurations if the union of the sub-sets does not reliably reproduce a failure. This algorithm does not try configurations such as $< (s1, t1), (s3, t2) >$ and $< (s2, t1), (s3, t2) >$ to minimize the cost of configuration search. Future work should evaluate the trade-offs of OBO and bisection compared to other algorithms like delta-debugging [40].

Ultimately, we find that OBO finds slightly fewer failures than Bisection, while being only 2.4% faster than Bisection (Section IV-A provides more details). Given OBO's minimal time savings, we set FlakeRake to use Bisection as the default for our other experiments.

To determine whether a configuration is failure-inducing, FlakeRake performs an exploratory check for each of the configurations produced by the algorithms. For each sleepyLine in the configuration, FlakeRake injects a sleep call of $initSleepTime$ for the corresponding threadID. The sleep time at a sleepyLine is calculated as $\lfloor initSleepTime * (\frac{1}{2})^i \rfloor$, where $i$ is the number of times that specific sleepyLine had previously suspended the corresponding threadID's execution. Intuitively, the more times the same sleepyLine is executed, e.g., in a loop, the less we sleep at that location. To determine the $initSleepTime$, we performed a preliminary study where we explored $initSleepTime$ values of 5, 10, and 15 seconds on the FlakeFlagger dataset [5]. We find that $initSleepTime$ set to 5 and 15 seconds reproduces the same number (136) of failures from the FlakeFlagger dataset, while $initSleepTime$ set to 10 seconds reproduces one more failure (137). As $initSleepTime$ set to 5 seconds reproduces a similar amount of failures as the other values and a lower $initSleepTime$ means less time needed to search for configurations and reproduce failures, we set the default $initSleepTime$ to be 5 seconds and use this value for our experiments.

### C. Confirmation

For each failure-inducing configuration, FlakeRake reruns the test five times, and outputs the configuration only if it produced the same failure (i.e., same failure stack trace) at least three times. We rerun five times because prior work [41] found that rerunning a test five times is generally sufficient to observe both passing and failing results from a flaky test (Section IV-B shows our results rerunning 100 times).

FlakeRake might generate a failure-inducing configuration that results in a test getting stuck, e.g., in a deadlock or livelock. Similar to techniques used by mutation analysis tools, such as PITest [42], FlakeRake sets a per-test timeout based on the average run time of the test run normally, the total amount of sleep added, and a constant offset. If applying a configuration leads to the test execution time exceeding the expected time, then FlakeRake discards the configuration. FlakeRake can be configured to explore and confirm all failures (as we evaluate in Section IV), or to confirm only a specific failure that a developer is interested in debugging.

FlakeRake produces a *reproduction script* using the failure-inducing configurations outputted by the previous steps. This script includes (1) the failure, represented as the exception message and stack trace, and (2) the failure-inducing configuration, represented as a set of sleepyLines and their corresponding threadIDs for reproducing the specific failure. FlakeRake uses these scripts to dynamically apply the appropriate sleeps when re-running the test. Alternatively, developers can use the information to manually reproduce the failure.

## IV. Evaluation

Figure 3 shows a high-level overview of our evaluation methodology and research questions. Our experimental design takes as input a set of known flaky tests along with their respective logs generated by Alshammari et al.'s experiments [5]; we refer to this dataset as the FlakeFlagger dataset. These logs contain the failing exception and stack trace for each flaky-test failure, which serves as the ground truth of flaky tests and flaky-test failures that we aim to reproduce (Step 1 in Fig. 3). Their dataset was constructed by executing each test suite as developers would in a continuous integration environment – repeatedly running the test suite by executing `mvn test`. We run FlakeRake on each of the flaky tests (Step 2a) to output a list of failures and the configurations to reproduce them. Then we run each of the configurations 100 times to see whether the failures can be *reliably* (>50%) reproduced (Step 2b).

We consider two standard baseline approaches for reproducing flaky-test failures: Rerun-Repl and Isolated Rerun (Step 2c). *Rerun-Repl* is an exact replication of Alshammari et al.'s experiment [5], which invokes the entire test suite for each project 10,000 times. To reduce the differences in our replication, we use the same projects and scripts released by Alshammari et al. *Isolated Rerun* runs each flaky test 10,000

TABLE I
NUMBER OF FLAKY TESTS AND UNIQUE FAILURES DETECTED BY EACH APPROACH AND INTERSECTION OF THOSE FAILURES WITH RERUN FROM [5]. Unique failures are identified by stack trace. This table contains only the tests known to be flaky by Rerun from [5].

