DETECTING, CHARACTERIZING, AND TAMING FLAKY TESTS

BY

WING LAM

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2021

Urbana, Illinois

Doctoral Committee:

Professor Tao Xie, Co-Chair
Professor Darko Marinov, Co-Chair
Assistant Professor Tianyin Xu
Partner Researcher Suman Nath, Microsoft Research

ABSTRACT

As software evolves, developers typically perform regression testing to ensure that their code changes do not break existing functionalities. During regression testing, developers can waste time debugging their code changes because of spurious failures from flaky tests, which are tests that non-deterministically pass or fail on the same code. These spurious failures mislead developers about their code changes because the failures are often due to bugs that existed before the code changes.

One prominent category of flaky tests is *order-dependent (OD)* flaky tests. Each OD test has at least one order in which the test passes and another order in which the test fails, and for every test order, the test either passes or fails in all runs of that test order. Another prominent category is *async-wait (AW)* flaky tests. Each AW test makes at least one asynchronous call and passes if the asynchronous call finishes on time but fails if the call finishes too early or too late.

This dissertation tackles three main aspects of flaky tests. First, this dissertation presents novel techniques to *detect* flaky tests so that developers can preemptively prevent the problem of flaky tests from affecting their regression testing results. Second, this dissertation presents novel techniques to *characterize* flaky tests to help developers better understand their flaky tests and to help researchers invent new solutions to the flaky-test problem. Lastly, this dissertation presents novel techniques to *tame* the problem of flaky tests by accommodating the flakiness so that flaky tests do not mislead developers during regression testing.

For detecting flaky tests, this dissertation presents (1) iDFlakies, a framework for detecting and partially classifying flaky tests, and IDoFT, an increasingly used dataset of flaky tests found in popular open-source projects; (2) an analysis of the probability to detect OD tests from randomizing test orders and a novel algorithm to systematically explore all consecutive pairs of tests, guaranteeing to detect all OD tests that depend on one other test; and (3) a large-scale longitudinal study of flaky tests to determine when flaky tests become flaky and what changes cause them to become flaky—the results provide guidelines for when and how developers should spend their efforts to detect flaky tests.

For characterizing flaky tests, this dissertation presents (1) the first automated technique to help developers debug flaky-test failures; and (2) a study to understand the effect that test orders have on non-deterministic tests, which can pass and fail even for the same test order—the results suggest that many of these tests can fail with significantly different failure rates for different test orders.

Lastly, for taming flaky tests, this dissertation presents (1) the first automated techniques to reduce the number of spurious failures from OD tests, reducing such failures by 73%; and (2) the first automated techniques to speed up AW flaky tests while also keeping the number of spurious failures low, speeding up such tests by 38%.

Overall, the work in this dissertation has helped detect more than 2000 flaky tests in over 150 open-source projects and fix more than 500 flaky tests in over 80 open-source projects.

To my family and friends for their love and support.

ACKNOWLEDGMENTS

During the six years that I spent working on my Ph.D., I received help from many people, and this dissertation could not have been possible without everyone's help. In this acknowledgment, I do my best to express my gratitude to every one of you, but I will inevitably forget to mention some of you. Nevertheless, please understand that you have my deepest gratitude for all of the help and encouragement that you provided me over the years.

First of all, I would like to thank all of my mentors. In particular, I would like to thank my advisors, Professor Tao Xie and Professor Darko Marinov. When I began my Ph.D. in 2015, both of them expressed interest in working with me, and I am happy to see that the two of them ended up accepting me as their student. I became Tao's student early on due to my interest in pursuing Android testing related topics. I remember quite well the long hours that Tao would spend with me in the early years of my Ph.D. and how the lessons learned from those meetings would shape me to become the researcher that I am today. For example, one saying that Tao has is that I should not just keep cooking my research, and instead, I should let him smell or get a taste of the soup. Another saying that Tao has is that he is the "wide receiver sitting in the end zone waiting for us to throw the ball to him". Essentially, the sayings taught me the importance of frequently engaging and iterating with my advisors on my research. The sayings also highlight the many occasions that Tao invited me and others over to his place to eat food and watch major sporting events together. I found these occasions to be very comforting and am grateful to Tao for hosting them. Additionally, Tao always encouraged me to think more about the big picture of my work and to dream big with the work that I want to do. I am especially thankful to Tao for all of the lessons that he has taught me, and I hope to instill such ideals in my future students.

On the other hand, I started working with Darko during the Fall of 2018. Although I took three years to begin working with Darko, he was always a helpful mentor to me even before I officially became his student. In fact, Darko helped provide the connection and interview preparation for my internship at Microsoft Research Cambridge in the summer of 2016. This internship helped me foster a good connection with developers and researchers at the Tools for Software Engineering group and eventually led to two more internships. During one of those additional internships, my project was to work on automated techniques for debugging flaky tests (included in Chapter 5 of this dissertation). After my internship that summer, I became interested in researching flaky tests again after working on the topic during my undergraduate and focusing on other topics in the early years of my Ph.D. As Darko would

jokingly put it, this moment was when I "began to see the light" in research as the topic of flaky tests was one that Darko has had an interest in for a few years by then. After I began working on flaky tests and with Darko, I completed most of the work in this dissertation. I made progress because the topic is one that I am passionate about and because of how much Darko cares about his students' well-being and learning. One saying that Darko has is that he has heard that micro-managing students is bad, which is why he prefers to "nano-manage" his students instead. Although micro-managing (or "nano-managing") implies that he will control every part of his student's work and this management style generally carries a negative connotation, I found Darko's "nano-managing" to be instrumental to my growth as a researcher. In my initial projects with Darko, I lacked many of the skills to prototype ideas quickly. Being a much more experienced programmer, Darko could have quickly implemented the prototypes. However, instead of implementing them himself or just waiting for me to slowly learn how to implement them, Darko "nano-managed" me. Specifically, he spent much more time teaching me the necessary skills to implement the prototypes myself, thereby providing me crucial future research skills. I hope to demonstrate the same amount of care to my students in the future, possibly by "nano-managing" them as well ☺.

After I became Darko's student, I began interacting much more often with Owolabi Legunsen and August Shi, who braved the job market in 2019 and 2020, respectively. The two of them graciously spent their time helping me prepare for the job market this year and made themselves available for me to share the difficulties that I had to overcome in recent years and successes that I finally achieved. I like to think of the two of them as my unofficial advisors. I am deeply grateful for the time that the two of them spent with me and I will look to provide to others the same care and help that they provided me.

Beyond my advisors, other mentors of mine include my internship mentors. Namely, I would like to thank Marc Brockschmidt, Patrice Godefroid, Ben Livshits, Kivanç Muşlu, Suman Nath, Mukul Prasad, Ripon Saha, Hitesh Sajnani, Anirudh Santhiar, and Suresh Thummalapenta. I am particularly grateful to Suman for officially mentoring me for one internship, unofficially mentoring me for another, and for serving on my Ph.D. committee. I am also particularly grateful to Suresh for being my official mentor and advocating for my hiring for two internships. I also have some unofficial mentor friends. Specifically, Sean Hurley, Vivek Nair, and Michele Tufano all interned at Microsoft with me. The four of us spent much time together that summer and have kept in touch with one another throughout the years. As Vivek and Michele spent much more time in academia than I have, the two of them are unofficial mentors of mine for everything related to research and academia.

In some ways, all of my co-authors have mentored me through their contributions to the work that we shared. Namely, I would like to thank my co-authors: Angello Astorga, Blake

Bassett, Jonathan Bell, Long Chen, Yuetang Deng, Michael D. Ernst, Patrice Godefroid, Jonathan de Halleux, Darioush Jalali, Pratap Lakshman, Dengfeng Li, Ge Li, Beihai Liang, Hui Luo, Yingjun Lyu, Peyman Mahdian, Darko Marinov, Kivanç Muşlu, Suman Nath, David Notkin, Reed Oei, Mukul Prasad, Ripon Saha, Hitesh Sajnani, Anirudh Santhiar, August Shi, Siwakorn Srisakaokul, Victoria Stodden, Suresh Thummalapenta, Qianxiang Wang, Wenyu Wang, Anjiang Wei, Stefan Winter, Zhengkai Wu, Jochen Wuttke, Fan Xia, Tao Xie, Peng Yan, Wei Yang, Pu Yi, Hiroaki Yoshida, Hao Yu, Xia Zeng, Sai Zhang, Haibing Zheng, and Wujie Zheng. There is no way that I would have accomplished the work that I did in my Ph.D. without the mentoring and support from everyone. I look forward to more collaboration opportunities in the future with everyone.

Looking beyond my mentors, I would like to thank my friends. I started my Ph.D. at the same time as Angello Astorga, Chiao Hsieh, and Siwakorn Srisakaokul. I collaborated with Angello and Siwakorn on some papers. Although I never collaborated with Chiao on a paper, all four of us spent much time together in the early years of our Ph.D. studies, and we learned a lot from one another. The three of them often served as my "sounding board" for many research ideas and for my complaints about the stressful moments of grad school.

I would also like to thank many friends of mine in various groups. First, I would like to thank the members of the group led by Professor Tao Xie. Specifically, I thank Angello Astorga, Liia Butler, Dengfeng Li, Linyi Li, Xueqing Liu, Siwakorn Srisakaokul, Wei Yang, Zirui Zhao, and Zexuan Zhong for their companionship during the many group meetings and outings that we have had together. Second, I would like to thank the members of the group led by Professor Darko Marinov. Specifically, I thank Alex Gyori, Farah Hariri, Owolabi Legunsen, and August Shi for their guidance during my Ph.D. and their discussions about research ideas. Third, I would like to thank the members of the group led by Professor Sasa Misailovic. Specifically, I thank Saikat Dutta, Vimuth Fernando, Zixin Huang, Keyur Joshi, and Jacob Laurel for their discussions about research and for their participation in the many extracurricular activities we shared (e.g., volleyball games, BBQs). Lastly, I would like to thank Wajih Ul Hassan, Umang Mathur, and Amarin Phaosawasdi, who provided valuable comments regarding my research and job application material.

During my Ph.D., I was also fortunate enough to have the privilege of mentoring many extraordinary undergraduate and Master's students. I would like to thank all of them for the privilege of mentoring them and for their hard work, which in one way or another contributed to the research that I have done. I would particularly like to thank Dengfeng Li, Reed Oei, Anjiang Wei, Pu Yi, and Hao Yu for their hard work and the papers we published together. I hope that all of you find much success in your next endeavors.

For the preparation of my job search this year, many faculty members at the University

of Illinois at Urbana-Champaign (UIUC) provided me valuable advice. Specifically, I would like to thank Professors Sarita Adve, Vikram Adve, Nancy Amato, Saugata Ghose, Colleen Lewis, and Lingming Zhang for their words of encouragement, job-search advice, and mock interviews. I would also like to thank Professors Carl Gunter, Indranil Gupta, Jiawei Han, Julia Hockenmaier, Reyhaneh Jabbarvand, Robin Kravets, Sasa Misailovic, Madhusudan Parthasarathy, and Tianyin Xu for their advice on my job talk presentation and my research in general. A special thank you goes to Tianyin Xu for also serving on my Ph.D. committee.

I would also like to thank the many CS department staff at UIUC who helped facilitate my funding, prepare me for graduation, and the many other support activities that I needed during my Ph.D. Specifically, I would like to thank Kim Baker, Kimberly Bogle, Maggie Chappell, Jancie Harris, Samantha Hendon, Viveka Kudaligama, and Kara MacGregor.

My research would not have been possible without funding from various agencies, such as the National Science Foundation, Facebook, Futurewei, Google, and Microsoft. I was also supported by the Google – CMD-IT LEAP Alliance Fellowship, Ray Ozzie Computer Science Fellowship, State Farm Companies Foundation Doctoral Scholarship, and Yunni & Maxine Pao Memorial Fellowship.

The work in this dissertation was published at seven different conferences: ICST 2019 (Chapter 2), TACAS 2021 (Chapter 3), OOPSLA 2020 (Chapter 4), ISSTA 2019 (Chapter 5), ISSRE 2020 (Chapter 6), ISSTA 2020 (Chapter 7), and ICSE 2020 (Chapter 8). I would like to thank all of the anonymous reviewers who reviewed the submissions and provided comments to improve the work. I would also like to thank the audiences who attended and commented on the presentations of the work in this dissertation.

Lastly and most importantly, I would like to thank my family. In particular, I would like to thank my parents, Veda Lam and Vikki Miu, for their care and support over the years. My mom devoted much of her life to caring for my sister and me, but she unfortunately passed away before I started college. I attribute almost all of my accomplishments to my memory of her, and this dissertation is no different. I would also like to thank my aunt, Teresa Miu, for taking care of me and often putting her own interest behind the interest of others, such as myself. Growing up, I spent much of my time with my sister, Ming Lam, and my cousins, Arthur Chang, Philip Chang, Bo Kwok, and Camy Kwok. All of them provided much help and companionship to me over the years. Eventually, Arthur married Michelle Lai and had Katherine Chang and Kelly Chang, while Philip married Fei Yuan and had Heidi Chang and Hermes Chang. It provides me immense joy to babysit the little Changs and watch them grow up. Finally, I want to thank my girlfriend, Karen Lam. I am very excited for the next chapters of our lives together and am extremely grateful to her for everything that she does for me and for all of the happiness that she brings to my life.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

With software permeating all kinds of systems, the reliability of software has monumental societal impact. The most common and practical way for developers to ensure that their software is reliable is through the use of *regression testing* as they develop software. Namely, developers run tests to check that recent changes do not break existing functionalities. Researchers have proposed a variety of regression testing techniques to improve regression testing. These regression testing techniques produce an order (a permutation of a subset of tests in the test suite) in which to run tests. Examples of such traditional regression testing techniques include test prioritization (run all tests in a different order with the goal of finding failures sooner) [45, 89, 101, 117, 171, 173, 192, 227], test selection (run only a subset of tests whose outcome can change due to the code changes) [20, 77, 83, 146, 157, 227, 234], and test parallelization (run tests in parallel across multiple machines) [93, 102, 140, 193].

Both proprietary software development and the open-source community have embraced the use of regression testing as part of their *continuous integration (CI)* model of software development and releases [81, 180]. In this model, every check-in is validated through an automated pipeline, perhaps running as a service in the cloud, that fetches source code from a version-controlled repository, builds source code, and runs tests against the built code. These tests must all pass in order for the developer to integrate changes with the main development branch. Thus, tests play a central role in ensuring that the changes do not introduce regressions.

During software development, tests should run quickly and reliably without imposing undue load on underlying resources such as build machines. That is, in an ideal world, test failures would reliably signal issues with the developer's changes and every test failure would warrant investigation. Unfortunately, the reality of CI pipelines today is that some tests may pass and fail with the *same* version of source code and the *same* configuration. These tests are commonly referred to as *flaky tests* [126]. Tests can be flaky due to the use of concurrency, timeouts, asynchronous calls, network/IO, dependency on the order that tests are run, etc. [42, 126]. More details about the categories of flaky tests are described in Section 1.2.

Many software organizations report that flaky tests are one of their biggest problems in software development. For example, Facebook released a position paper on the importance of flaky tests [76] and recently ran a call for research projects focused on flaky tests [52]. Several papers and blog posts from Google [54, 69, 134, 136, 240] and Microsoft [78, 106, 107] have reported several challenges with flaky tests, even estimating the monetary cost of flaky

tests on developer productivity [80]. Other organizations, including Apple [103], Fitbit [2], Gradle [220], Huawei [90], Mozilla [42, 165, 201], Netflix [149], Salesforce [56], Saucelabs [66], and ThoughtWorks [197], also publicly report their problems with flaky tests. In fact, at Facebook, Harman and O'Hearn [76] have even proposed to adopt the position that all tests are flaky (ATAF) and that researchers should rethink testing techniques knowing that they will be used in an ATAF world.

The work in this dissertation tackles three main aspects of flaky tests. First, this dissertation presents novel techniques to *detect* flaky tests [108, 112, 217] so that developers can preemptively prevent the problem of flaky tests from affecting their regression testing results. Second, this dissertation presents novel techniques to *characterize* flaky tests [106, 111] to help developers debug flaky-test failures and to help researchers invent new solutions to the flaky-test problem. Lastly, this dissertation presents novel techniques to *tame* the problem of flaky tests [107, 109] by accommodating the flakiness, so that flaky tests do not mislead developers during regression testing. Besides flaky tests, I have also worked on test-input generation [215, 231, 239], automatic program repair [179], code-clone detection [228], parameterized unit tests [110], and record and replay [113], but the focus for this dissertation is solely on the work related to flaky tests.

1.1 THESIS STATEMENT

The thesis statement of this dissertation is the following.

> Proactively detecting, characterizing, and taming flaky tests can help mitigate the problems of flaky tests.

Every time developers make a code change, they perform regression testing where they run tests after the change. During the process of regression testing, a test failure is supposed to indicate that the change has introduced a regression. However, in the presence of flaky tests, the developers may realize after manual inspection that the test failure is because the test is flaky and that the failure is actually unrelated to the code change. The developers have essentially "detected" a flaky test, but it is happening at an inopportune time. Detecting, characterizing, and taming flaky tests at this time is what we would refer to as *reactively* detecting, characterizing, and taming flaky tests.

Much existing work [23, 86, 114, 134, 144, 176, 178] has found that providing developers with feedback regarding problematic code is most helpful if provided as soon as possible in the development process. For example, for the static bug-detection tool, Infer [86], at Facebook, Harman and O'Hearn [76] found that bugs reported to developers at *"post land"*

(after code has been merged into the main development branch) had a close to 0% fix rate; yet bugs reported at *"diff time"* (when developers submit code changes) had a fix rate of over 70%. In the case of flaky tests, it is likely far more useful to notify developers soon after they made a change that introduced test flakiness, rather than to notify developers weeks or months after they made the change. In fact, the standard testing practice at organizations, such as Mozilla [201] and Netflix [149], already aims to find whether newly added tests are flaky as soon as possible.

The work in this dissertation proposes techniques and develops tools to *proactively* detect, characterize, and tame flaky tests before they manifest as flaky-test failures so that developers would experience reactive detection, characterization, and taming much less often. To help with this goal, I have co-authored 9 publications on flaky tests, and this dissertation focuses on the 7 publications where I was the lead author. Specifically, three of the publications [108, 112, 217] help developers detect flaky tests, two of them [106, 111] help developers characterize flaky tests, and finally, two of the publications [107, 109] help developers tame the problems of flaky tests.

1.2 CATEGORIES AND EXAMPLES OF FLAKY TESTS

Prior studies on flaky tests identified many categories for why tests can be flaky [42, 126]. One state-of-the-art flaky-test detection tool is iDFlakies [108], which runs tests in various orders to detect flaky tests and partitions flaky tests into two categories. One category is *order-dependent (OD)* tests that can *deterministically* pass or fail based on the order in which the tests are run. Such tests are deterministic in that their failure rates are either 0% or 100% for each order, and they have at least two orders whose failure rates differ. A *failure rate* is defined as the ratio of the number of failed runs over the number of total runs for a particular test order. We represent a test order as a sequence of tests $\langle t_1, t_2, \ldots, t_l \rangle$. In Java, each test order is executed by a Java Virtual Machine (JVM) that starts from the initial state (e.g., all shared pointer variables initialized to `null`) and then runs each test, which potentially modifies the shared state. Each test is run *at most once* in one JVM run. For completeness of categorization, we also define tests that are *not flaky*—they either always pass (all orders have 0% failure rate) or always fail (all orders have 100% failure rate).