| | Flaky Tests & Failures by Technique | | | | | | | | | | Intersection of Failures with Rerun from [5] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Rerun from [5] | | Rerun-Repl | | Isolated Rerun | | FlakeRake | | FlakeRake-OBO | | | | | |
| Project | Tests | Failures | Tests | Failures | Tests | Failures | Tests | Failures | Tests | Failures | Rerun-Repl | Iso. Rerun | FlakeRake | FlakeRake-OBO |
| activiti-activiti | 32 | 32 | 15 | 15 | 6 | 6 | 28 | 95 | 28 | 72 | 15 | 6 | 24 | 24 |
| Alluxio-alluxio | 116 | 183 | 2 | 2 | 2 | 2 | 55 | 157 | 53 | 122 | 2 | 2 | 28 | 24 |
| apache-ambari | 52 | 53 | 0 | 0 | 0 | 0 | 1 | 3 | 1 | 3 | 0 | 0 | 2 | 2 |
| apache-commons-exec | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| apache-hbase | 145 | 250 | 42 | 42 | 72 | 121 | 89 | 165 | 73 | 91 | 13 | 14 | 15 | 6 |
| apache-httpcore | 22 | 22 | 5 | 5 | 2 | 2 | 16 | 22 | 16 | 17 | 5 | 2 | 4 | 1 |
| apache-incubator-dubbo | 19 | 21 | 8 | 8 | 1 | 3 | 6 | 29 | 6 | 27 | 0 | 0 | 0 | 0 |
| doanduyhai-Achilles | 4 | 4 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 |
| elasticjob-elastic-job-lite | 3 | 4 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | 2 | 0 | 0 | 0 | 0 |
| hector-client-hector | 33 | 33 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| jknack-handlebars.java | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| joel-costigliola-assertj-core | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| kevinsawicki-http-request | 18 | 18 | 18 | 18 | 0 | 0 | 0 | 0 | 1 | 1 | 18 | 0 | 0 | 0 |
| ninjaframework-ninja | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| orbit-orbit | 7 | 7 | 2 | 2 | 5 | 5 | 6 | 18 | 7 | 20 | 2 | 5 | 3 | 5 |
| qos-ch-logback | 22 | 23 | 2 | 2 | 2 | 2 | 16 | 22 | 16 | 22 | 2 | 2 | 12 | 12 |
| spring-projects-spring-boot | 163 | 287 | 0 | 0 | 30 | 31 | 5 | 24 | 6 | 14 | 0 | 24 | 0 | 1 |
| square-okhttp | 100 | 121 | 15 | 24 | 12 | 18 | 26 | 41 | 23 | 38 | 22 | 17 | 17 | 13 |
| tootallnate-java-websocket | 23 | 45 | 22 | 41 | 21 | 36 | 22 | 26 | 21 | 21 | 41 | 36 | 22 | 21 |
| undertow-io-undertow | 7 | 12 | 1 | 1 | 0 | 0 | 7 | 19 | 7 | 19 | 1 | 0 | 4 | 4 |
| wildfly-wildfly | 23 | 23 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| wro4j-wro4j | 16 | 23 | 2 | 2 | 1 | 1 | 4 | 5 | 4 | 4 | 1 | 1 | 4 | 4 |
| zxing-zxing | 2 | 2 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 2 | 2 | 0 | 0 |
| **Total** | 811 | 1,167 | 139 | 167 | 160 | 233 | 284 | 630 | 265 | 475 | 127 | 115 | 136 | 118 |

times in isolation and most closely mirrors what is typically done by developers when reproducing flaky-test failures.

The list of flaky-test failures found by each approach is passed into a matching script (Step 3), which considers two failures as matched if they have the *exact same* stack trace. We set FlakeRake's reproduction to require a failure be reproduced at least three times, therefore we also set a similar goal for the baseline approaches: an approach reproduces the failure if it reports the failure at least three times. We use this failure data to evaluate our research questions:

**RQ1:** How effective are FlakeRake, Rerun-Repl, and Isolated Rerun at reproducing flaky-test failures at least three times?

**RQ2:** How reliably can FlakeRake, Rerun-Repl, and Isolated Rerun reproduce flaky-test failures?

**RQ3:** How efficient are FlakeRake, Rerun-Repl, and Isolated Rerun at reproducing flaky-test failures multiple times?

**RQ4:** What are the characteristics of the failure-inducing configurations outputted by FlakeRake?

*Dataset and Environment:* We use the same FlakeFlagger dataset from Alshammari et al. [5]. We conducted our experiments using virtual machines, each running Ubuntu 20 and Oracle's Java 1.8.0_301, with 16GB RAM and 4 virtual CPU cores. We use this same environment to run all of our experiments (Rerun-Repl, Isolated Rerun, and FlakeRake).

### A. RQ1: Number of reproduced failures

Our first research question leads us to examine which known flaky tests could be reproduced as flaky by each approach, and how many unique failures we observe. Table I shows the number of tests detected as flaky and the number of unique failures detected by each approach. We find that the distribution of failures can vary dramatically between projects. That is, in some cases (e.g., `activiti`, `commons-exec`, `orbit`, `wildfly`), the number of unique failures matched the number of tests exactly: each flaky test always failed with the same failure stack trace. However, in other cases (e.g., `alluxio`, `hbase`, `spring-boot`), some flaky tests fail with many different stack traces.

Comparing the results from Rerun (Alshammari et al's data [5] - extracted from their Table 1 and supplemental data) with Rerun-Repl and Isolated Rerun, we observe that approximately 80% of the flaky tests from the FlakeFlagger dataset could not be reproduced as flaky. Given the non-determinism of flaky tests, it is understandably difficult to reliably reproduce them. Alshammari et al. conducted experiments several years before ours, so although we use the same versions of the same tests, SNAPSHOT versions of dependencies could have changed, resulting in different behaviors.

Comparing FlakeRake with Bisection partition (columns FlakeRake) and FlakeRake with OBO partition (columns FlakeRake-OBO), we find that the default Bisection partition was slightly more effective than the OBO partition. In total, the two partition schemes could together reproduce failures for 290 flaky tests. Overall, FlakeRake reproduced failures for more tests than Rerun-Repl or Isolated Rerun (284 compared to 139 and 160, respectively), and also reproduced more unique failures (630 compared to 167 and 233, respectively). FlakeRake's performance per-project was quite consistent, detecting at-least-as-many flaky tests as the baselines in 17 of the 23 projects.