The other category is *non-deterministic (NOD)* tests that are flaky but not OD. Such tests have at least one order where the test fails non-deterministically (failure rate is neither 0% nor 100%). We further break down these NOD tests to non-deterministic, order-independent (NDOI) tests and non-deterministic, order-dependent (NDOD) tests. Specifically, *NDOI* tests are NOD tests where all orders' failure rates do *not* significantly differ, e.g., all orders

Table 1.1: Categorization of tests. $\mathbb{F}$ are failure rates per order for a test. DIF is true iff any rate statistically significantly differs from others.

Non-deterministic (**NOD**)	Deterministic
$(\exists f \in \mathbb{F}. 0\% < f < 100\%) \wedge$ $DIF(\mathbb{F})$ [**NDOD**]	$(\forall f \in \mathbb{F}. f = 0\% \vee f = 100\%) \wedge$ $(\exists f', f'' \in \mathbb{F}. f' \neq f'')$ [**OD**]
$(\exists f \in \mathbb{F}. 0\% < f < 100\%) \wedge$ $\neg DIF(\mathbb{F})$ [**NDOI**]	$(\forall f \in \mathbb{F}. f = 0\%) \vee$ [**not flaky, pass**] $(\forall f \in \mathbb{F}. f = 100\%)$ [**not flaky, fail**]

of a test have a 1% failure rate. A statistical test can be used to determine if any one order's rate significantly differs from the others. Conversely, *NDOD* tests are NOD tests where at least one order's failure rate significantly differs from other orders' failure rates, e.g., a test that has a 99% failure rate in one order but 0% in another order.

Some prior studies of flaky tests [42, 61, 107, 126] categorized NOD tests into fine-grained categories (e.g., concurrency, network, I/O), but the authors *manually* examined the source code of the tests or conducted surveys of developers after the tests were fixed and the causes of flakiness were removed. The NOD-related categories that we present may be more coarse-grained than the ones studied in prior work. However, unlike the categories studied in prior work, the categories that we present do not require manual examinations or surveys and can be *automatically* obtained. Table 1.1 shows the precise definitions for our OD category and our coarse-grained NOD-related categories. The rest of this section shows examples of OD, NDOI, and NDOD tests.

1.2.1 Example of Order-Dependent (OD) Test

OD tests are flaky tests that can pass or fail depending on only the order in which the tests are run [237], i.e., OD tests can be made to deterministically pass or fail by fixing the order of tests [59]. Detecting OD tests is important in general, because test frameworks can change the test order, even when running all the tests, thereby causing the failures of OD tests to affect developers. Moreover, techniques that aim to shorten the time of regression testing—including test-suite reduction [170, 183, 185, 223], test selection [20, 64, 116, 146, 157, 169, 187, 234], and test parallelization [51]—select only a subset of tests to run and could additionally expose failures of OD tests.

Our iFixFlakies work [186] has studied the *causes* of failures for OD tests. We find that the vast majority of OD tests are related to *pairs* of tests, i.e., each OD test would pass or fail due to the sharing of some global state with just one other test. Our iFixFlakies work has also defined multiple types of tests related to OD tests. Each OD test belongs to one of two categories:

```
// OD Victim (in ShutdownListenerManagerTest class)
@Test
public void assertIsShutdownAlready() {
  shutdownListenerManager.new InstanceShutdownStatusJobListener().dataChanged(
    "/test_job/instances/127.0.0.1@-@0", Type.NODE_REMOVED, "");
  verify(schedulerFacade, times(0)).shutdownInstance();
}
// Polluter (also in ShutdownListenerManagerTest class)
@Test
public void assertRemoveLocalInstancePath() {
  JobRegistry.getInstance().registerJob("test_job", jobScheduleController, regCenter);
  shutdownListenerManager.new InstanceShutdownStatusJobListener().dataChanged(
    "/test_job/instances/127.0.0.1@-@0", Type.NODE_REMOVED, "");
  verify(schedulerFacade).shutdownInstance();
}
// Cleaner (in FailoverServiceTest class)
@Test
public void assertGetFailoverItems() {
  JobRegistry.getInstance().registerJob("test_job", jobScheduleController, regCenter);
  ... // 12 more lines
  JobRegistry.getInstance().shutdown("test_job");
}
```

Figure 1.1: Example OD victim, polluter, and cleaner tests from the Elastic-Job [43] project.

1. *brittle*, which is a test that fails when run by itself but passes in a test order where the test is preceded by a *state-setter*; and

2. *victim*, which is a test that passes when run by itself but fails in a test order where the test is preceded by a (state-)*polluter* unless a (state-)*cleaner* runs in between the polluter and the victim. In other words, a test v is a victim if it passes in the order $\langle v \rangle$ but fails in another order; the other order usually contains a single polluter test p (besides many other tests) such that v fails in the order $\langle p, v \rangle$. If the test suite contains a cleaner test c, then v passes in the order $\langle p, c, v \rangle$. Note that test orders *may contain more tests* besides polluters and cleaners for a victim v, but these other tests do not modify the relevant state and do not affect whether v passes or not in any order. Precise definitions for these tests are in our prior work [186].

Most of the work in this dissertation focuses on victim tests because most OD tests are victims rather than brittles (e.g., 91% of the 110 truly OD tests in the iDFlakies dataset are victims [85]), and brittles can often be treated as a simple special case of victims.

Figure 1.1 shows an example OD test that our flaky-test detection tool (iDFlakies) found in a popular open-source project. The test, `assertIsShutdownAlready`, is an OD victim because its passing depends on some tests *not* to run before it. One such test is `assertRemoveLocalInstancePath`, which is also shown in Figure 1.1. This test is a polluter because it starts

```
1 [TestMethod]
2 public void DelayedTaskStaticBasicTest() {
3   int delay = 1000; int i = 0;
4   Scheduler.ScheduleTask(DateTime.UtcNow.AddMilliseconds(delay),
5     new LoggedTask("TestDelayedTaskFrameworkTask", () => { i = 1; },
6       new Dictionary<string, string> { { "test", "value" } }));
7   Thread.Sleep(500);
8   Assert.IsTrue(i == 0);
9   Thread.Sleep(delay);
10  Assert.IsTrue(i == 1);
11 }
```

Figure 1.2: NDOI, AW flaky test from a Microsoft proprietary project.

a job manager that is shared with other tests (Line 11) but does not shut it down at the end of the run. Running the polluter before the victim is problematic in this case because Line 6 of `assertIsShutdownAlready` checks whether the shared job manager is shut down, and because the job manager is started and not shut down by the polluter, the victim will fail if the polluter is run first. The victim passes by itself or in orders where the tests that start the instance are run after the OD test. Figure 1.1 also shows an example of a cleaner (`assertGetFailoverItems`) for the victim. `assertGetFailoverItems` is a cleaner because it shuts down the instance started by polluters on Line 21. Therefore, as long as `assertGetFailoverItems` runs between polluters and `assertIsShutdownAlready`, then `assertIsShutdownAlready` will pass. In popular open-source projects, we find that up to 50.5% of flaky tests are OD tests. More details about our findings regarding OD tests are in Section 2.

1.2.2 Example of Non-Deterministic Order-Dependent (NDOI), Async-Wait (AW) Test

NDOI tests are flaky tests that can pass or fail depending on any reason other than solely on the order in which the tests are run, i.e., how often these tests would fail is not affected in any way by the order in which tests are run. Tests of this type are flaky due to asynchronous calls, concurrency, timeouts, network/IO, etc. [126]. For example, these tests can be flaky due to an *async-wait (AW)* issue, which is when a test execution makes an asynchronous call and does not properly wait for the result of the call to become available before using it. Note that the asynchronous call may be in the test code, code under test, or in a library that the project depends on. Whether such flaky tests pass or fail depends on whether the asynchronous call was able to finish or not.

Figure 1.2 shows an example of an AW test. `DelayedTaskStaticBasicTest` schedules a task to run at a pre-defined time (the current time + 1000 milliseconds) on Line 4. The test

can be flaky due to two main reasons.

1. When the asynchronous task on Line 4 finishes executing before Line 8, then the test will fail. In passing executions of this test, the value of `i` has not changed to `1`, but in the failing executions, delays before the assertion on Line 8 can actually be greater than the time it takes to execute the asynchronous task on Line 4. In such cases, the value of `i` when Line 8 executes is already `1` and the assertion will fail.

2. When the assertion on Line 10 finishes executing before the asynchronous task on Line 4 finishes executing, then the test will fail. In passing executions of this test, the value of `i` is changed to `1` before Line 10 executes, but in the failing executions, the asynchronous task on Line 4 runs so slow that the delays on Lines 7 and 9 are not enough to prevent Line 10 from executing before the asynchronous task finishes.

We find that the most common category of flaky tests in Microsoft proprietary projects is the async-wait category [107]. More details regarding our findings about flaky tests in Microsoft proprietary projects are in Section 5 and Section 8.

1.2.3 Example of Non-Deterministic Order-Dependent (NDOD) Test

One example NDOD test is shown in Figure 1.3. The `shouldRetryWithDynamicDelayDate` test is from a project [166] that implements client-side response routing. This test is flaky because some runs take longer than the timeout limit of 1500ms. We find that the test is NDOD because Line 4 launches a server, and in runs where the test passes, other tests running before this test trigger the Java just-in-time (JIT) compiler, thereby reducing the latency to start the server. On the other hand, in runs where the test fails, the JIT compiler is not triggered, thereby increasing the latency to start the server and resulting in the test exceeding 1500ms. We confirmed our understanding by logging the time that it takes to run the test and then running the test multiple times. With our changes, we see that after $\sim$5 runs, the runtime of this test would decrease from $\sim$1500ms to $\sim$1000ms.

We refer to the number of times that a flaky test fails consecutively as the test's *burst length*. In 4000 runs of the test suite containing this NDOD test, we find that this test's maximal burst length is 7 in one order, with the average maximal burst length being 3.1 across the 21 test orders that we ran. We also find that this test has a failure rate of 39.1% when run by itself in isolation, while its failure rate is 10.9% when it is run in its test suite, with the failure rate ranging from 0% to 20.1% depending on the test order. This test's failure rate varies a lot because some other tests running before it can exercise code that

```
@Test(timeout = 1500)
public void shouldRetryWithDynamicDelayDate() {
  ... // test setup
  atLeast(Duration.ofSeconds(1),() -> unit.get("/baz").dispatch(...).join());
}
```

Figure 1.3: NDOD test from the riptide [166] project.

triggers the JIT compiler, in which case the test often passes because it then takes closer to 1000ms. In popular open-source projects, we find that the majority of NOD tests are NDOD instead of NDOI. More details about our findings regarding NOD tests are in Section 6.

1.3 DETECTION

As Section 1.1 points out, providing developers with feedback regarding problematic code is most helpful if provided as soon as possible in the development process. Organizations such as Mozilla [201] and Netflix [149] have already made changes to incorporate this insight into their testing practice.

To help developers detect flaky tests as soon as possible, we develop a tool, called *iDFlakies*, that can (1) detect flaky tests by reordering and rerunning tests in a project and (2) *partially* classify flaky tests as likely OD or NOD tests by checking various test orders. We implement our tool as a Maven plugin for Java projects that use JUnit tests. The tool offers five configurations to run the tests and detect flaky tests. The base configuration simply reruns the original order of the tests many times to check whether the result of any test changes; any test that passes and fails for the same code version in the same test order is by definition a flaky, NOD test. The other configurations reorder the test methods and classes to focus on detecting OD tests, but these configurations can also detect NOD tests along the way. Following Zhang et al. [237], iDFlakies reorders tests using random orderings or reversing the original order of the tests. However, iDFlakies differs from Zhang et al.'s tool in that iDFlakies's random orderings do not interleave test methods from different test classes. An ordering that interleaves tests from test classes would not be produced by popular testing frameworks, such as JUnit [96], TestNG [204], Cucumber [30], and Spock [190].

We also develop a methodology to analytically obtain the flake rates of OD tests and propose a simple change to the random sampling of test orders to increase the probability of detecting OD tests [217]. A *flake rate* of a test is the probability that the test fails in a randomly sampled test order from all possible orders and is defined as the ratio of the number of test orders in which a test fails divided by the total number of possible orders. Flake rates can help researchers analytically compare various algorithms (e.g., comparing

reversing a passing order to sampling a random order as shown in Section 3.3.4) and help practitioners prioritize the fixing of flaky tests. Specifically, we study the following problem: determine the flake rate for a given victim test with its set of polluters and a set of cleaners for each polluter. Our analysis finds that some OD tests have a rather low flake rate, as low as 1.2%.

Because random sampling of test orders may miss test orders in which OD tests fail, we also propose a systematic approach to cover all consecutive test pairs to detect OD tests. We present an algorithm that systematically explores all consecutive test pairs, guaranteeing the detection of all OD tests that depend on one other test, while running substantially fewer tests than a naive exploration that runs every pair by itself. Our algorithm builds on the concept of Tuscan squares [68], studied in the field of combinatorics. Given a test suite, the algorithm generates a set of test orders, each consisting of at least two distinct tests and at most all of the tests from the test suite, that cover all of the consecutive test pairs, while trying to minimize the cost of running those test orders. Our analysis shows that the algorithm runs substantially fewer tests than naive exploration.

One important obstacle to performing research on flaky tests is obtaining a dataset of current flaky tests in real-world projects, similar to datasets such as SIR [36] and Defects4J [98] that helped research studies on regression testing, automated debugging, and program repair. Some prior work studies only flaky tests from older code versions [18] or focuses on only flaky tests that had been fixed [126]. Other work does not classify flaky tests into OD or NOD tests, or performs studies on only a relatively small number of projects [237].

To offer a dataset of current flaky tests in real-world projects, we apply our flaky-test detection techniques on a large number of open-source projects, and create a dataset of such flaky tests. Through many pieces of this dissertation's work and open-source contributions from others, our dataset has grown to over 2000 flaky tests detected and over 500 flaky tests fixed. We make our tools and dataset publicly available [8, 85, 105], allowing other researchers to use them in their research experiments on flaky tests.

Overall, this dissertation makes the following main contributions for detecting flaky tests.

- **Tool.** We develop and make publicly available a tool called iDFlakies to detect flaky tests and classify them into two categories; our tool can be easily integrated into Maven projects that use JUnit.

- **Probability analysis.** We develop a methodology to analytically obtain the flake rates of OD tests and propose a simple change to the random sampling of test orders to increase the probability of detecting OD tests. Our analysis finds that some OD tests have a rather low flake rate, as low as 1.2%.

- **Systematic test-pair exploration.** We present an algorithm that systematically explores all consecutive test pairs, guaranteeing the detection of all OD tests that depend on one other test, while running substantially fewer tests than a naive exploration that runs every pair by itself.

- **Dataset.** We describe a collection of artifacts, including Docker images and test-run logs, that we use to create a dataset of flaky tests [85].

- **Study.** We present a study of flaky tests in open-source Java projects. Our findings include how prevalent OD and NOD types of flaky tests are, how to automatically detect these tests, when flaky tests are introduced, what changes cause tests to be flaky, and how should developers use flaky-test detection tools.

1.4 CHARACTERIZATION

The presence of flaky tests imposes a significant burden on developers using CI pipelines. In a survey conducted on 58 Microsoft developers, we find that they considered flaky tests to be the top 2 most important reasons, out of 10 reasons, for slowing down software deployments. A further detailed survey conducted on 18 of the developers showed that they value debugging and fixing existing flaky tests as one of the most important course of action for Microsoft to take regarding flaky tests [106]. These debugging and fixing efforts are often complicated by the fact that the test failures may only occur intermittently, and are sometimes reproducible only on the CI pipeline but not on local machines. When we re-run flaky tests locally 100 times, we find that 86% of them pass in all runs, i.e., they manifest as flaky only in the CI pipeline. This result is not surprising since reproducing flaky-test failures entails triggering a particular execution among many possible non-deterministic executions for a given flaky test. Non-determinism can arise from the non-availability of external I/O resources, such as network or disk, or from the order of thread and event scheduling. Prior work [126, 237] uncovering these factors has examined in detail the extent to which these factors contribute towards flakiness, and developers often find it too difficult and expensive to identify the root cause of flakiness.

To help developers with this problem, we have proposed a tool called RootFinder [106] that analyzes the logs of passing and failing executions of the same test to suggest method calls that could be responsible for the flakiness. We also describe our framework, which encompasses RootFinder, that can be used to instrument flaky tests and automatically obtain passing and failing execution logs for RootFinder to help developers debug flaky-test

failures. We implemented RootFinder with Microsoft developers and evaluated the tool in an industrial setting using production data obtained from Microsoft proprietary projects.

Beyond our work on flaky tests in proprietary projects, we also study flaky tests in open-source projects. We believe that two key challenges have limited the amount of in-depth work on NOD tests. The first challenge is the machine cost for rerunning tests. Many NOD tests may fail rather infrequently (e.g., once in 4000 test runs, as we observe from our experiments) or only under specific circumstances, so it takes substantial time and reruns to observe even one failure, let alone a few failures to study when and how they occur. The second challenge is the human cost for debugging NOD tests. In contrast to OD tests that fail deterministically and could be somewhat easier to reproduce and debug, NOD tests fail non-deterministically, potentially infrequently, and can take a lot of time to debug, especially for researchers unfamiliar with some open-source code that has flaky tests. For example, in our study [111], inspecting each new flaky test took one of the authors about a day on average to precisely understand the root cause of non-determinism. To help developers gain a more in-depth understanding of NOD tests, we present a study that is organized around four main research questions. Our study uses popular, open-source projects and is aimed at improving our understanding of how to rerun, detect, debug, and prioritize flaky tests.

Overall, this dissertation makes the following main contributions for characterizing flaky tests.

- **Tool.** We develop and make publicly available a prototype tool [168], called RootFinder, that analyzes the logs of passing and failing executions of the same test to suggest method calls that could be responsible for the flakiness.
- **Framework.** An end-to-end framework, developed within Microsoft that uses RootFinder to root-cause flaky tests.
- **Proprietary project study.** A qualitative study of flaky tests in Microsoft proprietary projects. Our qualitative study provides insights on root causing flaky tests with in-depth examples that demonstrate the root causes of flaky tests.
- **Open-source project study.** Our empirical evaluation of flaky tests in open-source Java projects provides actionable guidelines and practical suggestions for developers and researchers to rerun, detect, debug, and prioritize flaky tests.

1.5 TAMING

To demonstrate how developers should tame flaky tests, let us consider the example from

Section 1.2.1 where a server not being shut down would cause `assertIsShutdownAlready` to encounter an order-dependent test failure when it is run after another test. The main ways to help developers tame this OD test are to *reduce* the chance of OD-test failures or completely *remove* the chance of OD-test failures. For reducing the chance of OD-test failures, developers can accommodate the OD-test failures by running the tests in only some orders. For removing the chance of OD-test failures, developers can fix these tests by cleaning the relevant parts of the shared state between the tests. Developers can also fix the code by having the tests not share any state, such as changing the server not to be shared between tests. However, doing so may not always be possible, or it can incur substantial costs.