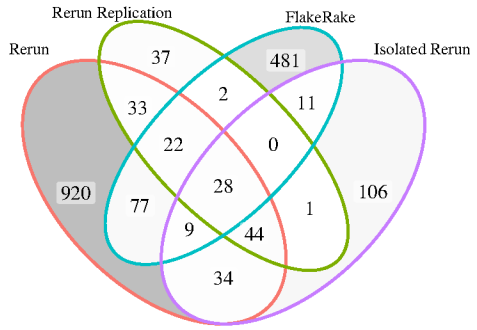


Fig. 4. Intersection of failures detected by each technique. Of the techniques evaluated, FlakeRake was most successful at reproducing failures from the original Rerun dataset (136 failures), compared to Rerun-Repl (127 failures) and Isolated Rerun (115 failures).

TABLE II
TOP 4 CATEGORIES OF FAILURES BY DETECTION SOURCE

| Detected By | Category | Number of Failures |
|---|---|---|
| Only Rerun | UnknownHostException | 224 |
| | AssertionError | 148 |
| | IOException | 144 |
| | ArtifactResolutionException | 104 |
| Only FlakeRake | AssertionError | 124 |
| | Timeout | 117 |
| | IOException | 113 |
| | ConcurrentUpdate | 67 |
| Rerun-Repl or Isolated Rerun (not FlakeRake) | Address in use | 60 |
| | IllegalArgumentException | 34 |
| | TableNotFoundException | 29 |
| | AssertionError | 28 |

Not only does FlakeRake reproduce more flaky-test failures than the alternative techniques, it also reproduces more of the exact *same* failures that appeared in the FlakeFlagger dataset [5]. Specifically, FlakeRake reproduced 136 failures, while Rerun-Repl reproduced 127 and Isolated Rerun reproduced 115 failures. This result shows that, despite perturbing the specified test behavior (by inserting sleeps), FlakeRake still reproduces *more* of the same failures that were reported in the original dataset than simply rerunning the tests normally.

Figure 4 is a four-way Venn diagram that shows the number of failures that were matched between each of the four flaky-test reproduction approaches that we presented in Table I. To gain more insights into our results, we categorize each of the failures in our dataset by the exception type, showing the top four exception types in Table II (our supplementary material contains the complete result). While some categories are somewhat generic (e.g., "AssertionError"), others provide more insights as to the failure cause, such as "UnknownHostException." Through this analysis, we find that many (472/787) of the failures from the FlakeFlagger dataset that we could not reproduce are caused by transient network or disk conditions ("UnknownHostException", "IOException" and "ArtifactResolutionException"). We also run a state-of-the-art OD test detector, iDFlakies [43], on the entire dataset with 100 random orders, identifying 96 OD tests in the FlakeFlagger dataset (which cannot be detected by FlakeRake or Isolated Rerun). Reproducing these non-TD failures is outside the scope of this work.

Of the failures detected only by FlakeRake, roughly 24% (117/481) were related to a test exceeding a timeout, which we expect FlakeRake to be effective at reproducing, given that it inserts sleeps. Other than generic "AssertionError"s, the largest category of failures produced by Rerun-Repl or Isolated Rerun, but not by FlakeRake are "Address in use" errors. These errors occur when a test binds to a free network port but races with the operating system that is working on releasing the port. These errors are most common when using Isolated Rerun, which repeatedly executes the same test in quick succession, preventing cleanups common in other approaches.

Note that FlakeRake should be used to reproduce all flaky-test failures even without knowing the test category, because FlakeRake is quick to know if a test can be TD, i.e., FlakeRake requires just one instrumented test run to know if a test involves multiple threads or not. In our evaluation, we evaluate FlakeRake on all flaky tests from the FlakeFlagger dataset, *without* knowing whether a test is TD or not, and we still find that FlakeRake reproduces more flaky-test failures than any baseline. Future work can incorporate flaky-test category predictions [44] to determine if a test is likely TD before even using FlakeRake.

### B. RQ2: Reliability of failure reproduction

Our RQ1 experiments find that there are 247 failures that either FlakeRake or the baseline approaches (either Isolated Rerun or Rerun-Repl) reproduced from the FlakeFlagger dataset. To evaluate FlakeRake's ability to create scripts that allow developers to more reliably reproduce flaky-test failures, we attempt to reproduce each unique failure (in the FlakeFlagger dataset that we reproduced in RQ1) 100 times. Specifically, we run FlakeRake and the baselines in their reproduction mode 100 times, and compare FlakeRake's reproduction rate with the reproduction rate of the best baseline for that failure.

For each configuration that FlakeRake's exploration phase reports as a likely candidate (failing at least three out of five runs) for reproducing a specific flaky-test failure, we measure what percentage of 100 runs can reproduce the same failure (its reproduction rate). For failures that either Isolated Rerun or Rerun-Repl reproduced in RQ1, we also run the corresponding baseline approach 100 times and measure how often the approach can reproduce that failure. This measure helps estimate how effective FlakeRake and the baseline approaches are for a developer aiming to *repeatedly* reproduce a flaky test for debugging (e.g., logging different parts of code [31], fault localization [45]) or validating patches.

Table III compares the reproduction rate of each failure for FlakeRake and the baseline approaches: each cell shows the number of failures that were reproduced by FlakeRake at the rate specified in the left column and also reproduced by the baseline approach at the rate specified in the top row. The table shows that Isolated Rerun and Rerun-Repl are *very* ineffective at reproducing failures, with most failures reproduced less than 10% of the time and only a handful between 10-50% of the time. More importantly, these baseline approaches cannot

TABLE III
COMPARISON OF FAILURE REPRODUCTION RATES FOR FLAKERAKE COMPARED TO THE BEST OF RERUN-REPL AND ISOLATED RERUN WHEN ALL APPROACHES ATTEMPT TO REPRODUCE FAILURES 100 TIMES. Rerun-Repl and Isolated Rerun had no failures with a rate greater than 50%.

| | Best of Rerun-Repl and Isolated Rerun | | | | |
|---|---|---|---|---|---|
| FlakeRake Repro | 0 | (0%, 10%] | (10%, 25%] | (25%, 50%] | Total |
| 0 | 25 | 61 | 23 | 5 | 114 |
| (0%, 10%] | 7 | 13 | 0 | 0 | 20 |
| (10%, 25%] | 1 | 2 | 0 | 0 | 3 |
| (25%, 50%] | 2 | 1 | 0 | 0 | 3 |
| (50%, 75%] | 3 | 5 | 0 | 0 | 8 |
| (75%, 99%] | 1 | 5 | 0 | 0 | 6 |
| (99%, 100%] | 35 | 52 | 4 | 2 | 93 |
| Total | 74 | 139 | 27 | 7 | 247 |

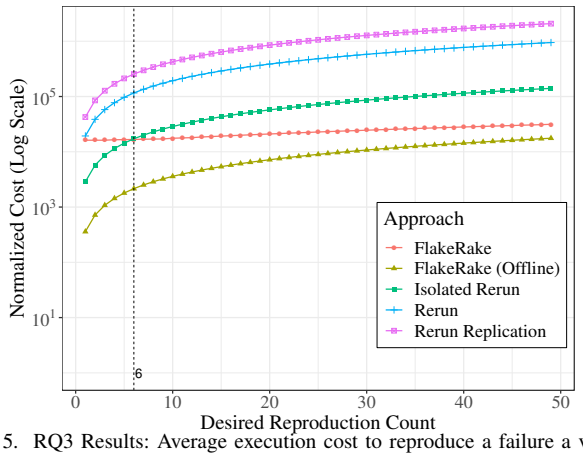


Fig. 5. RQ3 Results: Average execution cost to reproduce a failure a varied number of times with each approach. For each of the 28 failures detected by all five approaches, we compute the time to reproduce each failure $N$ times, normalized to the time to execute that test in isolation once. After 6 reproductions, FlakeRake is faster than Isolated Rerun.

reproduce any failure >50% of the time. In contrast, of the 136 failures that FlakeRake reproduced (Table I), 107 (8+6+93) of these failures are reliably reproduced >50% of the time. In fact, many (93) of these failures are *extremely* reliably reproduced – in over 99% of the 100 runs. We conclude that FlakeRake can *more reliably* reproduce failures than state-of-practice approaches, helping developers repeatedly reproduce TD failures for debugging and fixing.

### C. RQ3: Efficiency of reproducing failures multiple times

Reproducing a flaky-test failure multiple times is an integral step to debugging the failure [31], [32]. As such, we aim to understand the runtime cost needed to reproduce the same set of failures multiple times for tests known to be flaky.

For each of the 28 failures reproduced by all three approaches (Fig. 4), we compute the expected amount of times a developer needs to use that approach to reproduce the failure at least $N$ times, ranging $N$ from 1 to 50. For FlakeRake, this time is computed as the time to generate the reproduction script (namely, to search for the failure-inducing configuration), followed by the time needed to run with the failure-inducing configuration enough times to reproduce the failure at least $N$ times. Note that running the reproduction script involves using inserted delays, so each run is slower than when run normally, but each run is also more likely at reproducing a failure, requiring fewer runs to reproduce the same number of failures. We also consider the runtime cost of "FlakeRake (offline)", which excludes the up-front time needed to generate the reproduction script. Excluding the up-front time simulates the use-case where FlakeRake is executed in the cloud *before* a developer begins rerunning the test to observe the same failure multiple times as part of debugging (i.e., the "offline" refers to process of generating the reproduction script happening offline, not on the critical path of developer debugging). For the baseline approaches, we calculate the $E(N)$, the expected number of executions needed to witness $N$ failures, as the ratio of executions in which the approach reproduced that failure. Then, the time is computed as the time needed to run the test (or test suite, for Rerun-Repl), multiplied by $E(N)$. For each failure, we normalize the reproduction cost relative to the time needed to run that test a single time (in isolation), showing the normalized cost to reproduce each failure relative to how long it would take to run that test a single time. This normalization allows us to aggregate the time to reproduce some number of failures in a single figure for all approaches.

Figure 5 shows the relation between the number of desired reproduced failures and the normalized runtime cost to achieve that many failures for each approach. We see that FlakeRake (offline) is the fastest approach, regardless of the number of reproductions needed, by excluding the one-time cost of generating a reproduction script while still reliably reproducing a failure for each run.

If we disregard FlakeRake (offline), we see that using Isolated Rerun would be the fastest approach if the developer only needs to reproduce the failure fewer than six times. However, if a developer needs more than six failure reproductions, FlakeRake quickly surpasses Isolated Rerun in terms of performance, as each additional failure can be reproduced very quickly (as RQ2 showed, nearly every run with FlakeRake's reproduction script reproduces the failure). This finding suggests that if a developer wishes to reproduce a failure fewer than six times, it may be faster to just use Isolated Rerun, with the given risk that Isolated Rerun is the least likely to reproduce the failure at least once (reproducing 21 fewer failures in RQ1). A recent study [46, Table 4] also found that developers run tests an average of six times and a maximum of 32 times when debugging normal test failures that fail deterministically. Developers likely require a similar or even larger number of times to observe a failure when debugging flaky-test failures.

As developers cannot know a priori how many times they need to reproduce a failure before finalizing a patch, we recommend a combination of Isolated Rerun and FlakeRake. A developer can run FlakeRake in the background to generate a reproduction script while using Isolated Rerun to reproduce failures. If they still need to observe more failures for debugging, and FlakeRake has generated the reproduction script, the developer can switch to the reproduction script to more reliably and efficiently reproduce failures going forward.