For our example with `assertIsShutdownAlready`, accommodating this OD test may be done by making it so that `assertIsShutdownAlready` must always run before tests that would start the server but not shut it down. By doing so, the OD tests are allowed to run in only some orders, but the benefit is that these tests may run faster because the tests are allowed to reuse states. Namely, by accommodating these tests, the tests would only need to initialize and shut down the server once instead of multiple times. Running tests in only some orders is particularly popular at some organizations such as Microsoft [107]. In this dissertation, we present one piece of the work on accommodating OD tests (Section 7) and another piece of the work on accommodating async-wait tests (Section 8).

On the other hand, for fixing, we may have every test create and test on their own instance of the server. By changing the code, `assertIsShutdownAlready` would now be able to run in any order, which enables the use of traditional regression testing techniques, and still get reliable results. One of our past projects [186] proposes a technique to automatically fix OD tests and developers from popular open-source projects have accepted more than 64 of these fixes that we submitted from this work.

In general, we find that there are developers who prefer fixing OD tests but there are also those who prefer just accommodating such tests. Regardless of what the developer prefers, one constant between the two ways of taming is that OD tests should not fail due to the test order changes but fail only when there is really a fault in the developer's changes.

Overall, this dissertation makes the following main contributions for taming flaky tests.

- **Study effect of flaky tests on regression testing techniques.** A study of how OD tests affect traditional regression testing techniques such as test prioritization, test selection, and test parallelization. When we apply regression testing techniques to test suites containing OD tests, 82% of the human-written and 100% of the automatically generated test suites contain one or more OD tests that fail.

- **Approach to accommodate OD tests.** A general approach to enhance tradi-

tional regression testing techniques to be dependent-test-aware. We apply our general approach to 12 traditional, regression testing algorithms, and make them and our approach publicly available [5]. An evaluation of the 12 enhanced algorithms shows that the orders produced by the enhanced algorithms can have 80% fewer OD-test failures, while being only 1% slower than the orders produced by the unenhanced algorithms.

- **Study lifecycle and categories of flaky tests.** We study the lifecycle and categories of flaky tests in Microsoft proprietary projects. Our study results suggest the need for an approach to accommodate AW tests. As part of our study, we are the first to investigate the reoccurrence, runtimes, and time-before-fix of flaky tests. The data that we use for our study is available online [34].

- **Approach to accommodate AW tests.** We propose an automated approach, called FaTB, to balance test flakiness and runtime. Our empirical experiments show that FaTB can help AW tests run up to 78% faster, and the AW tests will still have the same flaky-test-failure rates as before.

1.6 DISSERTATION ORGANIZATION

The remainder of this dissertation is organized as follows.

Chapter 2: [Detecting] iDFlakies: A Framework to Detect and Partially Classify Flaky Tests

This chapter presents iDFlakies, a tool to automatically detect and partially classify flaky tests and a study of flaky tests in open-source Java projects.

Chapter 3: [Detecting] Probabilistic and Systematic Coverage of Consecutive Test-Method Pairs to Detect Order-Dependent Flaky Tests

This chapter presents a probability analysis of detecting OD tests and a systematic approach to guarantee the detection of OD tests that depend on one other test.

Chapter 4: [Detecting] A Large-Scale Longitudinal Study of Flaky Tests

This chapter presents our in-depth study on when flaky tests are introduced, what changes cause tests to be flaky, and how developers should utilize their efforts to detect flaky tests.

Chapter 5: [Characterizating] Root Causing Flaky Tests in a Large-Scale Industrial Setting

This chapter presents RootFinder, a technique developed with Microsoft collaborators to automatically root cause flaky-test failures.

Chapter 6: [Characterizating] Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects

This chapter presents our in-depth study of non-deterministic tests and provides actionable guidelines to deal with flaky tests.

Chapter 7: [Taming] Accommodating Order-Dependent Flaky Tests

This chapter presents our work on enhancing regression testing techniques to accommodate OD tests so that these tests encounter fewer spurious failures when regression testing techniques are used.

Chapter 8: [Taming] Accommodating Async-Wait Flaky Tests

This chapter presents our work on FaTB, a technique developed with Microsoft collaborators to speed up async-wait flaky tests while also reducing their spurious failures.

Chapter 9: Related Work

This chapter presents an overview of related work on the topics of detecting, characterizing, and taming flaky tests.

Chapter 10: Conclusions and Future Work

This chapter concludes the dissertation and discusses future work that can be done following the work in this dissertation.

CHAPTER 2: [DETECTING] IDFLAKIES: A FRAMEWORK TO DETECT AND PARTIALLY CLASSIFY FLAKY TESTS

This chapter presents iDFlakies, a tool to automatically detect and partially classify flaky tests. Using the tool, we also create and publicize a dataset of flaky tests in open-source Java projects that we hope can help spur more research in the topic of flaky tests. In fact, this dataset of flaky tests is already used in chapters 3, 4, 6, and 7 and another work of ours [186]. Using the dataset, we additionally present a study of flaky tests. Our study findings include the prevalence of OD and NOD types of flaky tests and how to automatically detect these tests. Section 2.1 presents the iDFlakies tool. Section 2.2 describes the end-to-end framework that researchers and practitioners can use to easily extend and apply iDFlakies to detect flaky tests and classify them into two types. Section 2.3 presents our study setup, and Section 2.4 presents the results of our study. Section 2.5 then presents the threats to validity of our work, and Section 2.6 concludes this chapter.

2.1 IDFLAKIES

We develop a tool called iDFlakies that detects flaky tests and classifies each test as either OD or NOD (as defined in Section 1.2); iDFlakies does not further classify the NOD tests into more precise causes of flaky tests [42, 126]. As inputs, iDFlakies conceptually takes a test suite, a configuration for ordering the tests, and the number of times to run the test suite based on the configuration. The available configurations are described in Section 2.1.1. As output, iDFlakies produces the detected flaky tests, the type of each flaky test (OD or NOD), and the exact order in which each flaky test fails. To detect flaky tests, iDFlakies repeatedly runs the test suite based on the configuration specified by the user. We refer to a single run of the test suite as a *round*. The default configuration orders the tests using random-class-method with 20 rounds. In our evaluation, we find that the random-class-method configuration detects the most flaky tests.

We implement iDFlakies as a Maven plugin that can be integrated into any project that builds using Maven [131] and runs tests using JUnit [96] (specifically, iDFlakies supports JUnit versions 3 to 5). A Maven project is organized into one or more *modules*, and each module contains its own code and tests. When we refer to *modules* in this dissertation, we do not refer to the Java Platform Module System [88], but instead refer to Maven modules, which are (sub)directories that organize code under test and test code, with no particular visibility/access guarantees imposed by the compiler or runtime.

Like most Maven plugins, iDFlakies runs separately on each module. iDFlakies uses our

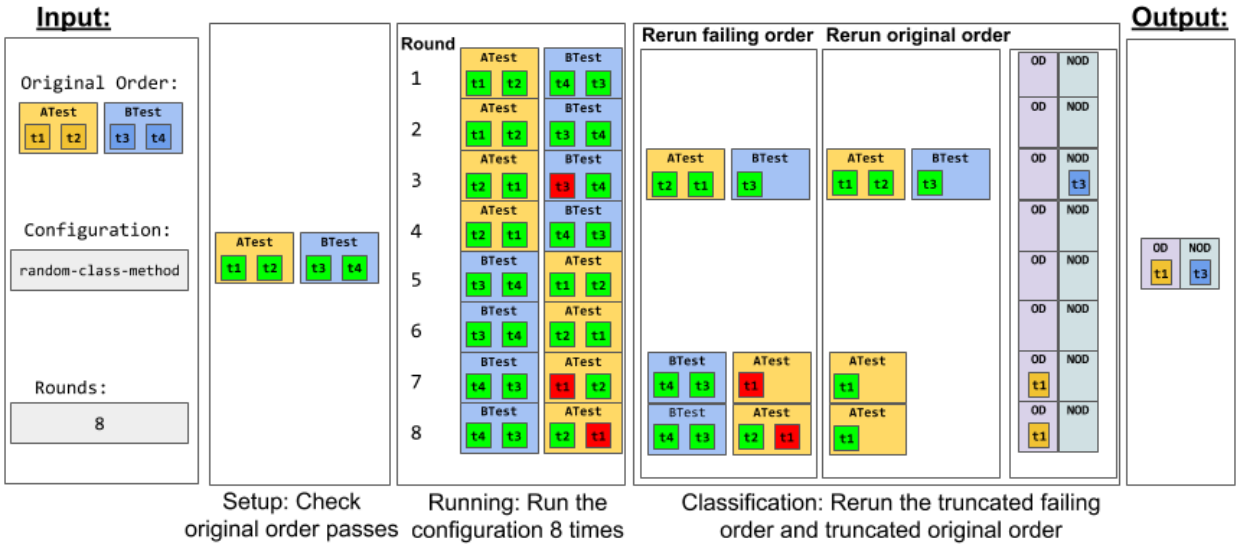


Figure 2.1: A sample run of iDFlakies using the random-class-method configuration with 8 rounds, detecting an OD test and an NOD test.

own custom test runner to control the order of running JUnit test methods, hence, iDFlakies can work on only Maven projects whose tests are written using JUnit. There are three main steps in iDFlakies. The *setup* step checks whether all tests of a module pass or not; if not, iDFlakies stops further exploration for that module. If all tests pass, the module proceeds to the next step. The *running* step runs the module's test suite based on the user-specified configuration and the number of rounds. For each round that contains some test failure(s), iDFlakies performs the classification step. The *classification* step reruns failing and passing orders of a test to classify it as OD or NOD.

During the setup step, iDFlakies checks whether all tests pass in the *original order*. To determine this order, iDFlakies runs Maven's unit-test plugin, Surefire [132] and collects the standard output (stdout) from running Surefire and the `.xml` files outputted by Surefire. From the standard output, iDFlakies extracts the order in which Surefire ran the test classes, and from the `.xml` files, iDFlakies extracts the order in which Surefire ran the test methods within each test class. Even if the tests pass with Surefire, they could fail with our plugin that uses our custom test runner. Thus, our tool runs the tests in the original order using our custom runner, and checks if the result of every test is `PASS` or `SKIP`. `SKIP` indicates that a developer intentionally ignores the test. When run with our test runner, a test could fail in the original order because the test is flaky, but also because of several other factors, including our testing environment being wrong, our tool having limitations, or the code under test being actually broken. We cannot easily distinguish these factors. In an attempt to get all tests to pass, even in the presence of some NOD tests, our tool runs the original order up to a user-specified number of times (by default three). If every run has some failing test(s),

our tool currently discards the module. In the future, we plan to improve how iDFlakies handles failing tests, e.g., it could remove failing tests from the test suite and proceed with the remaining tests. In our evaluation, the original order does pass for the majority of the modules (945 modules pass, 476 modules do not pass).

Figure 2.1 shows an example run of iDFlakies, using the random-class-method configuration and 8 rounds. In the setup step, the tool runs the original order and all four tests pass. In the running step, the tool runs these tests 8 times based on the specified configuration. In the end, it detects two flaky tests: an OD test `t1` from the `ATest` class (`ATest.t1`) and an NOD test `t3` from the `BTest` class (`BTest.t3`).

To classify each failed test, the classification step reruns two test orders: (1) the *truncated failing order* with all tests from the failing order up to and including the failing test; and (2) the *truncated original order* with all tests from the original order up to and including the failing test. If the test fails in the truncated failing order *and* passes in the truncated original order, our tool classifies the test as OD. If the test passes in the truncated failing order *or* fails in the truncated original order, our tool classifies the test as NOD. The classification reruns of the truncated failing order are critical to classify each test as OD or NOD; when a test fails in an order different from the original order (in which the test passed), the tool cannot immediately determine whether the test fails due to the test order change or due to some other flakiness. The reruns of the truncated original order are not cost-beneficial, and in our evaluation failed in only 3 of 7441 classification runs, so we recommend that only truncated failing orders be run.

In our example, `BTest.t3` fails in round `3`. In the classification step, when rerunning the truncated failing order, `BTest.t3` passes. Therefore, the tool classifies `BTest.t3` as an NOD test, because it failed and passed in the same order. In contrast, `ATest.t1` fails in rounds `7` and `8`. In round `7`, when rerunning the truncated failing order, `ATest.t1` fails again, and when rerunning the truncated original order, `ATest.t1` passes. Therefore, the tool classifies `ATest.t1` as an OD test.

Even if a test fails twice in the same order, it is no guarantee that the test is really OD, because other factors could have made an NOD test to fail twice. For example, the test shown in Figure 1.3 could time out twice in a row due to the machine load, independent of the test order. iDFlakies can recheck a test failure again even if it previously classified the test. A test classified as OD can be later reclassified as NOD in a future round. However, a test classified as NOD can never be reclassified as OD. In our example, the same test `ATest.t1` fails in round `8` and is classified again as an OD test.

2.1.1 Configurations

iDFlakies has five configurations for ordering tests:
(1) **original-order** repeatedly runs tests in the original order and classifies any failing test as NOD. This configuration cannot detect OD tests, because the order is always the same.
(2) **random-class** (*RandomC*) repeatedly runs test classes in a random order but keeps methods in each class in the same order as in the original order (e.g., orders in rounds `2` and `5` from Figure 2.1). Maven Surefire can already randomize the order of test classes, but it neither runs the test suite repeatedly nor classifies flaky tests as OD or NOD.
(3) **random-class-method** (*RandomC+M*) repeatedly runs test methods in a random order, hierarchically randomizing first the order of the test classes and then the methods within test classes but *not* interleaving methods from different classes. The 8 rounds in Figure 2.1's running step illustrates this configuration.
(4) **reverse-class** (*ReverseC*) reverses the order of all test classes from the original order but keeps the test methods in the same order as the original order (e.g., order `5` from Figure 2.1); iDFlakies runs this configuration only once to limit the time for experiments (although repeated runs could detect some more NOD tests but no new OD tests).
(5) **reverse-class-method** (*ReverseC+M*) reverses the order of all test classes and methods from the original order (e.g., order `8` from Figure 2.1); similar to reverse-class, this configuration runs only once.

All configurations, except the original-order, reorder some tests from the original order and can detect OD tests. For these configurations, if the tool finds a failing test, it proceeds to the classification step (Section 2.1.2). The original-order configuration skips the classification step because all failing tests from this configuration are classified as NOD tests.

2.1.2 Classification

When iDFlakies finds a test failure in an order (called failing order) different from the original order (in which the test passed), it needs to classify whether the test is OD or NOD. For this classification, iDFlakies can run the test again in the failing order and in the original order. If the test both fails again in the failing order and passes again in the original order, iDFlakies classifies the test as an OD test. Otherwise, iDFlakies classifies it as an NOD test.

The test classification can happen in two stages. When a test fails for the first time, its classification is unknown, so the classification step must be run. If the same test fails later, its prior classification is known (OD or NOD), so one need not run the classification step again. However, we allow a certain percentage of failures to be *rechecked*, i.e., the

classification step is rerun although the prior classification is known. If this percentage is 100%, the classification step runs for every failing test, with a potential high runtime cost. If this percentage is 0%, no test is rechecked, increasing the chance to mis-classify some NOD tests as OD. If this percentage is in between, then each failing test is selected with that percentage to be rechecked. In our experimentation, we use 20% to control the runtime cost but still have some benefit of increased accuracy. We find that 29 tests out of 242 (29 + 213 OD tests) are first mis-classified as OD tests and later re-classified as NOD tests. For greater accuracy in classification, we recommend setting this percentage to 100% when using our tool with spare machine time available (e.g., overnight or over the weekend).

If iDFlakies ever classifies a test as NOD, including during rechecking, it overall classifies the test as NOD, even if some classifications were OD. In other words, the tool classifies as NOD all tests that fail non-deterministically for some order, even if they fail largely deterministically in other orders and thus have characteristics of both types of flaky tests. Many NOD tests fail in more than one round (in our evaluation, 125 out of 209 NOD tests fail more than once), so even if the test is incorrectly classified as OD in one round, later rechecking can likely correctly re-classify the test as NOD.

2.1.3 Rounds and Timeouts

iDFlakies can be set to run for a specified number of rounds (*Rounds*) for each module of a project, a specified amount of time (*Timeout*) for an entire project, or the minimum of the two (*Both*). We expect that developers would use Rounds, because they know how long their test suite runs, but we used Both in our large-scale experiments, because we did not know a priori how long various test suites run.

Rounds. Given a number of rounds, the tool runs each module for that number of rounds before proceeding to the next module. Section 2.4.4 discusses the trade-off between running modules "depth-first" vs. "breadth-first".

Timeout. Given a total amount of time, the tool computes the number of rounds to run as $\lfloor T_{\text{timeout}}/T_{\text{original}} \rfloor$, where T_{original} is the time the original order took to run the entire project.

Both. Given both a number of rounds and a timeout, iDFlakies first calculates the number of rounds with the given Timeout, and then chooses the minimum of that calculated number and the given number of rounds.

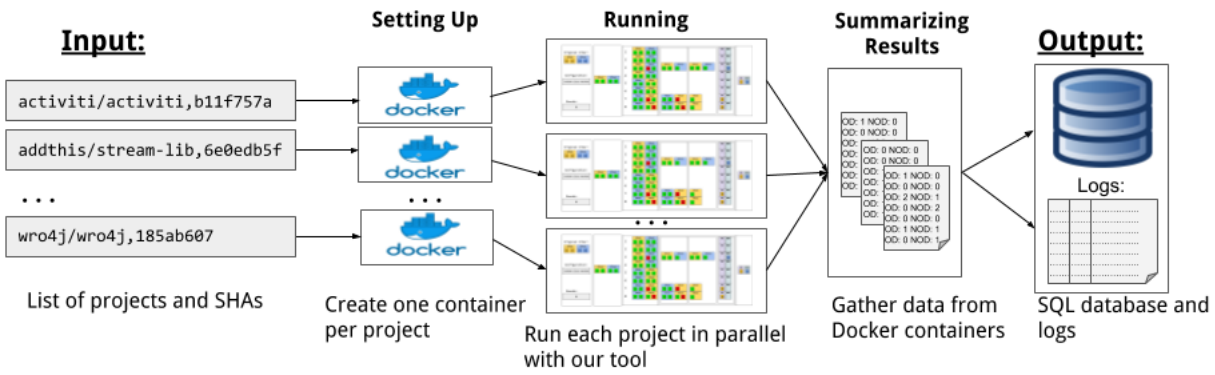


Figure 2.2: Overview of the end-to-end framework.

2.2 END-TO-END FRAMEWORK

In addition to our iDFlakies tool, we also develop a framework for using iDFlakies on various projects. At a high level, the framework takes as input a list of project URLs and commits, and outputs a database with various information including how long a module's test suite takes to run, in which configurations a module's test suite is run, and the OD and NOD tests detected for each configuration. Figure 2.2 shows an overview of the framework. It has three main steps: (1) *setup* of the projects, (2) *running* iDFlakies on the projects to detect flaky tests, and (3) *summarizing* the results for the user. The code for all three steps is publicly available [85].

2.2.1 Setup Step

Given a list of project URLs and Git SHAs corresponding to a commit for each project, our framework first constructs a Docker image [37] for each project and commit pair. Each image provides an isolated environment for each project and eases the reproduction of our experimentation. Our Docker images are also publicly available [85].