When comparing Rerun (blue line) with Rerun-Repl (pink

TABLE IV
TOP FIVE TIMING-DEPENDENT (TD) APIS OCCURRING IN SLEEP CONFIGURATIONS THAT INDUCED FAILURES MATCHING THE FLAKEFLAGGER DATASET FAILURES. For each TD API, we show the number and percentage of tests in which that API was called, unique failures that the API reproduced, and failure-inducing configurations that were a part of the failures that were reproduced.

| Timing-Dependent API | Tests (%) | Failures (%) | Configs. (%) |
|---|---|---|---|
| ExitSyncBlock | 240 (85%) | 492 (79%) | 2706 (18%) |
| ExitSyncMethod | 239 (85%) | 436 (70%) | 2368 (16%) |
| EnterSyncBlock | 238 (85%) | 478 (76%) | 2648 (18%) |
| EnterSyncMethod | 226 (80%) | 426 (68%) | 2276 (16%) |
| System.currentTimeMillis | 219 (78%) | 369 (59%) | 2190 (15%) |

TABLE V
TOP FIVE TIMING-DEPENDENT APIS THAT CAN BE USED TO INDUCE THE SAME FAILURE, MATCHING THE FLAKEFLAGGER DATASET. For each pair of TD APIs, we show the Pearson correlation coefficient $R$, indicating the frequency with which two APIs co-occur. A score of 1 indicates that two APIs can always be interchanged.

| Timing-Dependent API | Timing-Dependent API | $R$ |
|---|---|---|
| ByteBuffer.allocate | ByteBuffer.array | 1.00 |
| EnterSyncBlock | ExitSyncBlock | .99 |
| EnterSyncMethod | ExitSyncMethod | .99 |
| Selector.wakeup | Thread.currentThread | .96 |
| System.currentTimeMillis | Thread.currentThread | .93 |

line), Rerun-Repl is projected to be *slower*. On average, we find tests failed more frequently in the FlakeFlagger dataset than in our Rerun-Repl, suggesting that reproduction would be faster using the environment that was used to build the dataset. We conducted Rerun-Repl with 16GB RAM per VM, while Alshammari et al. reported using 8GB RAM per VM [5]. We leave it to future work to further study the impact of system configurations on flaky-test failure reproduction [47].

### D. RQ4: Characteristics of configurations

To better understand why FlakeRake is effective, we study the characteristics of the failure-inducing configurations found by FlakeRake that reproduced failures from the FlakeFlagger dataset. We study the TD APIs methods (Section III-A) that are a part of the failure-inducing configurations.

Overall, we find 270 unique TD APIs that appear in the reported failure-inducing configurations. These 270 TD APIs are used in a mean of 201 and a median of 75 failure-inducing configurations. The most frequently occurring API is ExitSyncBlock, which represents the exit of a `synchronized` block, appearing in 2,706 or 18% of failure-inducing configurations. Table IV shows the top five TD APIs that are part of failure-inducing configurations. Our supplemental data archive includes each of the failure-inducing configurations that FlakeRake produced, in addition to a listing of the 270 unique TD APIs that appeared in those configurations [37]. Our results suggest that if developers are manually reproducing TD flaky-test failures, inserting sleeps around APIs such as ExitSyncBlock can help reproduce a failure in 85% of the tests for which FlakeRake can reproduce a failure.

Of the 2,591 TD APIs in FlakeRake's pre-configured list (Section III-A), we find that only 394 TD APIs were used in the tests we ran. Of these 394 TD APIs, we find that there are 124 APIs that FlakeRake found, but these APIs are not part of any minimal failure-inducing configuration. Future work might study how FlakeRake's results would be affected if we remove the 2,197 unused TD APIs and the 124 used but unnecessary APIs, as doing so may simplify FlakeRake's search process. Overall, the number of unique sleepyLines (call sites to TD APIs) invoked by any given test is on average 76.33. Manually investigating all of those call sites to understand whether they are related to a flaky-test failure can be incredibly time consuming. FlakeRake automates this process, generating minimized failure-inducing configurations. Our findings suggest that developers looking for a quick-fix before deploying FlakeRake might first try to insert sleeps around some of the top TD APIs in Table IV, such as ExitSyncBlock.

Beyond looking at statistics of individual TD APIs, we also study how often two APIs occur together (i.e., they both can independently reliably reproduce the same test failure). By understanding which APIs occur together, developers or tools can better prioritize which APIs to explore, depending on the APIs that have been explored already. Table V shows the five pairs of APIs from Table IV with the highest correlation score. With a correlation of 1, we find that ByteBuffer.allocate and ByteBuffer.array can *always* be interchanged — the same set of test failures can be reproduced by sleeping at both or just one of these API calls. Based on the failures observed and description of these methods, we find that they tend to co-occur in the same basic block. Hence, sleeping at one may have comparable effects to sleeping at the other.

To better understand some of the less obviously correlated pairs, we manually inspected the failures of one pair: System.currentTimeMillis and Thread.currentThread. We find that these two APIs are a part of failure-inducing configurations for 199 test failures. Of these 199 failures, we find that 53% of these failures are because FlakeRake slept in a logging framework (the `LoggingEvent` class from `log4j` [48]) used by the test and code under test. These failures originate from tests belonging to 14 different projects. When we further inspect, we find that the failures are likely unrelated to these APIs specifically, but the failures occur because the tests expect logging to only take some limited amount of time. This finding suggests that many flaky tests may be timing-dependent because they depend on asynchronous code (e.g., logging events) unrelated to the test code or code under test. In the future, we can optimize FlakeRake to look for certain library dependencies instead of just a pre-configured list of APIs found from searching in the standard JDK. Alternatively, future work can consider how to automatically refine this list for specific developers and their projects.