Our framework first builds a base Docker image on top of Ubuntu 16.04 by installing the basic necessary software such as Git, Java, and Maven. In particular, our framework currently uses Java 8 and Maven 3.5.4. On top of this base Docker image, the framework builds a Docker image for each project by cloning the version of the project's repository specified by the commit SHA. The framework then builds the commit SHA and runs its tests, specifically with `mvn clean install -DskipTests -fn -B` followed by `mvn test -fn -B`. Our framework aims to run as many modules as possible, and the `-fn` option instructs Maven to not stop at the first failing module but still execute the other modules. Modules that fail `mvn test` do not proceed to the running step.

2.2.2 Running Step

The running step runs iDFlakies for each project in its own Docker container. The framework starts up a Docker container for each Docker image and first modifies the project's `pom.xml` files (the build configuration files for a Maven project) to include iDFlakies. Next, the framework determines the number of rounds to run iDFlakies; in the Timeout or Both modes (Section 2.1.3), the framework finds the time that Maven took to run all the tests in the setup step and uses that time to compute the number of rounds. The framework then proceeds to run iDFlakies for each tool configuration that the user specified.

2.2.3 Summarizing Step

While the running step logs various information from the projects into log files, the summarizing step parses these logs to create a SQL database. The database contains several tables that allow easily querying the details for each module, for all modules of a project, or even across projects. The user can obtain information such as the time a module's test suite takes to run, the various configurations the module was run with, the number of rounds that iDFlakies runs for each configuration, the rounds that contain at least one failing test and the names of those failing tests, the results of the classification steps, which round detected which flaky test, and whether each test was classified as OD or NOD. All of the logs used to create the database are also saved, including test orders, test results, stack traces of failed tests, output from tests, and build output. More details about the database and logs are on our website [85].

2.3 STUDY SETUP

All projects in our study are Java projects that build with Maven [131] and use JUnit [96]. We check whether a project builds with Maven by looking for a `pom.xml` file at the root of the project's repository. We collected the projects from three sources: (1) 44 projects from related work [18], (2) 150 most popular Java projects from GitHub [63] up until October 2018, and (3) 500 most popular Java projects from GitHub that were updated in November 2018. We determine the popularity of GitHub projects using the number of stars.

The projects from related work [18] are prominent Java projects that have flaky tests. Instead of using the same, mostly old, versions of the projects used in prior work [18], we use a more recent version, because we may report the flaky tests that we detect to the project developers, and researchers may want to study not-yet-fixed flaky tests, e.g., such that tools

for automated fixing do not overfit to the history. In total, we use 44 projects from the two papers [18]. When we union those projects with the top 150 most popular projects from GitHub, we obtain 183 projects that contain a total of 2921 modules and 1880362 tests.

We break our projects into two sets, *comprehensive* and *extended*. The comprehensive set includes all 183 projects from sources (1) and (2), and we evaluate all five configurations of iDFlakies on these 183 projects. We find random-class-method to be the most effective configuration for detecting flaky tests. The extended set includes all of the projects from source (3), and to limit the cost of our experimentation, we evaluate only the random-class-method configuration on these projects. The extended set consists of 500 projects disjoint from the projects from the comprehensive set. These 500 projects contain a total of 2250 modules and 93722 tests. The extended set has fewer tests than the comprehensive set although the extended set has more projects, because it has relatively smaller projects.

Of all 5171 modules from the 683 projects, our framework is able to explore 945 modules for flaky tests. Our framework cannot explore the other 4226 modules (from 597 projects) because 462 modules could not be built by Maven, 2830 modules do not declare JUnit as a dependency in their `pom.xml` files or have no tests, 476 modules' tests do not pass in any of the three original-order rounds, and 458 modules encounter some limitations of iDFlakies.

In summary, among the 945 modules that our framework can explore, it detects 38 modules (from 31 projects) with at least one OD test and 82 modules (from 63 projects) with at least one NOD test, for a union of 111 modules (from 82 projects) with at least one OD or NOD test (and some modules have both OD and NOD tests). Our project website [85] provides more details for all of the projects used in our study.

2.4 STUDY RESULTS

The main goal of our study is to detect flaky tests in open-source projects and to compare the configurations that one could use to detect these tests. More specifically, our study addresses the following main research questions:

RQ1: What is the breakdown of OD and NOD tests in open-source projects?

RQ2: What is the probability of a round (test-suite run) containing at least one flaky-test failure?

RQ3: What ordering configurations detect the most flaky tests?

The reason for studying RQ1 is to understand which types of flaky tests are more prevalent among those detected in open-source projects. The reason for studying RQ2 is to understand how often flaky tests impact developers' development cycle and to understand the need for better solutions to detect flaky tests. The reason for studying RQ3 is to help developers

understand the potential trade-offs of different ordering configurations and better utilize their resources (e.g., developers' time and machine resources) in detecting flaky tests.

As described in Section 2.3, our dataset contains two sets: for comprehensive, we run all five configurations on the 183 projects; for extended, we run only the random-class-method configuration on the 500 projects. RQ1 (Section 2.4.1) uses both sets of our dataset, while RQ2 and RQ3 (Section 2.4.2 and Section 2.4.3, respectively) use only the comprehensive set, because they compare the configurations.

2.4.1 RQ1. Breakdown of Flaky-Test Types

Our evaluation detects a total of 422 flaky tests from 111 modules in 82 projects. Of 422 flaky tests, 213 (50.5%) are classified as OD tests and 209 (49.5%) as NOD tests, based on the observed runs. While the overall percentage of OD tests is slightly higher than the percentage of NOD tests, the two are rather close. Note that our study heavily focuses on randomizing test orders to detect OD tests, because automatically distinguishing/classifying OD tests from NOD tests can be done fairly well. Section 2.4.3 describes the breakdown of the flaky tests detected for each test reordering configuration. The projects in our study likely have many more NOD tests that could be detected by changing some aspects of our experiments. For example, running multiple test suites in the same machine (not each in its own machine) would allow competing for machine resources to more likely cause failures of NOD tests.

2.4.2 RQ2. Probability of a Round Containing a Flaky-Test Failure

The probability that an individual flaky test fails—measured as the ratio of the number of rounds in which a test fails over the number of rounds in which the test was run—varies a lot, from under 1% to over 50% in our experiments. In practice, a developer running tests usually cares not about individual tests but the status of the entire test suite, i.e., whether all the tests pass or some fail. Given this concern, we study the probability of a round failing, i.e., containing at least one flaky-test failure. Figure 2.3 shows for each configuration the percentage of failing rounds further broken down into the percentages for rounds that contain at least one OD or NOD test.

The percentage of failing rounds can be calculated assuming the test failures to be either (1) correlated with one another or (2) independent of one another. If failures are correlated, then rounds with multiple failures affect the percentage as much as rounds with one failure. If failures are independent, then one round with multiple failures could have been multiple

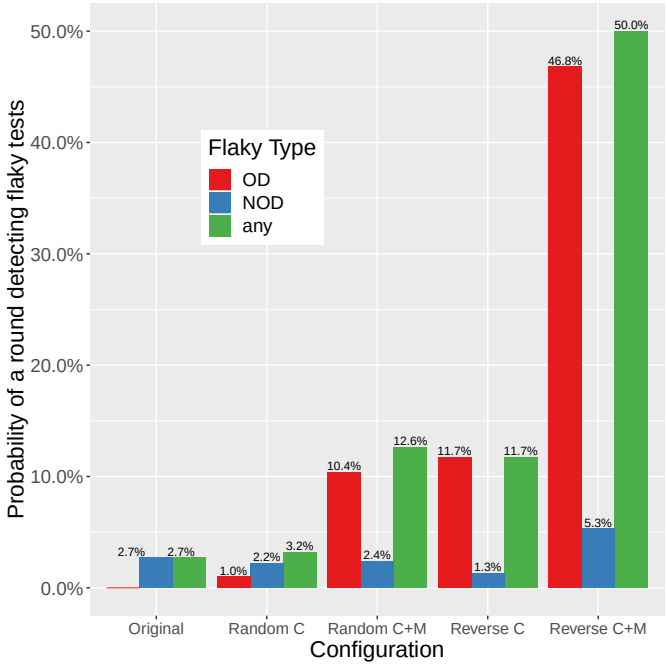


Figure 2.3: Probability of a configuration to detect at least one flaky test for modules that have at least one flaky test.

rounds with fewer failures per round. Due to the difficulty of precisely determining whether failures are independent, we compute percentages for round failures simply based on the observed rounds. Namely, we compute the percentages as the ratio of the number of rounds where one or more flaky tests fail over the total number of rounds for each configuration (but only for modules that have flaky tests). If a failing round contains both OD and NOD tests, then that round counts as one for both types of flaky tests as well as for "Any". In Chapter 3 and Chapter 6, we further study how often failures of different flaky tests are related to one another.

Figure 2.3 visualizes the results. We see that the reverse-class-method configuration has the highest probability of producing a failing round, 50.0% overall probability of detecting one or more flaky tests just from running one round. More precisely, reverse-class-method has a 46.8% probability of producing a failing round due to OD tests and a 5.3% probability of producing a failing round due to NOD tests. For quickly determining whether a test suite may contain flaky tests, our results suggest that the developers should run reverse-class-method. In Chapter 3, we expand on this recommendation by running the reverse-class-method of the last order instead of another random order if all tests in the last order pass. We also find that developers who run their tests only in the Maven-specified, original order have a low overall probability of producing a failing round, 2.7%.

Of particular interest are the percentages of rounds that fail due to OD tests for random-

class and random-class-method. Intuitively, an OD test can fail because either a "bad" test is run before it (polluter) or a "good" test is not run before it (state-setter). Consider the case where an OD test fails due to some "bad" test(s) running before the OD test and "polluting" the shared state, causing the OD test to start running in an undesirable state [16, 17, 73]. Assume that there is one such polluting test and one OD test. Given a uniformly random ordering of the tests, there is a 50% probability of the polluting test to be ordered before the OD test. If there are more polluting tests, the probability is even higher that at least one polluting test runs before the OD test. However, with the exception of the reverse-class-method configuration having a 46.8% probability of a failing round, our reordering configurations have the percentages much lower than 50%. These low percentages suggest that the test suite has some "cleaner" tests, which clean the polluted state such that the OD test can then run successfully, and these cleaner tests are frequently ordered to run between the polluting test(s) and the OD test.

The other case is a missing "good" test (state-setter): if an OD test needs another test to run before it to set up a desirable state for that OD test, then not having that set up test run before the OD test causes the OD test to fail. The probability of failure should be again 50% unless there are many tests that can set up the OD test. We further explore the notions of "bad" and "good" tests in another work [186] that is not included in this dissertation.

2.4.3 RQ3. Configurations Detecting Most Flaky Tests

Table 2.1 shows the breakdown of the number of flaky tests detected by the different configurations for each project from our comprehensive set. The table shows the breakdown for both OD and NOD tests, except for the original order that can detect only NOD tests. The table also shows the number of rounds run for the original-order and random-class-method configurations; the number of rounds for random-class is similar to the number for random-class-method. While these numbers would be ideally the same, there are various reasons for different numbers, including timeouts, tool crashes, and repeated experiments. The numbers of rounds for reverse-class-method and reverse-class are much lower; in fact, iDFlakies runs each of those two configurations for only one round in one experiment, but we performed multiple experiments while developing iDFlakies and kept most of the logs to provide the largest dataset for analysis of flaky tests.

OD tests: As shown in Table 2.1, among all configurations, the random-class-method detects the greatest number of unique OD tests, 162 (i.e., 88.0% of all OD tests detected across all configurations). This result matches our expectations: randomly reordering test methods provides the most reordering flexibility among configurations, giving more oppor-

Table 2.1: The number of flaky tests that each configuration detects in the comprehensive set. "All" is the number of unique tests.

	Original		RandomC		RandomC+M			ReverseC		ReverseC+M		All		
Project Slug - Module	**Round**	**NO**	**OD**	**NO**	**Round**	**OD**	**NO**	**OD**	**NO**	**OD**	**NO**	**OD**	**NO**	**All**
activiti/activiti	88	0	0	0	66	20	0	0	0	0	0	20	0	20
alibaba/fastjson	67	0	12	0	158	13	2	3	0	4	0	13	2	15
apache/hadoop - m1	14	0	0	0	14	2	0	18	10	0	0	20	10	30
- m2	14	0	22	1	7	22	1	0	0	0	0	22	1	23
- m3	15	0	1	0	10	1	0	1	0	1	0	1	0	1
- m4	15	1	0	0	14	2	0	0	0	4	7	6	8	14
apache/hbase	14	1	0	0	13	0	1	0	0	0	1	0	1	1
apache/incubator-dubbo - m1	32	0	0	1	53	0	0	0	0	0	0	0	1	1
- m2	33	0	2	7	76	4	2	0	0	1	0	4	9	13
- m3	39	0	0	0	37	1	0	0	0	1	0	1	0	1
- m4	42	0	1	0	91	4	0	0	0	2	0	4	0	4
- m5	47	0	0	0	38	3	0	0	0	0	0	3	0	3
- m6	131	2	0	0	49	0	0	0	0	0	0	0	2	2
apache/jackrabbit-oak	16	0	0	0	14	2	0	0	0	2	0	2	0	2
apache/struts	114	0	0	0	342	4	0	0	0	0	0	4	0	4
crawlscript/webcollector	4140	1	0	1	16503	0	1	0	0	0	0	0	1	1
doanduyhai/achilles	356	0	0	0	278	0	1	0	0	0	0	0	1	1
dropwizard/dropwizard	76	0	0	1	248	1	1	0	0	0	0	1	1	2
elasticjob/elastic-job-lite - m1	288	2	0	0	839	0	0	0	0	0	0	0	2	2
- m2	307	3	0	0	826	0	1	0	0	0	0	0	3	3
- m3	335	0	2	0	815	7	1	0	0	2	0	7	1	8
google/jimfs	42	1	0	0	108	0	0	0	0	0	0	0	1	1
jfree/jfreechart	166	0	0	0	290	1	0	0	0	0	0	1	0	1
jodaorg/joda-time	206	1	0	0	146	0	0	0	0	0	0	0	1	1
kevinsawicki/http-request	2317	0	0	0	2013	28	0	0	0	28	0	28	0	28
knightliao/disconf	344	0	0	0	1359	0	1	0	0	0	0	0	1	1
looly/hutool	842	0	0	0	650	0	1	0	0	0	0	0	1	1
orbit/orbit	35	0	0	1	123	0	0	0	0	0	0	0	1	1
oryxproject/oryx	60	1	0	0	131	0	1	0	0	0	0	0	1	1
querydsl/querydsl	14	3	0	0	0	0	0	0	0	0	0	0	3	3
spotify/helios	25	1	0	1	71	0	1	0	0	0	0	0	1	1
spring-projects/spring-boot - m1	8	1	0	0	12	0	0	0	0	0	0	0	1	1
- m2	12	0	0	0	13	2	0	0	0	2	0	2	0	2
square/otto	815	1	0	0	3243	0	0	0	0	0	0	0	1	1
square/retrofit - m1	87	0	0	2	331	0	2	0	0	0	0	0	2	2
- m2	87	0	0	4	331	0	5	0	0	0	0	0	7	7
tootallnate/java-websocket	666	21	0	30	1653	0	47	0	0	0	1	0	52	52
undertow-io/undertow	15	0	0	4	65	1	3	0	0	0	0	1	4	5
wildfly/wildfly	10	0	0	0	35	44	0	0	0	0	0	44	0	44
wro4j/wro4j	34	0	0	0	116	0	2	0	0	0	0	0	2	2
Total	**11968**	**40**	**40**	**53**	**31181**	**162**	**74**	**22**	**10**	**47**	**9**	**184**	**122**	**306**

tunities for different reorderings to expose OD tests. In general, Table 2.1 shows that reordering test methods rather than just test classes helps with detecting flaky tests; both random-class-method and reverse-class-method detect more flaky tests than random-class and reverse-class, respectively (while the corresponding configurations explore a similar number of rounds).

Considering that the random-class-method configuration runs many more rounds than the reverse-class-method configuration, it is expected that random-class-method detects more (OD and NOD) flaky tests. Indeed, the reverse-class-method configuration detects only 47 OD tests (25.5% of all OD tests detected across all configurations). Interestingly, the reverse-class-method configuration detects 4 tests not detected by the random-class-method config-

uration. However, the random-class-method configuration detects 119 tests not detected by the reverse-class-method configuration. As a result, we strongly recommend developers to first run the reverse-class-method configuration once to quickly detect a portion of the OD tests and then use the random-class-method configuration to detect more OD tests.

Overall, we find that the random-class-method configuration performs the best, although the other configurations also sometimes detect flaky tests not detected by random-class-method. Our findings suggest that it is desirable to research new approaches that can help quickly find the test orders that would detect the most OD tests. Such new approaches could be substantially better than randomly selecting test orders. For example, we present a cost-effective, systematic way to explore all consecutive test pairs, guaranteeing the detection of all OD tests that depend on one other test in Chapter 3.

NOD tests: Table 2.1 and Figure 2.3 show that most configurations have similar probability to detect NOD tests. The percentages for the two reverse configurations differ from the other configurations, but these two configurations have much fewer rounds and thus by chance could have much higher or lower probabilities. Even if some configuration has a higher probability to detect at least one failure in a round, it may be repeatedly detecting the same NOD tests. A benefit of rerunning original-order is that every failure is immediately known to be an NOD test. In contrast, failures from randomized orders need to be classified using the classification step of iDFlakies.

Detection of many NOD tests from randomized orders (and the reverse-class-method classification) shows that the classification step is important for properly classifying a flaky test as an OD test or an NOD test. Of the 122 NOD tests in our comprehensive set, we find that iDFlakies classifies 91 as NOD tests using the classification step; the remaining 31 unique NOD tests need no classification step because iDFlakies classifies them as NOD using the original-order configuration. For NOD tests detected by both original-order and random-class-method, we compare the probability of a round detecting the test but find no generalizable differences. Specifically, a Wilcoxon signed-rank test shows that the probabilities are statistically different, with $p < 0.05$, for the (26) tests in the comprehensive set but not statistically different for the (33) tests including both comprehensive and extended sets. In Chapter 6, we further explore whether running the tests in different reorderings leads to differences that can more easily expose flakiness in NOD tests.

Our results suggest that simply rerunning tests in the original order where they pass is *not* a good configuration for detecting flaky tests—it cannot detect any OD test, and it does not have a much higher probability to detect even NOD tests. It is better to reorder the tests to increase the probability of detecting any type of flaky tests, not just OD tests. In our experiments, the randomizing configurations, along with the classification step, detect

more NOD tests than rerunning the tests many times in the same original passing order (but with the caveat that randomizing configurations had more rounds). Our tool currently cannot further analyze or classify the cause of flakiness for these NOD tests; we leave that topic as important future work.

2.4.4 Results Discussion

Running iDFlakies. Currently, iDFlakies runs tests in a multi-module Maven project in a *depth-first* manner: given a user-specified number of rounds (or a user-specified timeout from which the tool calculates the number of rounds), iDFlakies first runs that number of rounds for one module before proceeding to the next module. An alternative would be *breadth-first*: our framework would first run iDFlakies on every module once before running iDFlakies on every module again for the second round, and so on. However, breadth-first would invoke iDFlakies, and consequently Maven, each time it needs to run through all modules for one round. Invoking iDFlakies and Maven adds extra overhead in checking what modules exist, what needs to be rebuilt, what the tests are, etc. Comparing advantages and disadvantages of depth-first and breadth-first, depth-first avoids the extra overhead of invoking Maven multiple times and more closely matches the usual Maven approach to plugins, with a plugin finishing work on a module before proceeding to the next module. The disadvantage is that depth-first requires knowing the number of rounds, so iDFlakies can finish running tests for one module before proceeding to the next one. The advantage of breadth-first is that it allows developers to run the framework with no a priori timeout, running overnight or whenever a machine has idle time. The developer can then stop the framework at any time and receive all of the flaky tests detected. The disadvantage of breadth-first is the extra overhead needed for Maven. Currently, we do not know which way of running modules is faster and provides more benefits in terms of detecting more flaky tests; we plan to implement breadth-first and compare empirically with depth-first in the future.

Regression Testing. iDFlakies runs tests in many different orders aiming to detect the most flaky tests. However, rerunning tests takes a long time and is worth doing only if a developer is purposefully trying to detect a substantial number of flaky tests and has the resources for this task. Another way to use the findings from our study (e.g., that changing the order of the tests increases the chances of detecting flaky tests) is to incorporate the reorderings with continuous integration and regression testing. The developer can run the tests in different orders after every change when tests are naturally rerun as part of the development process, and flaky-test detection from iDFlakies would effectively come "for

free". In fact, we find that 8 of the 683 projects from our study already configure their Surefire (setting the option `runOrder` to `random`) to run test classes (but not test methods) in random order.

First Failure. Our framework counts *all* tests that fail during a failing round as flaky tests. However, multiple flaky tests that fail in the same failing round can all be failing due to the same root cause. As such, multiple flaky tests can all be *fixed* in the same way, and the number of fixes may be smaller than the number of flaky tests. For example, in a run with multiple failing tests, all failing tests may be classified as OD, but the tests after the first failure simply depend on the first failing test. When that first failing test is fixed, these later OD tests may also all be fixed. In fact, in our iFixFlakies work [186], we indeed find such an OD test. Namely, if this test fails, then 43 other OD tests would also fail, but if this test passes, then the other 43 OD tests would also pass.

Ratio of Types of Flaky Tests. Our results show that the percentages of flaky tests classified as OD and NOD are quite close (50.5% and 49.5%, respectively). However, prior work [126] classifying fixed flaky tests found a much lower percentage of flaky tests being OD, 12%. iDFlakies uses random orderings to focus on detecting OD tests. iDFlakies likely misses many NOD tests and can be improved by adding more variations to test runs to detect more NOD tests.

FixMethodOrder. We find that 23 of the OD tests detected by iDFlakies are in test classes annotated with `@FixMethodOrder`. This annotation indicates that the test methods in a test class must run in a certain order, e.g., in the ascending order based on the test-method names. iDFlakies still detects and reports such OD tests although running them through JUnit would not reorder the tests. However, it is still beneficial to explore different orderings of test methods in such annotated test classes. First, it could be that there are actually no dependencies among the tests, so the annotation is no longer needed and can be removed. Second, it is important to still detect OD tests to help developers know which exact tests are OD. For example, we observe that while iDFlakies detected several OD tests in `@FixMethodOrder`-annotated test classes from the `Activiti/Activiti` project [6] at commit SHA `b11f757a`, the developers introduced a patch that removed the ordering of such dependencies and the `@FixMethodOrder` annotation in a later commit SHA, `5a1cb8ae`.

2.5 THREATS TO VALIDITY

Our iDFlakies tool and framework may contain faults that could have affected our results. To mitigate such threat, we implement extensive logging for our framework and manually investigate a sample of logs generated on a variety of projects. We are more confident in our tool but less confident in the results of the framework due to its complexity. For example, the output of a SQL database to store and process the results of our framework can add much complexity (e.g., joining of tables, creating SQL queries) without providing much benefit. To improve upon this issue, subsequent work of ours using the iDFlakies flaky-test dataset (e.g., chapters 3, 4, 6, and 7) simply relies on CSV files and no longer relies on a SQL database.

The exact results of our study, namely the flaky tests detected and their rate of failures, may not be easily reproducible due to the nature of our experimentation using random orders and the nature of flaky tests non-deterministically passing and failing. We attempt to mitigate this threat by logging the (random) orders in which iDFlakies runs the tests so that others can reproduce the flaky-test behavior by running the same orders. The logs for all rounds are publicly available [85].

Our classifications of flaky tests into OD or NOD tests may occasionally be incorrect. For example, an NOD test could fail due to a timeout or network issue, and rerunning in the classification step could lead to it failing again in the same order, misleading iDFlakies to classify the test as an OD test. We attempt to mitigate this threat by having the framework recheck a substantial number of flaky tests' classifications.

Moreover, the actual number of flaky tests in the projects that we study may be (much) higher than what we report. For example, we find more OD tests in Chapter 3 and more NOD tests in Chapter 4 and Chapter 6 using the same projects. Also, we currently run only unit tests from `mvn test` and not integration tests from `mvn verify` because the latter can take much longer.

Our findings that random-class-method detects the most flaky tests among all configurations that we study may not generalize to projects other than those we study. We attempt to mitigate this threat by obtaining a sizable number of popular Java projects from GitHub and prior studies. Nevertheless, projects written in other languages [167], or even Java projects not using Maven or JUnit, may not yield similar results. We use the number of stars on GitHub to obtain popular Java projects, but they may not be representative of the test suites in all Java projects.

2.6 SUMMARY

We have presented our *end-to-end framework*, which automates experimentation to detect and partially classify flaky tests using *iDFlakies* for Maven-based Java projects with JUnit tests. We have applied our framework on 683 projects. We provide a *dataset* of 422 flaky tests that we then use for our *study* on flaky tests. From our dataset, 50.5% of flaky tests are OD, while 49.5% are NOD, based on the observed runs. We also find that running the random-class-method configuration can detect the most flaky tests overall. Both our framework and dataset are publicly available [85], and we hope that they can help involve more researchers in the topic of flaky tests, e.g., to develop better techniques to detect flaky tests, reduce non-determinism or even fix it altogether, label test failures as flaky or not, or prevent future flaky tests.

CHAPTER 3: [DETECTING] PROBABILISTIC AND SYSTEMATIC COVERAGE OF CONSECUTIVE TEST-METHOD PAIRS TO DETECT ORDER-DEPENDENT FLAKY TESTS

This chapter presents a probability analysis of detecting OD tests and a systematic approach to guarantee the detection of OD tests that depend on one other test. Section 3.1 presents a more in-depth OD test example than the one presented in Section 1.2.1. Section 3.2 presents the preliminaries relevant for the contributions of this chapter. Section 3.3 presents our probability analysis of detecting OD tests and a simple change to iDFlakies to increase the probability of detecting such tests. Section 3.4 presents our systematic approach to guarantee the detection of OD tests that depend on one other test, and Section 3.5 concludes this chapter.

3.1 IN-DEPTH EXAMPLE OF ORDER-DEPENDENT (OD) TEST

Figure 3.1 shows a snippet of a victim test, `testMRAppMasterSuccessLock` (in short `testV`), from the widely used Hadoop project [9]. The test suite for this test has 392 tests. This test is from the MapReduce (MR) framework and aims to check an MR application. This test is a victim because it passes when run by itself but has two polluter tests. If the victim is run after either one of its polluter tests (and no cleaner runs in between the polluter and the victim), then the victim fails with a `NullPointerException`. Figure 3.2 shows a snippet of one of these two polluter tests, `testSigTermedFunctionality` (in short `testP`).

These tests form a polluter-victim pair because they share a global state, namely all "active" jobs stored in a *static* map in the `JobHistoryEventHandler` class. (In JUnit 4, only the heap state reachable from the class fields declared as static is shared across tests; JUnit does *not* automatically reset that state, but developers can add `setup` and `teardown` methods to reset the state.) To check an MR application, `testV` first sets up some state (Line 2), then creates an MR application (Line 3), and starts the application (Line 7). The `NullPointerException` arises when the test tries to stop the MR application (Line 10). Specifically, the `appMaster` accesses the shared map data structure that tracks all jobs run by any application. When `testV` is run after `testP`, then `appMaster` will attempt to stop a job created by the polluter, although the job has already been stopped.

This static map is empty when the JVM starts running, and it is also explicitly cleared by some tests. In fact, we find 11 cleaner tests that clear the map, and the victim passes when any one of these 11 tests is run between `testP` and `testV`. Interestingly, for the other polluter test, `testTimelineEventHandling` (in short `testP'`), the victim fails for the same reason, but

```
public void testMRAppMasterSuccessLock() { // testV for short
  ... // setup MapReduce job, e.g., set conf and userName
  MRAppMaster appMaster =
      new MRAppMasterTest("appattempt_...", "container_...", "host", -1,
        -1, System.currentTimeMillis(), false, false);
  try {
    MRAppMaster.initAndStartAppMaster(appMaster, conf, userName);
  } catch (IOException e) { ... }
  ... // assert the state and some properties of appMaster
  appMaster.stop();
}
```

Figure 3.1: Victim OD test from Hadoop's `TestMRAppMaster` class.

```
public void testSigTermedFunctionality() { // testP for short
  JHEventHandlerForSigtermTest jheh =
    new JHEventHandlerForSigtermTest(Mockito.mock(AppContext.class), 0);
  jheh.addToFileMap(Mockito.mock(JobId.class));
  ... // have jheh handle a few events
  jheh.stop();
  ... // assert whether the events were handled properly
}
```

Figure 3.2: Polluter test from Hadoop's `TestJobHistoryEventHandler` class.

`testP'` has 31 cleaners—the same 11 as `testP` and 20 other cleaners. Our manual inspection finds that the `testP'` polluter has other cleaners because the job created by `testP'` is named `job_200_0001`, while the job created by the `testP` polluter is a mock object. The 20 other cleaners also create and stop jobs named `job_200_0001` and therefore act as cleaners for the `testP'` polluter but not the `testP` polluter. This example illustrates the complexity of victims and polluters and how these tests interact with cleaners.

Unlike failure rates, which focuses on one specific test order and are defined in Section 3.3.2, we next explore how to compute the *flake rate* for a victim test, i.e., the probability that the test fails in a randomly sampled test order of all tests in the test suite. For this example, the 392 tests could, in theory, be run in 392! ($\sim 10^{848}$) test orders (permutations), but in practice, JUnit never interleaves test methods from different test classes. These tests are split into 48 classes that actually have $\sim 10^{234}$ test orders that JUnit could run. The relevant 34 tests (1 victim, 2 polluters, and 31 cleaners) belong to 8 test classes: 2 polluters belong to one class (`TestJobHistoryEventHandler`), 11 cleaners belong to the same class as the polluters, 1 cleaner belongs to the same class as the victim (`TestMRAppMaster`), and the remaining 19 cleaners belong to six other classes. For this victim, (uniformly) randomly sampling the orders that JUnit could run gives a flake rate of 4.5%. In Section 3.3.4, we propose a simple change to increase the probability of detecting OD tests by running a reverse of each passing test order. For this victim, the conditional probability that the reverse order fails is 4.9%.

3.2 PRELIMINARIES

We next formalize the concepts that we have introduced informally and define some new concepts. Let $T = \{t_1, t_2, \ldots, t_n\}$ be a set of n tests partitioned in k classes $\mathbb{C} = \{C_1, C_2, \ldots, C_k\}$. We use class($t$) to denote the class of test t. Each class C_i has $n_i = |\{t \in T \,|\, \text{class}(t) = C_i\}|$ tests.

We use $\omega(T')$ to denote a test order, i.e., a permutation of tests in $T' \subseteq T$, and drop T' when clear from the context. We use ω_i to denote the i-th test in the test order ω, and $|\omega|$ to denote the length of a test order as measured by the number of tests. We use $t \prec_\omega t'$ to denote that test t is before t' in the test order ω. We will analyze some cases that allow all $n!$ permutations, potentially interleaving tests from different classes. We use $\Omega_A(T)$ to denote the set of all test orders for T. Some testing tools [237] explore all these test orders, potentially generating false alarms because most testing frameworks [30, 96, 190, 204] do not allow all these test orders.

We are primarily concerned with *class-compatible* test orders where all tests from each class are consecutive, i.e., if class(ω_i) = class($\omega_{i'}$), then for all j with $i < j < i'$, class(ω_i) = class(ω_j). We use $\Omega_C(T)$ to denote the set of all class-compatible test orders for T. The number of such class-compatible test orders is $k! \prod_{i=1}^{k} n_i!$. Section 3.3.2 presents how to compute the flake rate, i.e., the percentage of test orders in which a given victim test (with its polluters and cleaners) fails.

Section 3.4 presents how to systematically generate test orders to ensure that all test pairs are covered. A *test pair* $\langle t, t' \rangle$ consists of two distinct tests $t \neq t'$. We say that a test order ω *covers* a test pair $\langle t, t' \rangle$, in notation cover($\omega, \langle t, t' \rangle$), iff the two tests are consecutive in ω, i.e., $\omega = \langle \ldots, t, t', \ldots \rangle$. Considering consecutive tests is important because a victim may not fail if not run right after a polluter, i.e., when a cleaner is run between the polluter and the victim. A set of test orders Ω covers the union of test pairs covered by each test order $\omega \in \Omega$. In general, test orders in a set can be of different lengths. Each test order ω covers $|\omega| - 1$ test pairs.

We distinguish *intra-class* test pairs, where class(t) = class(t'), and *inter-class* test pairs, where class(t) $\neq$ class(t'). Of the total $n(n-1)$ test pairs, each class C_i has $n_i(n_i - 1)$ intra-class test pairs, and the number of inter-class test pairs is $2 \sum_{1 \leq i < j \leq k} n_i n_j$. Each class-compatible test order of all T tests covers $n_i - 1$ intra-class test pairs for each class C_i and $k - 1$ inter-class test pairs.

We aim to generate a set of test orders Ω that cover all test pairs[1]. Note that because

[1]This problem should not be confused with *pairwise testing* [151], which typically aims to cover pairs of values from different test parameters.

each test is run *at most once* in one JVM run, covering test orders and test pairs has to be done with a *set* of test orders and cannot be done with just one very long order, e.g., using superpermutations [82]. If we consider $\Omega_A(T)$ that allows all test orders, we need at least n test orders to cover all $n(n-1)$ test pairs. When we have only one class or all classes have only one test, then all test orders are class-compatible. However, consider the more common case when we have more than one class and some class has more than one test. If we consider $\Omega_C(T)$ that allows only class-compatible test orders, we need at least $\max_{i=1}^{k} n_i$ test orders to cover all intra-class test pairs and at least $M = 2\sum_{1 \le i < j \le k} n_i n_j/(k-1)$ test orders to cover all inter-class test pairs; because $M > \max_{i=1}^{k} n_i$, we need at least M class-compatible test orders to cover all test pairs.

More precisely, we aim to generate a set of test orders Ω that has the lowest cost for test execution. The cost for each test order ω can be modeled well as a sum of a fixed cost Cost_0 (e.g., corresponding to the time required to start a JVM and load required classes) and a cost for each test (e.g., the time to execute the test method): $\text{Cost}(\omega) = \text{Cost}_0 + \sum_{t \in \omega} \text{Cost}(t)$. The cost for a set of test orders is then simply the sum of individual costs $\text{Cost}(\Omega) = \sum_{\omega \in \Omega} \text{Cost}(\omega)$. For example, a trivial way to cover all test pairs is with a set of test orders where each test order is just a test pair: $\Omega_p = \{\langle t, t' \rangle \mid t, t' \in T \wedge t \neq t'\}$; however, the cost is unnecessarily high: $\text{Cost}(\Omega_p) = n(n-1)\text{Cost}_0 + 2(n-1)\text{Cost}(T)$, where $\text{Cost}(T) = \sum_{t \in T} \text{Cost}(t)$.

To simplify, we can assume that each test in T has the same cost, say, Cost_1, and then $\text{Cost}(\Omega_p) = n(n-1)\text{Cost}_0 + 2n(n-1)\text{Cost}_1$. In the optimal case, each test order would be a permutation of n tests covering $n-1$ test pairs, and the number of test orders would be just $n(n-1)/(n-1) = n$. Therefore, the lowest cost is $\text{Cost}(\Omega_{opt}) = n\text{Cost}_0 + n^2\text{Cost}_1$, demonstrating that the factor for Cost_0 can be substantially reduced, while the factor for Cost_1 is nearly halved ($\frac{n}{2(n-1)}$). However, in most realistic cases, due to the constraints of class-compatible test orders and the big differences in the number of tests across different classes, we cannot reach the optimal case.

3.2.1 Dataset for Evaluation

Besides deriving some analytical results, we also run some empirical experiments on flaky tests from Java projects. As described in Chapter 4, our recent work [112] ran the iDFlakies tool on most test suites in the projects from the iDFlakies dataset [85] using the configurations recommended by our iDFlakies work [108] (Section 2.4.3). Specifically, we ran 100 randomly sampled test orders from $\Omega_C(T)$ and 1 test order that is the reverse order of what Maven Surefire [132] runs by default. Note that unlike our work in Section 3.3.4, where

we propose running a reverse test order of *every* test order where all tests passed, the one reverse order that we ran in our recent work [112] is run only once and not for every passing test order.

Each project in the iDFlakies dataset is a Maven-based, Java project organized into one or more *modules*, which are (sub)directories that organize code under test and test code. Each module contains its own test suite. For the remainder of this chapter, we use the 121 modules in which our recent work [112] found at least one flaky test (but not necessarily OD test). To illustrate diversity among these 121 modules, the number of classes ranges from 1 to 2215, with an average of 61, and the total number of tests ranges from 1 to 4781, with an average of 287. The number of tests per class ranges from 1 to 200, with an average of 4.8.

When we run some of the test orders generated by our systematic test-pair exploration as described in Section 3.4.2, we detect a total of 249 OD tests in 44 of the 121 modules. Of the 249 OD tests, 57 are brittles and 192 are victims. Compared to the OD tests detected in our prior work [108, 111, 112] that used the iDFlakies dataset, we find 44 new OD tests that have not been detected before. Of the 44 OD tests, 1 is brittle and 43 are victims. One of the newly detected victim tests (`testMRAppMasterSuccessLock`) is shown in Section 3.1.

3.3 ANALYSIS OF FAILURE RATE AND SIMPLE ALGORITHM CHANGE

We next discuss how to compute the flake rate for each OD test. Let T be a test suite with an OD test. Prior work [108, 111, 112, 237] would run many test orders of T and empirically compute the flake rate for each test as a ratio of the number of test failures and the number of test runs. However, failures of flaky tests are probabilistic, and running even many test orders may not suffice to obtain the true flake rate for each test. Running more test orders is rather costly in machine time; in the limit, we may need to run all $|T|!$ permutations to obtain the true flake rate for OD tests. To reduce machine time needed for computing the flake rate for OD tests, we first propose a new procedure, and then derive formulas based on this procedure. We finally show a simple change for sampling random test orders to increase the probability of detecting OD tests.

3.3.1 Determining Test Outcome Without Running a Test Order

We use a two-step procedure to determine the test outcome for a given OD test. We assume that some prior runs already detected the OD test, and the goal is to determine the test outcome for some new test orders that were not run.