## V. DISCUSSION AND THREATS TO VALIDITY

### A. When and How to use FlakeRake

Many companies have management systems for dealing with flaky tests [8], [28], [49], [50] (e.g., Microsoft uses Flakes [28] to keep track of flaky tests), suppress their failures during continuous integration, and report flaky-test failures to

developers to be fixed later. When developers do have time to debug and fix flaky tests, they need to first reproduce the flaky-test failures, which is essential to debugging [31], [45]. Based on our findings, we suggest that developers attempting to reproduce flaky-test failures follow this workflow: 1) Determine if the test is TD or OD: run FlakeRake (to detect TD failures) and OD detection tools (to detect OD failures). Both of these tools are fully automated, and if they succeed, will produce a script to deterministically reproduce that failure. 2) As automated tools run in the background, re-run the test in isolation roughly six times. Continue to use isolated reruns until the test is sufficiently debugged and fixed. Our reproduction attempts found that isolation is not very effective at reproducing a failure even once (§IV-A), but when it can reproduce failures, isolation is efficient at doing so for a few failures (§IV-C). 3) If more failures are needed and the automated tools have produced a reproduction script, switch to the script to more reliably reproduce failures.

Our preliminary results suggest that FlakeRake can be useful not only for reproducing a flaky-test failure, but also for repairing flakiness. Specifically, prior work [17], [28] has investigated the use of delay injection for *reducing* flakiness, but the work often required rerunning tests many times to identify the delays needed to reduce flakiness. FlakeRake can be used to reduce the cost of prior work by having FlakeRake insert delays to reliably *reproduce* flaky-test failures so that prior work can more quickly derive the fix needed to reduce flakiness. Once a fix is found, there is no need for FlakeRake to insert delays anymore. We evaluate this idea by randomly sampling one test from each module of each project in our dataset for which FlakeRake could reproduce at least one failure. Our automated approach uses the flaky test reproduction script to reliably cause the test to fail and, while using the script, inserts a delay at each API call. We iteratively increase the delays until either the test reliably passes (five out of five trials), or we reach a maximum delay of 10 seconds.

Of the 20 tests that we examined, this prototype approach suggested a patch for 12 tests. Our prototype was incompatible with three of the tests and it could not find a patch in the code under test or test code for the remaining five. We plan to improve the compatibility of our prototype and expand its reach into library code in the future. We inspected the 12 patches and observed cases with simple read-after-write data races where one thread writes to an object while another thread reads from it. If the first thread does not write to the object within a certain timeframe, then another thread throws a `NullPointerException` or an assertion failure. In this scenario, FlakeRake finds the thread and location where inserting delays during the writing of the object would cause a failure to deterministically occur. The patch to reduce the flakiness, then, is effectively the dual of the reproduction script: inserting a delay before reading the object. We used this information to create patches for 12 tests. The patch for one test was closed without comments, the patch for another test was merged, with developers saying "LGTM, thanks", and developers suggested changes for the patches of two other tests. The results of our prototype showcases how FlakeRake can help not only reproduce failures but also help future work automatically repair TD flaky tests.

### B. Threats to Validity

As FlakeRake modifies code, one important question to consider is: does FlakeRake reproduce flaky-test failures that developers care about? There are many ways to unpack this question, and while perhaps the strongest evidence can be provided by a user study, we focus our experiments on reproducing flaky-test failures that prior work observed *without* applying any specialized flaky-test detection strategy. For example, the Shaker tool detects concurrency-related flaky tests by executing tests while placing a very heavy compute load on the CPU [9]. This approach might reveal many flaky-test failures that occur in extreme environments, which might over-approximate the set of flaky-test failures that would be witnessed by developers running their tests in a "normal" continuous integration environment. We carefully design our evaluation to avoid such problems by matching failure stack traces produced by reproducing flaky-test failures from a previously-collected dataset. The instrumentation that FlakeRake introduces for reproducing flaky-test failures may hide some existing failures. As such, our results may be a lower-bound on the number of reproducible flaky-test failures.

We were unable to find a dataset besides the FlakeFlagger dataset [5] that has flaky tests detected in a "normal" running scenario, along with the stack traces of each failure. While it was initially concerning to find that we were unable to reproduce many of the failures in the FlakeFlagger dataset, our detailed analysis has demonstrated that those failures were due to platform-related flakiness (e.g., DNS failures and disks running out of space). This fact about the dataset is supported by the publicly available failing test reports [51]. Future work may apply FlakeRake to other projects beyond Java.

Our evaluation results could also be biased due to faults in FlakeRake, or in the scripts that we wrote to collect and process the results. We have mitigated these threats through code review, with each component reviewed by an author of this paper who did not author that component. In critical steps of our analysis, we also employed differential testing, with different authors implementing the same high-level analysis, and then comparing the final outputs to ensure that analyses were correctly implemented. To aid future research, we include the code for FlakeRake, all of our intermediate and final results, and our evaluation scripts in our data archive [37].

## VI. Related Work

***Developer Reactions to Flaky Tests.*** Luo et al. [1], Eck et al. [31], and Gruber and Fraser [52] performed studies to better understand developers' perceptions of flaky tests. They found that "Concurrency" and "Async Wait" were among the most common causes of flakiness, accounting for up to roughly half of all flaky tests studied, and that flaky tests are time-consuming to debug and repair, because reproducing the failure can be difficult. Similarly, Habchi et al. [32] and Parry

et al. [33] performed a study on the sources, impacts, and mitigation strategies of flaky tests. They found that developers strongly needed help root causing and reproducing flaky-test failures. FlakeRake targets exactly this problem by helping reproduce TD flaky-test failures. FlakeRake could also be applied as a program understanding aid in debugging and fixing. Future work might continue on our prototype described in Section V-A by studying how FlakeRake can help developers understand, debug, and fix TD flaky-test failures.