In Step 1, we classify how each test from T relates to each OD test in a simple setting that runs only *up to three* tests. Specifically, we first determine whether an OD test t is a victim or a brittle by running the test in isolation, i.e., just $\langle t \rangle$, by itself 10 times: if t always passes, it is considered a victim (although it may be an NOD test); if t always fails, it is considered a brittle (although it may be an NOD test); and if t sometimes passes and sometimes fails, it is definitely an NOD, not OD, test. This approach was proposed for iFixFlakies [186], and using 10 runs is a common practice in proprietary software development for checking whether a test is flaky [137, 201].

We then find

- for each victim, *all* of its single polluters in T and also *all* single cleaners for each polluter, and
- for each brittle, *all* of its single state-setters in T.

To find polluters (resp. state-setters) of a victim (resp. brittle) test, iFixFlakies [186] takes as input a test order (of entire T) where the test failed (resp. passed) and then searches the prefix of the test in that test order using delta debugging [230] (an extended version of binary search). While iFixFlakies can find all polluters (resp. state-setters) in the prefix, it does not necessarily find all polluters in T, and it takes substantial time to find these polluters using delta debugging. The experiments show that in 98% of cases, binary search finds one test to be a polluter, although some rare cases need a *polluter group* that consists of two tests.

We propose a simpler and faster approach to find polluters (resp. state-setters) for the most common case: for each victim v (resp. brittle b) and each test $t \in T \setminus \{v\}$ (resp. $t \in T \setminus \{b\}$), we run a pair of the test and the victim (resp. brittle), i.e., $\langle t, v \rangle$ (resp. $\langle t, b \rangle$). If the victim fails (resp. brittle passes), then the test t is a polluter (resp. state-setter). Further, for each victim v, its polluter p, and a test $t \in T \setminus \{v, p\}$, we run a triple of $\langle p, t, v \rangle$, and if v passes, then t is a cleaner for the pair of v and p. Note that for the same victim v, different polluters may have different cleaners such as the example presented in Section 3.1.

In Step 2, we determine whether each OD test passes or fails in a given test order using only the abstraction from Step 1, without actually running the test order. We focus on victims because they are more complex than brittles; brittles can be viewed as special cases with slight changes (requiring a state-setter to *run* before a brittle to pass, rather than requiring a polluter *not* to run before a victim to pass). Without loss of generality, we consider one victim at a time. Intuitively, the victim fails in a test order if a polluter is run before the victim without a cleaner between the polluter and the victim. Formally, we define the test outcome as follows.

Definition 3.1 (Test Outcome from Abstraction)**.** Let T be a test suite with one victim $v \in T$, polluters $P \subset T$, and a family of cleaners $C_p \subset T$ indexed by each polluter $p \in P$. The outcome of v in a test order ω is defined as follows:

$$\text{fail}(\omega) \equiv \exists p \in P.\, p \prec_\omega v \wedge \nexists c \in C_p.\, p \prec_\omega c \wedge c \prec_\omega v; \quad \text{pass}(\omega) \equiv \neg\text{fail}(\omega). \tag{3.1}$$

This definition is an estimate of what one would obtain for all (repeated) runs of $|T|!$ permutations, for three main reasons:

- tests may behave differently in test orders than in isolation [111] (and an OD test may even be an NOD test in some orders [111]);
- polluters, cleaners, and state-setters may not be single tests but groups (iFixFlakies [186] reports that groups are rather rare); and
- a test that fails in some prefix may behave differently for the tests that come after it in a test order than when the test passes (again, iFixFlakies [186] reports this issue to be rare, finding just one such case).

Despite these potential sources of error, our evaluation shows that our use of abstraction obtains flake rates similar to iDFlakies for orders that iDFlakies ran. Most importantly, our use of abstraction allows us to evaluate many more orders without actually running them, thus taking much less machine time.

3.3.2 Computing Failure Rate

We next define flake rate, derive formulas for computing flake rate for two cases, and show why we need to sample test orders for other cases.

Definition 3.2 (Failure rate)**.** For a test suite T with exactly one victim, given a set of test orders $\Omega(T)$, the flake rate is defined as the ratio:

$$f(T) = |\{\omega \in \Omega(T) \mid \text{fail}(\omega)\}| \;/\; |\Omega(T)|; \tag{3.2}$$

we use the subscript f_A and f_C when we need to refer specifically to the flake rate for $\Omega_A(T)$ and $\Omega_C(T)$ (defined in Section 3.2), respectively.

We derive the formula for flake rate based on the number of polluters P and cleaners C for two special cases. In general, computing the flake rate can ignore tests that are not *relevant*,

i.e., not in $\{v\} \cup P \cup \bigcup_{p \in P} C_p$. It is easy to prove that $f(T) = f(T')$ if T and T' have the same victim, polluters, and cleaners—the reason is that the tests from $T \setminus T'$ are irrelevant in any order and do not affect the outcome of v. The further analysis thus focuses only on the relevant tests.

Special Case 1: Assume that (A1) all polluters have the same set C of cleaners: $C = C_p, \forall p \in P$; and (A2) all of the victim, polluters, and cleaners are in the same class: $\forall t, t' \in \{v\} \cup P \cup C.\text{class}(t) = \text{class}(t')$; it means that $\Omega_A(T) = \Omega_C(T)$ and $f_A = f_C$. Let $\pi = |P|$ and $\gamma = |C|$. The total number of permutations of the relevant tests is $(\pi + \gamma + 1)!$. While we can obtain $|\{\omega \in \Omega(T) \,|\, \text{fail}(\omega)\}|$ purely by definition, counting test orders where the victim fails, we prefer to take a probabilistic approach that will simplify further proofs. A victim fails if

- it is not in the first position, with probability $(\pi + \gamma)/(\pi + \gamma + 1)$, and
- its immediate predecessor is a polluter, with probability $\pi/(\pi + \gamma)$, giving the overall flake rate $f(T) = \pi/(\pi + \gamma + 1)$.

This formula is simple, but real tests often violate A1 or A2. Of the 249 tests used in our experiments, 13 violate both A1 and A2, 207 violate only A2, and 29 do not violate either.

Special Case 2: Keeping A1 but relaxing A2, assume that the victim is in class C_1 with π_1 polluters and γ_1 cleaners, and the other $k - 1$ classes have π_i polluters and γ_i cleaners, $2 \leq i \leq k$, where in general, either π_i or γ_i, but not both, can be zero for any class except for the victim's own class where both π_1 and γ_1 can be zero. Per Special Case 1, we have $f_A(T) = (\sum_{i=1}^{k} \pi_i)/(\sum_{i=1}^{k} \pi_i + \sum_{i=1}^{k} \gamma_i + 1)$. Next, consider class-compatible test orders, which do not interleave tests from different classes. The victim fails if (1) it fails in its own class, with probability $\pi_1/(\pi_1 + \gamma_1 + 1)$, or (2) the following three conditions hold: (2.1) the victim is the first in its own class, with probability $1/(\pi_1 + \gamma_1 + 1)$, (2.2) the class is *not* the first among classes, with probability $(k - 1)/k$, and (2.3) the immediately preceding class ends with a polluter, with probability $\pi_i/(\pi_i + \gamma_i)$ for each class i and thus the probability $\sum_{i=2}^{k}(\pi_i/(\pi_i + \gamma_i))/(k - 1)$ across all classes. Overall,

$$f_C(T) = \frac{\pi_1 + \frac{1}{k}\sum_{i=2}^{k} \frac{\pi_i}{\pi_i + \gamma_i}}{\pi_1 + \gamma_1 + 1}. \tag{3.3}$$

This formula is already more complex. Note that we can have either $f_A(T) \geq f_C(T)$ or $f_C(T) \geq f_A(T)$, based on the ratio of polluters and cleaners in the victim's own class vs. the ratio of polluters and victims in other classes, i.e., neither set of test orders ensures a higher flake rate. We show in Section 3.3.3 that both cases arise in practice.

General Case: In the most general case, relaxing A1 to allow different polluters to have a different set of cleaners, while also having all these relevant tests in different classes, it appears challenging to derive a closed-form expression for $f_A(T)$, let alone for $f_C(T)$. We thus resort to estimating flake rates by sampling orders from $\Omega_A(T)$ or $\Omega_C(T)$, and counting what ratio of them fail based on Definition 3.1 in Section 3.3.3.

3.3.3 Comparing Failure Rate for Different Sets of Test Orders

While tools such as iDFlakies [108] incorporate the requirement of not interleaving tests from different classes in a test order, some other tools [237] do not incorporate this requirement, so they allow all test orders. Recall that $\Omega_A(T)$ denotes the set of all test orders and $\Omega_C(T)$ denotes the set of test orders that satisfy the requirement. The reason to run $\Omega_A(T)$ is to try to maximize the detection of all potential OD tests at the risk that some detected failures would be false positives. In particular, a test failure observed in some non-class-compatible order may not be reproducible in any class-compatible prefix of that order, e.g., due to the various ways to customize JUnit [96] (with annotations such as `@Before`, `@BeforeClass`, `@Rule`) or similar testing frameworks. The reason to run only $\Omega_C(T)$ is to detect OD-test failures that developers can observe from running the tests and are therefore motivated to fix.

While both sets of test orders can detect all true positive OD tests, it is not clear which set of test orders are *more likely* to detect *true positive* OD tests. Intuitively, running $\Omega_A(T)$ test orders can more likely detect failures if cleaners and victims are in the same class, while polluters are in different classes; in such cases, polluters are less likely to come in between cleaners and the victim. For example, for the victim presented in Section 3.1, the $\Omega_A(T)$ flake rate is 10.5%, while the $\Omega_C(T)$ flake rate is 4.5%. On the other hand, running $\Omega_C(T)$ test orders can more likely detect failures if polluters and victims are in the same class, while cleaners are in different classes. Similar reasoning applies to brittles: if state-setters are more often in the same test class as the brittle, then the brittle is less likely to fail than if state-setters are more often in other classes.

To compare these sets of test orders on real OD tests, we use the 192 victim and 57 brittle tests described in Section 3.2.1. We collect all single test polluters for each victim and all single test cleaners for each polluter-victim pair. We also collect all single test state-setters for brittles. We then use either the formulas presented in Section 3.3.2 or many uniformly sampled test orders to obtain the flake rates, $f_A(T)$ and $f_C(T)$, for each test. Specifically, our formulas apply for 236 of the 249 tests. For the remaining 13 tests (all victims), we sample 100000 test orders from each of $\Omega_A(T)$ and $\Omega_C(T)$ to estimate their flake rates.

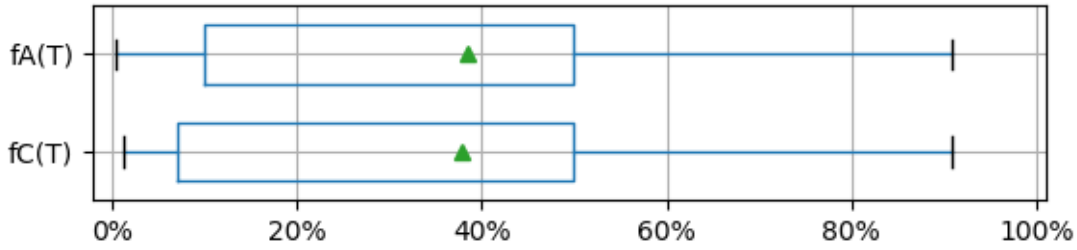


Figure 3.3: Distribution of flake rate for two sets of test orders.

Figure 3.3 summarizes the results. For each set of test orders, the figure shows a boxplot that visualizes the distribution of flake rates for 249 OD tests. The $f_A(T)$ flake rates have a slightly higher mean (38.4%) than the $f_C(T)$ flake rates (38.0%). Statistical tests for paired samples of the flake rates—specifically, dependent Student's t-test obtains a p-value of 0.47 and Wilcoxon signed-rank test obtains a p-value of 0.01—show that the differences could be statistically significant (at $\alpha = 0.05$ level). However, if we omit the 13 tests that required samplings, the means are 38.3% for $f_A(T)$ and 38.6% for $f_C(T)$, and the difference is not statistically significant (dependent Student's t-test obtains a p-value of 0.55, and Wilcoxon signed-rank test obtains a p-value of 0.19).

Prior work [59, 108, 111, 237] has not performed any explicit comparison between the two sets of test orders. Our results demonstrate that running $\Omega_A(T)$ might be more likely to detect true positive OD tests. However, using such test orders may contain false positives. Future work on detecting OD tests should explore how to address false positives if $\Omega_A(T)$ test orders are run.

3.3.4 Simple Change to Increase Probability of Detecting OD Tests

Inspired by our probability analysis, we propose a simple change to increase the probability of detecting OD tests. The standard algorithm for sampling S random test orders simply repeats S times the following steps:

1. $\omega \leftarrow$ sample a random test order from possible test orders ($\Omega_A(T)$ or $\Omega_C(T)$);
2. obtain result $r \leftarrow \text{run}(\omega)$;
3. if r is FAIL, then print ω.

(A variant [108] may store previously sampled test orders to avoid repetition, but the number of possible test orders is usually so large that sampling the same one is highly unlikely, so one can save space and time by not tracking previously sampled test orders.)

Our key change is to select the next test order as a *reverse* of the prior test order that passed: 4. if r is PASS, then $\omega_R \leftarrow \text{reverse}(\omega)$. The intuition for this change is that a passing order may have the polluter *after* the victim. Therefore, reversing the passing order would have the polluter *before* the victim, and thus the reverse of the passing order should have a higher probability to fail than a random order that may have the polluter before or after the victim. Note that the reverse of a class-compatible test order is also a class-compatible test order, so this change applies to $\Omega_C(T)$. The other changes are to run ω_R, print if it fails, and properly count the test orders to select exactly S samples of test orders.

We next compute the probability that the reverse of a passing order fails.

Special Case 1: Consider the Special Case 1 scenario from Section 3.3.2 with π polluters and γ cleaners. For the standard algorithm, $f(T) = f_A(T) = f_C(T) = \pi/(\pi+\gamma+1)$. For our change, the conditional probability that the second test order fails given that the first test order passes is $P(\text{fail}(\omega_R)|\text{pass}(\omega)) = P(\text{fail}(\omega_R) \wedge \text{pass}(\omega))/P(\text{pass}(\omega))$. We already have $P(\text{pass}(\omega)) = 1 - f(T) = (\gamma+1)/(\pi+\gamma+1)$.

To compute $P(\text{fail}(\omega_R) \wedge \text{pass}(\omega))$, we consider two cases based on the position of the victim in the passing test order ω.

1. If the victim is first, with the probability of $1/(\pi+\gamma+1)$, then the second test should be a polluter, with the probability of $\pi/(\pi+\gamma)$, so we get $\pi/((\pi+\gamma)(\pi+\gamma+1))$ for this case.

2. If the victim is not first, it cannot be the last in ω because otherwise, ω_R would not fail, so the victim is in the middle, with the probability of $(\pi+\gamma-1)/(\pi+\gamma+1)$. We also need a cleaner right before the victim, with probability $\gamma/(\pi+\gamma)$, and a polluter right after the victim, with probability $\pi/(\pi+\gamma-1)$. Overall, we get the probability $\pi\gamma/((\pi+\gamma)(\pi+\gamma+1))$ for this case.

We can sum up the two cases to get $P(\text{fail}(\omega_R) \wedge \text{pass}(\omega)) = \pi(\gamma+1)/((\pi+\gamma)(\pi+\gamma+1))$.

Finally, the conditional probability that the reverse test order fails given the first test order passes is $P(\text{fail}(\omega_R)|\text{pass}(\omega)) = (\frac{\pi(\gamma+1)}{(\pi+\gamma)(\pi+\gamma+1)})/(\frac{\gamma+1}{\pi+\gamma+1}) = \pi/(\pi+\gamma)$. This probability is strictly larger than $f(T) = \pi/(\pi+\gamma+1)$, because $\pi > 0$ must be true for the victim to be a victim.

Special Case 2: For the Special Case 2 scenario from Section 3.3.2, the common case is $\pi_1 + \gamma_1 > 0$ (i.e., the victim's class C_1 has at least one other relevant test). Based on the relative position of the victim in class C_1, we consider three cases: the victim runs

first, in the middle, or last in class C_1. After calculating the probability for the three cases separately and summing them up, we get the probability that the reverse test order fails and the first test order passes as $P(\text{fail}(\omega_R) \wedge \text{pass}(\omega)) = \frac{\pi_1+k\pi_1\gamma_1+\pi_1 S_\gamma+\gamma_1(\pi_1+\gamma_1+1)S_\pi}{k(\pi_1+\gamma_1)(\pi_1+\gamma_1+1)}$ where $S_\pi = \sum_{i=2}^{k} \frac{\pi_i}{\pi_i+\gamma_i}$ and $S_\gamma = \sum_{i=2}^{k} \frac{\gamma_i}{\pi_i+\gamma_i}$. In Section 3.3.2, we have computed $P(\text{pass}(\omega))$, so dividing $P(\text{fail}(\omega_R) \wedge \text{pass}(\omega))$ by $P(\text{pass}(\omega))$ gives the conditional probability that the reverse test order fails given the first test order passes. Due to the complexity of the formulas, it is difficult to show a detailed proof that $P(\text{fail}(\omega_R)|\text{pass}(\omega)) > f(T)$, so we sample test orders instead.

When we sample both $\Omega_A(T)$ and $\Omega_C(T)$ for 100000 random test orders on all 249 OD tests without reverse (i.e., the standard algorithm) and with reverse when a test order passes (i.e., our change), we find that our change does statistically significantly increase the chance to detect OD tests. Specifically, for $\Omega_A(T)$, test orders without reverse obtain a mean of 38.6%, while test orders with reverse of passing test orders obtain a mean of 45.3%. Statistical tests for paired samples on the flake rates without and with reverse for $\Omega_A(T)$ show a p-value of $\sim 10^{-38}$ for dependent Student's t-test and a p-value of $\sim 10^{-43}$ for Wilcoxon signed-rank test. Similarly, for $\Omega_C(T)$, test orders without reverse obtain a mean of 38.0%, while test orders with reverse of passing test orders obtain a mean of 45.3%. Statistical tests for paired samples on the flake rates without and with reverse for $\Omega_C(T)$ show a p-value of $\sim 10^{-42}$ for dependent Student's t-test and a p-value of $\sim 10^{-42}$ for Wilcoxon signed-rank test.

Based on these positive results, we have changed the iDFlakies tool [108] so that, by default, it runs the reverse of the previous order, instead of running a random order, if the previous order found no new flaky test.

3.4 GENERATING TEST ORDERS TO COVER TEST PAIRS

We next discuss our algorithm to generate test orders that systematically cover all test pairs for a given set T with n tests. The motivation is that even with our change to increase the probability to detect OD tests, the randomization-based sampling remains inherently probabilistic and can fail to detect an OD test.

3.4.1 Special Case: All Orders are Class-Compatible

We first focus on the special case where we have only one class, or many classes that each have only one test, so all $n!$ permutations are class-compatible. For example, for $n = 2$ we can cover both pairs with $\Omega_2 = \{\langle t_1, t_2\rangle, \langle t_2, t_1\rangle\}$, and for $n = 4$ we can cover all 12 pairs with 4 test orders $\Omega_4 = \{\langle t_1, t_4, t_2, t_3\rangle, \langle t_2, t_1, t_3, t_4\rangle, \langle t_3, t_2, t_4, t_1\rangle, \langle t_4, t_3, t_1, t_2\rangle\}$. Recall that

n is the minimum number of test orders needed to cover all test pairs, so the cases for $n = 2$ and $n = 4$ are optimal. The reader is invited to consider for $n = 3$ whether we can cover all 6 test pairs with just 3 test orders. The answer is upcoming in this section.

To address this problem, we consider *Tuscan squares* [68], objects studied in the field of combinatorics. Given a natural number n, a Tuscan square consists of n rows, each of which is a permutation of the numbers $\{1, 2, \ldots, n\}$, and every pair $\langle i, j \rangle$ of distinct numbers occurs consecutively in some row. Tuscan squares are sometimes called "row-complete Latin squares" [154], but note that Tuscan squares need *not* have each column be a permutation of all numbers.

A Tuscan square of size n is equivalent to a decomposition of the complete graph on n vertices, K_n, into n Hamiltonian paths [207]. The decomposition for even n has been known since the 19$^{\text{th}}$ century and is often attributed to Walecki [124]. The decomposition for odd $n \geq 7$ was published in 1980 by Tillson [207]. Tillson presented a beautiful construction for $n = 4m + 3$ and a rather involved construction for $n = 4m + 1$ with a recursive step and manually constructed base case for $n = 9$. In brief, Tuscan squares can be constructed for all values of n except $n = 3$ or $n = 5$. We did not find a public implementation for generating Tuscan squares, and considering the complexity of the case $n = 4m + 1$ in Tillson's construction, we have made our implementation public [8].

We can directly translate permutations from Tuscan squares into n test orders that cover all test pairs in this special case (where all test pairs are either only intra-class test pairs of one class or only inter-class test pairs of n classes). These sets of test orders have the minimal possible cost: $\text{Cost}(\Omega_n) = n(\text{Cost}_0 + \text{Cost}(T))$, substantially lower than $\text{Cost}(\Omega_p)$ for running all test pairs in isolation. For $n = 3$ and $n = 5$, we have to use 4 and 6 test orders, respectively, to cover all test pairs. For example, for $n = 3$ we can cover all 6 pairs with 4 orders $\{\langle t_1, t_2, t_3 \rangle, \langle t_2, t_1, t_3 \rangle, \langle t_3, t_1 \rangle, \langle t_3, t_2 \rangle\}$.

3.4.2 General Case

Algorithm 3.1 shows the pseudo-code algorithm to generate test orders that cover all test pairs in the general case where we have more than one class and at least one class has more than one test. The main function calls two functions to generate test orders that cover intra-class and inter-class test pairs.

The function `cover_intra_class_pairs` generates test orders that cover all intra-class test pairs. For each class, the function `compute_tuscan_square` is used to generate test orders of tests within the class to cover all intra-class test pairs. These test orders for each class are then appended to form a test order for the entire test suite T. The function `pick`, invoked on

Algorithm 3.1: Generate test orders that cover all intra-test-class and inter-test-class test-method pairs

```
Input: T                    # test suite, a set of test methods partitioned into test classes
Output: Ω                   # output is a set of test orders
Function cover_all_pairs():
    Ω = {}                  # empty set
    cover_intra_class_pairs()
    cover_inter_class_pairs()
Function cover_intra_class_pairs():
    map = {}                # map each class to all its intra-class orders
    for C ∈ classes(T) do
        map = map ∪ {⟨C, ω_C⟩ | ω_C ∈ compute_tuscan_square(C)}
    end
    while map ≠ {} do
        ω = ⟨⟩              # empty order
        Cs = {C | ∃ω_C. ⟨C, ω_C⟩ ∈ map}
        for C ∈ Cs do
            ω_C = pick({ω_C | ⟨C, ω_C⟩ ∈ map})
            map = map \ {⟨C, ω_C⟩}
            ω = ω ⊕ ω_C     # append order
        end
        Ω = Ω ∪ {ω}
    end
Function cover_inter_class_pairs():
    pairs = {⟨t, t'⟩ | t, t' ∈ T ∧ class(t) ≠ class(t')}\   # from all inter-class pairs..
      {⟨t, t'⟩ | ∃ω ∈ Ω. cover(ω, ⟨t, t'⟩)}                # ..remove covered by intra-class orders
    while pairs ≠ {} do
        ω = pick(pairs)     # start with a randomly chosen not-covered pair
        pairs = pairs \ {ω}
        while true do
            t_p = ω_{|ω|−1}                                 # previously last test
            ts = {t | ⟨t_p, t⟩ ∈ pairs ∧ class(t) ∉ classes(ω)}
            if ts = {} then
                break
            t_n = pick(ts)                                  # next test to extend order
            pairs = pairs\{⟨t_p, t_n⟩}
            ω = ω ⊕ t_n
        end
        Ω = Ω ∪ {ω}
    end
```

multiple lines, chooses a random element from a set. The outer loop iterates as many times as the maximum number of intra-class test orders for any class. When the loop finishes, Ω contains a set of test orders that cover *all intra-class* and *some inter-class* test pairs. Each test order that concatenates tests from l classes covers $l-1$ inter-class test pairs. (Using just these test orders, we already detected 44 new OD tests in the test suites from the iDFlakies

dataset.) Each intra-class test pair is covered by exactly one test order. Modulo the special cases for $n_i = 3$ and $n_i = 5$, each *covered* inter-class pair appears in *exactly one* test order in Ω, because Tuscan squares satisfy the invariant that each element appears only once as the first and once as the last in the permutations in a Tuscan square.

The function `cover_inter_class_pairs` generates more test orders to cover the remaining inter-class test pairs. It uses a greedy algorithm to first initialize a test order with a randomly selected not-covered test pair and then extend the test order with a randomly selected not-covered test pair as long as an appropriate test pair exists. Extending the test order as long as possible reduces both the number of test orders and the number of times each test needs to be run.

We evaluate our randomized algorithm on 121 modules from the iDFlakies dataset as described in Section 3.2.1. We use the total cost, which considers the number of test orders and the number of tests in all of those test orders. The number of test orders is related to Cost_0, while the number of tests is related to Cost_1 as defined in Section 3.2. We run our algorithm 10 times for various random seeds. The coefficient of variation [26] for each module shows that the algorithm is fairly stable, with the average for all modules being only 1.1% and 0.25% for the number of test orders and the number of tests, respectively.

Compared with Ω_p that has all test orders of just test pairs, our randomized algorithm's average number of test orders and the average number of tests are only 3.68% and 51.8%, respectively, that of all the Ω_p test orders. The overall cost of the test orders generated by our randomized algorithm is close to the optimal, because the number of test orders is reduced by almost two orders of magnitude, and 51.8% of the number of tests is close to the theoretical minimum of 50% that of Ω_p test orders for Cost_1.

3.5 SUMMARY

Order-dependent (OD) tests are one prominent category of flaky tests. Prior work [108, 111, 237] has used randomized test orders to detect OD tests. In this chapter, we have presented the first analysis of the probability that randomized test orders detect OD tests. We have also proposed a simple change for sampling random test orders to increase the probability of detecting OD tests. We have finally proposed a novel algorithm that systematically explores all consecutive pairs of tests, guaranteeing to find all OD tests that depend on one other test. Our experimental results show that our algorithm runs substantially fewer tests than a naive exploration that runs all pairs of tests. Our runs of some test orders generated by the algorithm detect 44 new OD tests, not detected in prior work [108, 111, 112] on the same evaluation dataset.

CHAPTER 4: [DETECTING] A LARGE-SCALE LONGITUDINAL STUDY OF FLAKY TESTS

This chapter presents our in-depth study on when flaky tests are introduced, what changes cause tests to be flaky, and how developers should utilize their efforts to detect flaky tests. Some software organizations, e.g., Mozilla [201] and Netflix [149], run some tools—which we call flaky-test *detectors*—to detect flaky tests as soon as possible. However, detecting flaky tests is costly due to their inherent non-determinism, so even state-of-the-art detectors are often impractical to be used on all tests for each project change.

To help researchers and developers decide when they should use detectors, this chapter answers the following research questions:

RQ1: How effective are flaky-test detectors if run only when tests are introduced?

RQ2: How do flaky-test categories affect the effectiveness of running flaky-test detectors only when tests are introduced?

RQ3: When should one run flaky-test detectors?

Our study finds that ∼75% of flaky tests (184 out of 245) are flaky when added, indicating substantial potential value for developers to run detectors specifically on newly added tests. However, running detectors solely on newly added tests would still miss detecting ∼25% of flaky tests. The percentage of flaky tests that can be detected does increase to ∼85% when detectors are run on newly added or directly modified tests. The remaining 15% of flaky tests become flaky due to other changes and can be detected only when detectors are applied on more than just newly added or directly modified tests. Our study is the first to empirically evaluate when tests become flaky and to recommend guidelines for applying detectors in the future.

The remainder of this chapter is organized as follows. Section 4.1 presents our study setup, and Section 4.2 presents our study methodology. Section 4.3 then presents our study results, and Section 4.4 presents our case studies of tests that are not flaky when they are introduced. Finally, Section 4.5 presents the threats to validity, and Section 4.6 concludes this chapter.

4.1 STUDY SETUP

Flaky tests should ideally be detected as soon as the change that introduces the flakiness is made. Developers at Mozilla [201] and Netflix [149] are already trying to detect flaky tests

right when tests are added or modified by repeatedly running such tests in isolation multiple times. Aside from rerunning tests in isolation, various other approaches have been proposed to detect flaky tests [59, 73, 84, 108, 184, 237]. We next describe the detection approaches that we use to obtain a dataset of flaky tests (Section 4.1.1), the flaky-test dataset that we use for our study (Section 4.1.2), and how we categorize flaky tests to reproduce their failures (Section 4.1.3).

4.1.1 Flaky-Test Detection Approaches

As flaky tests become a growing problem for developers, two flaky-test detection approaches have been proposed for some common categories:

1. Prior work [59, 73, 84, 108, 237] has proposed various ways to detect flaky tests whose test result depends on the order in which the tests are run; such tests are known as *order-dependent* (OD) flaky tests. Examples of OD tests are presented in Section 1.2.1 and Section 3.1.

2. Prior work [184, 201] has also proposed various ways to detect flaky tests whose test result depends on the implementation of a non-deterministic specification; we refer to such tests as *implementation-dependent* (ID) flaky tests.

For our study, we select one tool for each flaky-test detection approach, specifically iDFlakies [108] for detecting OD tests and NonDex [184] for detecting ID tests. Our selection of detectors is based on our goal to evaluate both detectors on a common set of projects. Therefore, we select detectors that share the same programming language and build infrastructure. For detecting OD tests, most of the detectors that we find are for Java projects [59, 73, 84, 108, 237] and some of the detectors are for Maven-based projects. We select iDFlakies [108], because it is co-authored by us (presented in Chapter 2) and we are thus familiar with it. For detecting ID tests, we select NonDex [184] because it is the only detector for Maven-based, Java projects.

iDFlakies. As described in Chapter 2, iDFlakies [108] is a testing tool developed for detecting flaky tests in Maven-based, Java projects. To detect flaky tests, iDFlakies repeatedly runs projects' test suites, by permuting the order of tests in a test suite, and compares test results across repeated runs. If a test has both pass and fail results for at least two runs, the detector flags the test as flaky. To increase the likelihood of detecting flaky tests, iDFlakies provides different ways to change the order in which tests are run. iDFlakies can detect three main categories of flaky tests:

- *Order-dependent brittle* (OD Brit) – tests that fail when run in isolation but pass when run after some specific tests;
- *Order-dependent victim* (OD Vic) – tests that pass when run in isolation but fail when run after some specific tests;
- *Non-deterministic* (NOD) – tests that non-deterministically pass or fail with no changes to test execution order or implementation of test dependencies.

NonDex. NonDex [184] is a tool for detecting incorrect assumptions by developers on specifications. Some APIs have underdetermined specifications [120], i.e., the specifications allow multiple implementations to return different results for the same input, even if each implementation is itself deterministic and always returns the same result for the same input. Such specifications allow implementations to be changed later to achieve various goals, e.g., to optimize performance or reliability. When developers use such APIs with assumptions that are not documented in the specifications, the developers may inadvertently create flaky tests. For example, imagine a test that uses a `HashSet` and makes an assumption regarding the order in which elements of the `HashSet` are iterated. While one Java version could provide a deterministic iteration order, there is no guarantee that another Java version provides the same order, e.g., the iteration order of `HashSet` in Java 7 may differ from that in Java 8. Since `HashSet`'s specification never specifies the iteration order of elements, a test may fail in Java 8 but pass in Java 7 if the implementation of `HashSet` between the two versions changes the iteration order.

NonDex is a detector for tests that make such assumptions. Specifically, NonDex detects a flaky test by exploring different allowed behaviors of under-specified APIs while the test is running. For example, in the case of `HashSet`, NonDex explores whether different iteration orders of the `HashSet` may cause the test to fail. By exploring different allowed behaviors of underdetermined APIs through repeated runs of a test, NonDex can detect two main categories of flaky tests:

- *Implementation-dependent* (ID) – tests that are dependent on a specific implementation of an API whose specification admits other implementations;
- *Non-deterministic* (NOD) – tests that non-deterministically pass or fail with no changes to test execution order or implementation of test dependencies.

4.1.2 Flaky Tests Used in Our Study

As described in Section 4.1.1, both detectors that we select to detect flaky tests work on Maven-based, Java projects. One detector that we use in this study is our iDFlakies tool, which also comes with a dataset of flaky tests that we detected with the detector in our prior work [108]. For this study, we use the same set of projects on the same commit[1] as the iDFlakies dataset. We refer to this commit for each project as the project's *iDFlakies-commit*. Specifically, the dataset has 683 projects that are all selected from GitHub [63] based on their popularity among Java projects. Each Maven-based project is organized into one or more *modules*, which are the basic units that Maven-based tools, including iDFlakies and NonDex, run.

We use iDFlakies version 1.0.2, which can be configured to permute tests in a module's test suite in several ways. Following the recommendation from our prior work [108] (described in Chapter 2), we first run the ReverseC+M configuration once (which runs all test methods in the reverse order of the default Maven order) and then proceed to run the RandomC+M configuration 100 times (which runs all test methods in a random order each time)[2]. The iDFlakies dataset consists of a *Comprehensive* dataset (projects on which all configurations of iDFlakies were evaluated) and an *Extended* dataset (a larger set of projects on which only the RandomC+M configuration was evaluated). For this study, we select all of the modules of all projects in the Comprehensive dataset except for seven projects that take more than three days to run—leaving us with 21 projects. We also run iDFlakies on each module where a flaky test is detected in the iDFlakies dataset, regardless of whether the module belongs to a project from the Comprehensive or Extended set of projects. We do not directly use the flaky tests from the iDFlakies dataset, because the test suites of some modules where iDFlakies detected flaky tests for the dataset were run up to 16503 times, enabling the dataset to contain flaky tests that require many runs to be detected. To limit the machine cost of our experiments, we want flaky tests that are likely to be detected in 100 runs, which, as described in Section 4.2.2, is the number of runs that we use for each applicable categorization step to reproduce flaky-test failures.

For NonDex, we set it to run for 10 times for each module's test suite. We do not provide a random seed to NonDex, which means that NonDex generates its own random seed for each of the 10 runs. We also set NonDex to the ONE level, where NonDex changes the order of any object only when it is first accessed, and this order is not changed by NonDex for the

[1]Only for the `alibaba/fastjson` project, the commit that we use (SHA 5c6d6fd4) differs because the original commit (SHA 57d0434) used in the iDFlakies dataset is no longer available [53].

[2]Note that the simple change described in Section 3.3.4 to increase the probability of detecting OD tests came after our study and is not in the iDFlakies version that we used for this study.

rest of the run. We use the ONE level, because compared to other levels of NonDex, the ONE level modifies the execution of tests the least, and failures found from this level are therefore more likely to be real flaky tests than the ones found by the other levels. With these settings for NonDex, we use version 1.1.2 and run the tool on all modules of all projects from the iDFlakies dataset.

Running iDFlakies and NonDex as described detects a total of 306 flaky tests from 55 projects. Table 4.1 shows the overall characteristics of these 55 projects, and Table 4.2 shows the number of flaky tests detected per project. The projects have a wide range of sizes (from only 387 to 958112 lines of Non-Test code[3]), and their test suites can be quite big (up to 8471 tests[4]).

Using Kendall's rank correlation coefficient, we investigate the relationship between the project characteristics (Table 4.1) and the number of flaky tests detected (Table 4.2). Because flaky tests are a subset of all tests, the cardinalities of the two sets have an obvious relationship: the former can never exceed the latter. However, it is not obvious to which degree the number of flaky tests is correlated with the total number of tests.

We find moderate positive correlations between the number of flaky tests in a project and all three project characteristics in Table 4.1; lines of Non-Test code ($\tau = 0.322$, $p = 7.8 \times 10^{-4}$), lines of Test code ($\tau = 0.408$, $p = 2.1 \times 10^{-5}$), and number of tests ($\tau = 0.365$, $p = 1.4 \times 10^{-4}$). We also investigate the correlations for individual categories of flaky tests. We find that the correlations are only significant (at $\alpha = 0.05$) for the number of ID flaky tests and lines of Non-Test code ($\tau = 0.261$, $p = 0.04$), lines of Test code ($\tau = 0.332$, $p = 0.01$), and number of tests ($\tau = 0.316$, $p = 0.02$).

Because the number of flaky tests is positively correlated with the project characteristics in Table 4.1, and these characteristics differ widely for the projects in our study, we also investigate whether the number of flaky tests increases *proportionally* with the project characteristics. We conduct a test for equality of proportions and find that flaky tests and project characteristics do not grow proportionally ($p < 2.2 \times 10^{-16}$). A visual investigation of the corresponding scatter plots in Figure 4.1 confirms that the number of flaky tests grows at a lower rate than project characteristics, but it still grows. To combat the growing number of flaky tests, we conclude that developers should strive to detect and fix flaky tests when they are introduced.

[3]Lines of code (LOC) have been counted using cloc (`https://github.com/AlDanial/cloc`). "Test" LOC have been counted as Java LOC in files whose names include the word "Test". "Non-Test" LOC have been counted as all Java LOC in a project, except for test LOC as specified before.

[4]Tests have been counted as test methods run according to Maven Surefire reports.

Table 4.1: Characteristics of the projects used in our study.