***Flaky Test Reproduction.*** Microsoft's recent FlakeRepro work aims to reliably reproduce TD flaky-test failures [53]. Whereas FlakeRake uses lightweight heuristics to determine where to insert delays, FlakeRepro uses backward slicing. When FlakeRepro inserts a delay at a code location, all threads execute that same delay, whereas FlakeRake associates a delay to both a location and a specific thread. FlakeRepro is not publicly available and only implemented for .NET applications, and hence, we could not compare empirically to it. While FlakeRepro was evaluated on 31 concurrency-related tests, we evaluate FlakeRake on 811 flaky tests of unknown root causes, and compare its performance to several state-of-the-practice baseline approaches. Our evaluation also provides many useful insights, e.g., the duration of delays inserted, the locations where delays should be inserted, and the characteristics of the failures that could and could not be reproduced. Other tests might be flaky due to test-order dependencies, failing if the order in which tests are run changes. iDFlakies automates the process of identifying such dependencies and outputting test orderings that induce (reliable) test failure [3]. We use iDFlakies to filter out such order-dependent flaky tests, as FlakeRake is not designed to reproduce their failures. In the future, we may compare failures reproduced by FlakeRake with failures induced by flaky-test simulation tools [54].

***Flaky Test Detection.*** Some approaches aim to use machine learning to classify tests as flaky based on some large training set of known flaky tests [5], [8], [10], [11], [55]. The baseline approach to detect flaky tests is to *re-run* them and to check whether the outcome changes. Researchers have created tools that modify the execution environment to make flaky failures more likely to occur. For example, tools that aim to detect OD tests modify the order in which tests are run, or perform dataflow analysis to track data dependencies between tests [4], [34], [36], [56]. FlakeScanner [57] induces flaky, user-interface failures by scheduling non-deterministic (async) events such that each test run explores different event execution orders. FlakeRake may also be used to detect flaky tests by inserting delays, and it additionally can also help identify the least amount of additional sleep needed to more reliably reproduce flaky-test failures. Shaker aims to detect flaky tests by running many concurrent "stressor" tasks as tests are run [9]. Similarly, Terragni et al. proposed to run tests under various resource constraints [58]. FlakeRake differs from these approaches in that it can reliably reproduce flaky-test failures without changing the environment. Malm et al. [17] performed a study on open-source concurrent programs, identifying the role that delays play in avoiding flaky-test failures. Future work might combine the two approaches: using FlakeRake to determine locations to insert delays, and their work to improve the robustness of the inserted delays.

***Concurrency Bug Detection.*** "Controlled execution" is a classical testing methodology for concurrent programs by exploring different execution interleavings [59]. This approach has been applied for detecting concurrency bugs in Java applications [60]–[64] by forcing context switches. In an approach most comparable to FlakeRake, Eytani and Latvala applied a minimization approach to select a minimal set of context switches that reproduce a concurrency bug [65]. FlakeRake builds on the core concept of controlled execution, differing from prior work in that (1) we show the importance of inserting delays around TD APIs (discussed in Section IV-D) as opposed to just synchronization primitives, and (2) FlakeRake inserts timed sleeps, as opposed to prior work that aims to reproduce data races by inserting calls to `Thread.yield()`. Prior studies on flaky tests [1], [28], [31] have studied how often their failures indicate bugs in the test code, code under test, etc. and found that between 27%-34% of flaky-test fixes require changing non-test code. FlakeRake aims to reproduce failures from TD flaky tests, regardless of the bug source, and can therefore be helpful even in the presence of concurrency bugs in code under test.

Event race detectors, such as EventRacer [66] and CAFA [67], profile applications and analyze happens-before relationships. Not all races can result in incorrect program behavior, and hence, race detectors can be paired with a testing system that reproduces the race and observes if the race is harmful [68]. FlakeRake does *not* perform any happens-before analysis to detect which events might race with each other, instead it directly jumps to this testing-based approach to identify races that result in flaky-test failures. We also could not find a race detector for Java that scaled to the complexity of the projects in our evaluation. Reproducing flaky-test failures is fundamentally different from traditional concurrency bug detection, as a flaky-test failure does not necessarily indicate a concurrency bug in the system-under-test.

## VII. Conclusion

We attempted to reproduce 1,167 flaky-test failures from an existing dataset by re-running those tests 10,000 times. We found that these approaches reproduce flaky-test failures infrequently (typically occurring in fewer than 10% of executions), highlighting the difficulty developers face when they attempt to debug and fix flaky tests. We introduced FlakeRake to *reliably* reproduce timing-dependent flaky-test failures and implemented it for Java applications. We found that FlakeRake can reproduce more failures than the baseline approaches and can reproduce failures more reliably. In the future, we look forward to utilizing FlakeRake to fix flaky tests and conducting user studies to understand the impact of FlakeRake on debugging. Our supplemental data archive [37] contains the source code for FlakeRake, a dataset of timing-dependent flaky tests, and our experimental results.

## REFERENCES


[1] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE*, 2014.

[2] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *TOSEM*, 2022.

[3] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST*, 2019.

[4] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA*, 2014.

[5] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "FlakeFlagger: Predicting flakiness without rerunning tests," in *ICSE*, 2021.

[6] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in Python," in *ICST*, 2021.

[7] A. Shi, A. Gyori, O. Legunsen, and D. Marinov, "Detecting assumptions on deterministic implementations of non-deterministic specifications," in *ICST*, 2016.

[8] J. Lampel, S. Just, S. Apel, and A. Zeller, "When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla," in *ESEC/FSE*, 2021.

[9] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! Detecting flaky tests caused by concurrency with Shaker," in *ICSME*, 2020.

[10] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *MSR*, 2020.

[11] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *TSE*, 2022.

[12] E. Wendelin, 2022. [Online]. Available: https://blog.gradle.org/gradle-flaky-test-retry-plugin

[13] 2022. [Online]. Available: https://eng.fitbit.com/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests

[14] P. Gochenour and R. Andre, 2022. [Online]. Available: https://wiki.saucelabs.com/display/DOCS/How+to+Deal+with+Flaky+Java+Tests

[15] M. Fowler, 2022. [Online]. Available: https://martinfowler.com/articles/nondeterminism.html

[16] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *ICSE SEIP*, 2020.

[17] J. Malm, A. Causevic, B. Lisper, and S. Eldh, "Automated analysis of flakiness-mitigating delays," in *AST*, 2020.

[18] 2022. [Online]. Available: https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019

[19] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *SCAM*, 2018.

[20] 2022. [Online]. Available: http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html

[21] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *ICSE SEIP*, 2017.

[22] J. Micco, "The state of continuous integration testing at Google," in *ICST*, 2017.

[23] C. Ziftci and J. Reardon, "Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale," in *ICSE*, 2017.

[24] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing," in *ICSE*, 2017.

[25] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *ICSE*, 2015.

[26] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *ICSE*, 2015.

[27] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA*, 2019.

[28] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *ICSE*, 2020.

[29] 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

[30] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in *ESEC/FSE*, 2018.

[31] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *ESEC/FSE*, 2019.

[32] S. Habchi, G. Haben, M. Papadakis, M. Cordy, and Y. Le Traon, "A qualitative study on the sources, impacts, and mitigation strategies of flaky tests," in *ICST*, 2022.

[33] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Surveying the developer experience of flaky tests," in *ICSE SEIP*, 2022.

[34] A. Gyori, A. Shi, F. Hariri, and D. Marinov, "Reliable testing: Detecting state-polluting tests to prevent test dependency," in *ISSTA*, 2015.

[35] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *ICST*, 2018.

[36] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in *FSE*, 2014.

[37] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell, "Supplemental data for "Automatically reproducing timing-dependent flaky-test failures"," https://github.com/gmu-swe/flakerake, 2024.

[38] 2022. [Online]. Available: https://github.com/qos-ch/logback

[39] R. L. Burden and J. D. Faires, *Numerical Analysis*. PWS Publishers, 1985, 2.1 The Bisection Algorithm.

[40] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *ESEC/FSE*, 1999.

[41] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects," in *ISSRE*, 2020.

[42] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A practical mutation testing tool for Java (Demo)," in *ISSTA*, 2016.

[43] 2022. [Online]. Available: https://mvnrepository.com/artifact/edu.illinois.cs/idflakies/2.0.0

[44] S. Rahman, A. Baz, S. Misailovic, and A. Shi, "Quantizing large-language models for predicting flaky tests," in *ICST*, 2024.

[45] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. L. Traon, "What made this test flake? Pinpointing classes responsible for test flakiness," in *ICSME*, 2022.

[46] A. Alaboudi and T. D. LaToza, "An exploratory study of debugging episodes," *arXiv*, 2021.

[47] D. Silva, M. Gruber, S. Gokhale, E. Arteca, A. Turcotte, M. d'Amorim, W. Lam, S. Winter, and J. Bell, "The effects of computational resources on flaky tests," *arXiv*, 2023.

[48] 2022. [Online]. Available: https://github.com/apache/logging-log4j2

[49] 2022. [Online]. Available: https://learn.microsoft.com/en-us/azure/devops/pipelines/test/flaky-test-management

[50] 2022. [Online]. Available: https://gradle.com/blog/a-pragmatists-guide-to-flaky-test-management/

[51] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flaky test dataset to accompany "FlakeFlagger: Predicting flakiness without rerunning tests"," 2021.

[52] M. Gruber and G. Fraser, "A survey on how test flakiness affects developers and what support they need to address it," in *ICST*, 2022.

[53] T. Leesatapornwongsa, X. Ren, and S. Nath, "FlakeRepro: Automated and efficient reproduction of concurrency-related flaky tests," in *ESEC/FSE*, 2022.

[54] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "FlakiMe: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and program repair," in *ICSE*, 2022.

[55] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *MSR*, 2021.

[56] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *ESEC/FSE*, 2015.

[57] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in Android via event order exploration," in *ESEC/FSE*, 2021.

[58] V. Terragni, P. Salza, and F. Ferrucci, "A container-based infrastructure for fuzzy-driven root causing of flaky tests," in *ICSE NIER*, 2020.

[59] S. K. Damodaran-Kamal and J. M. Francioni, "Nondeterminancy: Testing and debugging in message passing parallel programs," in *PADD*, 1993.

[60] Y. Eytani, E. Farchi, and Y. Ben-Asher, "Heuristics for finding concurrent bugs," in *IPDPS*, 2003.

[61] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation," in *JGI*, 2001.

[62] S. D. Stoller, "Testing concurrent Java programs using randomized scheduling," in *RV*, 2002.

[63] Y. Eytani, "Concurrent Java test generation as a search problem," *ENTCS*, 2006.

[64] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur, "Producing scheduling that causes concurrent programs to fail," in *PADTAD*, 2006.

[65] Y. Eytani and T. Latvala, "Explaining intermittent concurrent bugs by minimizing scheduling noise," in *HVC*, 2006.
[66] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," in *OOPSLA*, 2013.
[67] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race detection for event-driven mobile applications," in *PLDI*, 2014.
[68] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in Android apps," in *ISSTA*, 2016.