Project Slug from GitHub	iDFlakies SHA	Lines of Code Non-Test	Lines of Code Test	Test Count
activiti/activiti	b11f757a	93,827	53,958	2,045
alibaba/fastjson	5c6d6fd4	105,454	64,700	4,781
apache/hadoop	cc2babc1	958,112	688,013	5,358
apache/hbase	801fc05e	422,013	267,346	2,844
apache/incubator-dubbo	737f7a7e	69,473	32,681	2,156
apache/struts	13d90530	112,028	45,774	4,003
apereo/java-cas-client	574b74fa	7,547	2,862	160
c2mon/c2mon	d80687b1	55,931	21,494	284
codingchili/excelastic	6bb7884b	1,228	261	12
ctco/cukes	b483e1a8	7,631	1,154	54
davidmoten/rxjava2-extras	d0315b6e	7,467	5,324	389
doanduyhai/achilles	e3099bdc	30,289	17,827	613
dropwizard/dropwizard	07dfaed6	23,926	27,790	1,559
eclipse-ee4j/tyrus	d86e0cb0	30,615	25,944	554
elasticjob/elastic-job-lite	b022898e	8,277	7,947	562
espertechinc/esper	590fa9c9	461,075	30,656	1,604
feroult/yawp	b3bcf9c9	16,751	5,631	362
fhoeben/hsac-fitnesse-fixtures	a64c18d9	11,801	7,076	341
flaxsearch/luwak	c27ec08c	4,609	3,350	205
fluent/fluent-logger-java	da14ec34	739	899	18
fromage/redpipe	0aff891d	5,796	1,983	64
google/jimfs	ced6093f	7,882	9,484	454
hexagonframework/spring-data-ebean	dd11b976	2,984	508	48
javadelight/delight-nashorn-sandbox	da35edc0	1,029	1,459	79
jfree/jfreechart	520a4be6	94,000	39,847	2,176
jhipster/jhipster-registry	00db3661	2,580	965	53
kagkarlsson/db-scheduler	4a8a28e6	2,433	1,417	48
kevinsawicki/http-request	2d62a3e9	1,391	2,721	163
ktuukkan/marine-api	af000384	8,598	7,274	926
logzio/sawmill	e493c2e2	5,923	4,595	271
looly/hutool	91565d05	54,678	7,704	611
nationalsecurityagency/timely	3a8cbd33	18,290	5,422	166
openpojo/openpojo	9badbcc4	12,019	10,982	1,185
orbit/orbit	c4904af2	15,907	6,532	192
oryxproject/oryx	72ae4bb3	14,428	5,743	393
pholser/junit-quickcheck	9361b6da	8,338	16,893	1,081
pippo-java/pippo	ae898b6c	16,348	3,488	211
querydsl/querydsl	2bf234ca	72,487	37,317	8,471
ripe-ncc/whois	79e90f41	55,115	66,283	2,563
sonatype-nexus-community/nexus-repository-helm	60a9e8de	1,562	682	18
spinn3r/noxy	d53a4942	2,794	1,286	35
spotify/helios	aebf68dc	28,039	18,434	569
spring-projects/spring-boot	daa3d457	126,310	133,285	7,337
spring-projects/spring-data-envers	5637994b	387	251	10
spring-projects/spring-ws	e8d89c9e	29,727	19,252	1,367
tbsalling/aismessages	7b0c4c70	5,118	1,021	44
tools4j/unix4j	367da7d2	10,779	6,401	454
tootallnate/java-websocket	fa3909c3	7,679	3,239	146
undertow-io/undertow	d0efffad	107,888	29,315	852
vmware/admiral	e4b02936	130,480	61,289	1,383
wikidata/wikidata-toolkit	20de6f7f	20,156	11,089	787
wildfly/wildfly	b19048b7	366,899	148,605	1,102
wro4j/wro4j	185ab607	20,187	20,098	1,288
wso2/carbon-apimgt	a82213e4	75,958	26,459	1,292
zalando/riptide	8277e11f	5,526	7,147	337
Total	—	**3,768,508**	**2,029,157**	**64,080**

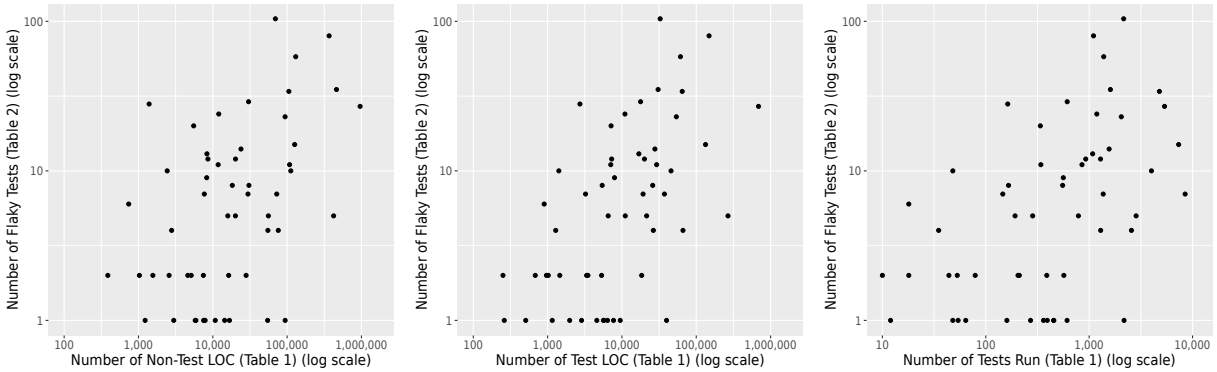


Figure 4.1: Relation of project characteristics and flaky test counts for projects with one or more flaky tests.

4.1.3 Flaky-Test Categorization Approaches

To study whether flaky tests are flaky when introduced (RQ1), and thus whether flaky-test detectors should run only when tests are introduced, we find the commit where each flaky test that we detect on the iDFlakies-commit first becomes flaky. As iDFlakies randomly permutes test orders, the flaky-test failures that it finds on the iDFlakies-commit in a particular test order may be expensive to find on a different commit due to the number of possible orders for the tests ($n!$, where n is the number of tests in the test suite of a particular commit). While we can sample a large number of orders on a *single* commit (the iDFlakies-commit), it is not scalable to use this approach across potentially many commits to identify the commit in which a flaky test first becomes flaky—which we call *Flakiness-introducing commit (FIC)*—for each test. Instead of running iDFlakies on each revision, we rely on three targeted approaches to allow us to faster check whether flaky tests are flaky (and when they first become flaky) and to study whether the point of introducing flakiness differs across different categories of flaky tests.

Specifically, we run isolation (running each test by itself) to detect OD Brit tests, NonDex (as described in Section 4.1.1) to detect ID tests, and One-by-one (running each test after every other test) to detect OD Vic tests. Although NonDex and iDFlakies are designed to detect specific categories of flaky tests (ID and OD, respectively), they can also both detect NOD tests. As described in Section 4.2.2, we use these same categories of flaky tests to study whether some kinds of flaky tests are easier to detect when tests are introduced than other kinds (RQ2).

Isolation. Prior work [15, 16, 142, 201] has proposed isolating tests from one another during their runs. Running tests in isolation is a common way for developers to detect flaky

tests. Its popularity largely stems from the fact that tests of the OD Brit category deterministically fail when run in isolation (and moreover, if one aims to *avoid* rather than detect flaky tests, then running tests in isolation can avoid failures from the OD Vic category). Furthermore, because each test is run in isolation (rather than in various permutations with other tests), the cost of this approach is relatively low. This approach also helps detect NOD tests, because the tests can be run more times in less time compared to other tools, such as OD test detectors, that typically run many permutations of the test suite.

This approach for detecting flaky tests also resembles the strategy used by developers at Mozilla [201] and Netflix [149]. By including this approach, we benefit from its complementary detection abilities compared to the other tools and its representativeness for pragmatic approaches commonly used by developers. One caveat for running tests in isolation is that the number of runs needed to detect any particular flaky test is unclear, and typically developers arbitrarily choose the number of times to run (e.g., for Mozilla, tests are run 10 times with no particular justification for the number 10). For our runs, we choose a much higher number of times to run (100) than what Mozilla developers use by default [201]. For running a specific test in isolation, we simply use an already existing feature in Maven Surefire [132], namely `mvn test -Dtest=TestClass#testMethod`.

One-By-One. While iDFlakies can effectively detect flaky tests, iDFlakies has one major limitation for our study. Namely, depending on the test suite size, randomized runs may (a) take substantial time, and (b) cover only a small fraction of the relevant permutations. Recall that one of our research questions is whether flaky tests are flaky when the tests are first introduced to the test suite (RQ1). Therefore, not covering all permutations is particularly problematic to our study because we may consider tests to not be flaky when introduced due to the necessary permutations not being covered (and not solely due to tests not being flaky when they were introduced). Specifically, when we do not observe a failure of an OD Vic test at a particular commit, the reason may simply be that the random orders tried by iDFlakies do not include any order that exposes the OD Vic test, and not because the test is not flaky at that particular commit. To address this limitation of iDFlakies, we run every potentially flaky test after every other test from the test suite. That is, we pair every potentially flaky test t with every other test, t', and run all pairs $\langle t', t \rangle$. We call this way of running tests as running them *one-by-one (OBO)*. We rely on OBO as a potentially more expensive, yet also more deterministic way than iDFlakies to confirm whether tests are OD at a particular commit. One caveat of using OBO compared to iDFlakies is that OBO may miss OD Vic tests that require multiple tests to run before t to fail (e.g., for t to fail, it must run $\langle t'', t', t \rangle$, while both $\langle t', t \rangle$ and $\langle t'', t \rangle$ pass). However, according to our prior

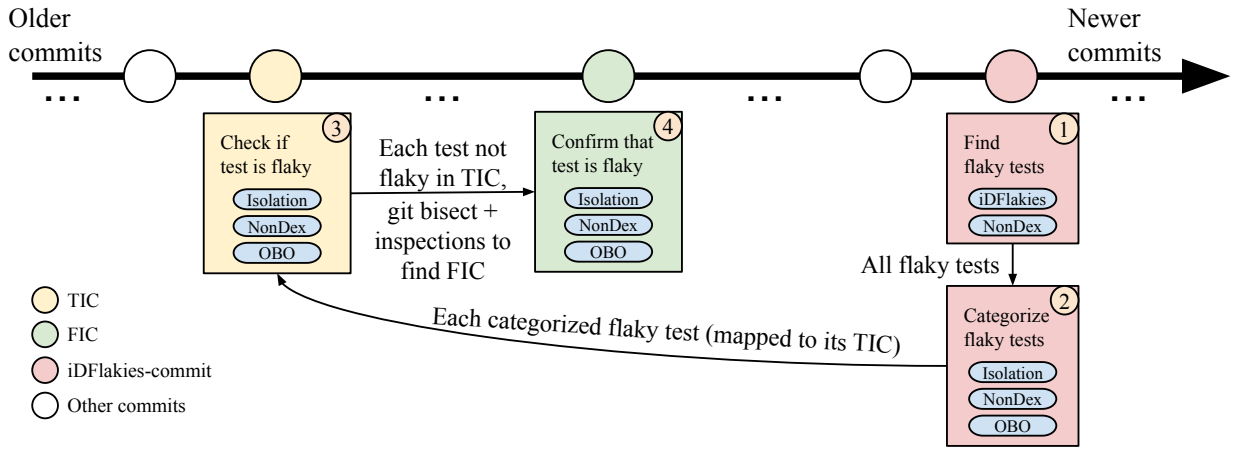


Figure 4.2: High-level overview of our study methodology. For Step 1, we use two different tools to detect flaky tests. For Step 2, we categorize each of these flaky tests. For Step 3, we identify the Test-introducing commit (TIC) and confirm whether the test was flaky in that commit. For Step 4, we identify the Flakiness-introducing commit (FIC) for each test that is not flaky in its TIC.

work [186], it is rarely the case that multiple tests are required for an OD Vic test to fail, and it is suggested that future work focuses on just individual tests to find OD Vic tests.

As far as we are aware, there is no existing tool that would support our OBO approach by running a specific test after every other test[5]. By default, Maven Surefire [132] allows one to specify which specific tests to run, but it does not enable one to specify the order of the tests. Therefore, to run tests in OBO, we build on top of Maven Surefire version 3.0.0-M6 to enable it to customize the order in which tests are run. Using our custom plugin, we first obtain the full list of tests for each test suite and then for any particular flaky test, we invoke our Surefire plugin with every other test coming before that particular test, resulting in an OBO test execution.

4.2 METHODOLOGY

This section describes how we use the flaky-test dataset described in Section 4.1.2 and the flaky-test categorization approaches described in Section 4.1.3 to answer our RQs.

4.2.1 RQ1: How Effective Are Detectors if Run Only When Tests Are Introduced?

The main goal of our study is to understand when one should run flaky-test detectors

[5]Zhang et al. [237] did suggest to detect order-dependent tests by running them in pairwise where every test is run with every other test. However, their tool does not allow one to easily specify a specific test to run with every other test.

such as iDFlakies [108] and NonDex [184]. Specifically, to reduce the cost of running these tools, should one run them only when new tests are added, and how would it impact the effectiveness of these tools, namely the number of flaky tests detected? To study this question, for each flaky test detected by iDFlakies or NonDex as described in Section 4.1.2, we apply the targeted approaches described in Section 4.1.3 to detect whether the test was flaky in its *Test-introducing commit (TIC)*. If the test is not flaky in its TIC, we proceed to run the targeted approaches on subsequent commits to obtain the *Flakiness-introducing commit (FIC)* that introduced the flakiness. If a test's TIC is the same as its FIC, it simply implies that the test is already detectable as flaky when it is introduced (TIC). Figure 4.2 shows a high-level overview of the process we used to find the TIC and FIC for all flaky tests detected by iDFlakies or NonDex.

Obtaining TICs. For each flaky test (uniquely identified by its fully qualified Java class and method name) that iDFlakies or NonDex detect on the iDFlakies-commit, we traverse the project's history to find the TIC. While finding the TIC is seemingly simple, there are two key challenges:

- the location of a test could have changed, e.g., to a different module or package, and
- test method names can occur in multiple classes.

To address both challenges, we first search by the test method name and then filter the results. Specifically, we use `git pickaxe` to search for all commits that add a line containing the test method name. Doing so helps us identify commits in which the test could be added, even if the test's location (i.e., file path) in TIC differs from the test's location in iDFlakies-commit. However, the check for the test method name is insufficient to differentiate common method names (e.g., `testList`) from different classes. To address this issue, after checking the method name, we check the name of the file that contains an added line with the method name. We continue traversing the commits from the oldest to the newest until we find a match for both the method and file name (i.e., the simple class name) of the flaky test, or we run out of commits without finding an appropriate TIC. Running this procedure automatically finds the TIC for the majority of our tests. This procedure can fail to find the TIC for some tests due to reasons such as a test method defined in a superclass but executed by a subclass, or a test class renamed after the test method name was added. In such cases, we manually find the TICs to ensure a complete dataset.

Obtaining FICs. Once we have found the TIC, we check to see if the test is flaky on that commit by using our Isolation, NonDex, and One-by-one targeted approaches. If we

find the test to be flaky on the TIC, then we have also found the FIC— it matches the TIC. For each flaky test that we are unable to detect as flaky on the TIC, we proceed to find the commit that introduces the flakiness (FIC). In theory, we could simply use `git bisect` [62] to binary search for the intermediate commit between TIC and iDFlakies-commit that introduces the flakiness. Specifically, at each commit of the binary search, `git bisect` would run the targeted approach that categorized the flaky test at the iDFlakies-commit. If the flakiness can be detected in a particular commit, then the next commit to search for would be closer to the TIC. If the test cannot be detected as flaky in a particular commit, then the next commit to search for would be closer to the iDFlakies-commit. Finally, `git bisect` outputs a commit as the FIC once it observes two consecutive commits where the first commit cannot detect the test to be flaky, but the second commit can.

However, the binary search that `git bisect` employs often gets misled and outputs a wrong commit for a variety of reasons: the project may not compile at some intermediate commit between TIC and iDFlakies-commit (although it does compile on both of those commits)[6]; the test may not exist in some intermediate commit (again, although it does exist in both the starting and ending commits); or the test may have a different category of flakiness, e.g., switching from NOD to OD, which increases the number and type of test runs that we need to perform to get a reliable signal about the flakiness status of the test. As a result, we use a semi-automated approach, by first running `git bisect` to get a suggested intermediate commit, then (manually) double checking if such commit is indeed the commit that introduced flakiness, and if not, we manually continue the search for the FIC. We eventually manually confirm all of the FICs that we find by confirming that the test is flaky in that FIC but not flaky on any parent of the FIC (and for this confirmation, we use many more runs than usual to increase the probability to properly categorize the test).

4.2.2 RQ2: How Do Flaky-Test Categories Affect the Effectiveness of Running Flaky-Test Detectors Only When Tests Are Introduced?

Because different flaky-test detectors are better suited to detect different categories of flaky tests, it is important to understand the high-level reason why each of the flaky tests that we studied is flaky. The goal of RQ2 is to extend the question in RQ1 to better understand how different flaky-test categories may affect when one should run flaky-test detectors. To study this question, we categorize each flaky test found by iDFlakies or NonDex based on the procedure shown in Figure 4.3. We perform this categorization on the iDFlakies-commit

[6]We have adjusted our `git bisect` script to use the exit code 125 as recommended for code that does not compile, but even that did not help in many cases to identify the appropriate intermediate commit.

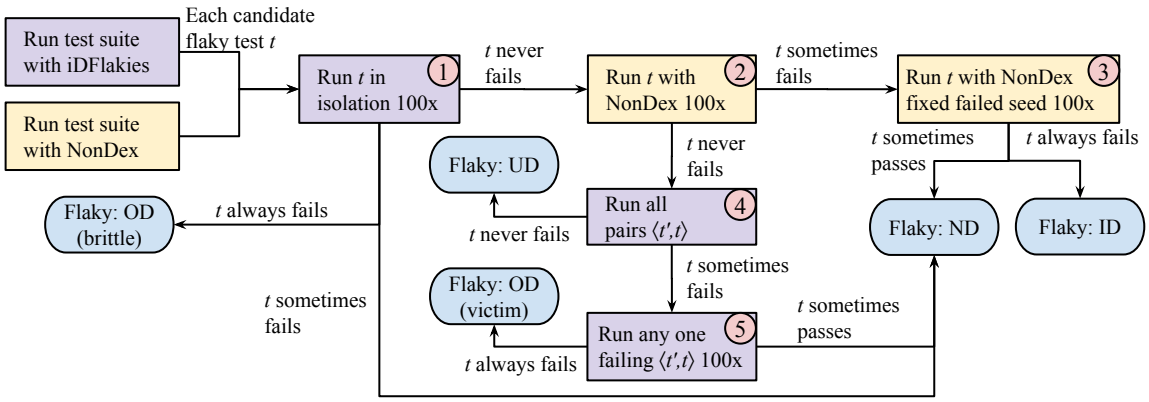


Figure 4.3: Procedure for categorizing flaky tests on the iDFlakies-commit.

for each test (before using the approach outlined in Section 4.2.1 to search for Flakiness-introducing commits).

Running the procedure to find the category of flaky tests is important for several reasons. First, it helps us to confirm that the test is indeed flaky, i.e., the test fails with iDFlakies or NonDex, and the failures can be reproduced using the Isolation, NonDex, or One-by-one approaches (Section 4.1.3). Second, it provides a lower-cost approach to search for FICs; not knowing the actual category would require running likely irrelevant approaches on many commits (e.g., finding the FIC for an OD Vic test would not benefit from running NonDex). Third, it allows us to analyze whether the results about TIC and FIC vary with the test category, because it could inspire using different strategies to detect flaky tests if one expects a project to mainly have flaky tests of some categories.

Figure 4.3 shows the five steps we perform to categorize each flaky test into the following categories:

- *Order-dependent brittle* (OD Brit) – tests that fail when run in isolation but pass when run after some specific tests;
- *Order-dependent victim* (OD Vic) – tests that pass when run in isolation but fail when run after some specific tests;
- *Non-deterministic* (NOD) – tests that non-deterministically pass or fail with no changes to test execution order or implementation of test dependencies;
- *Implementation-dependent* (ID) – tests that are dependent on a specific implementation of an API whose specification admits other implementations;
- *Unknown-dependent* (UD) – tests that are not dependent on the preceding factors.

As the figure shows, we take all flaky tests detected by iDFlakies or NonDex in their iDFlakies-commit and first run each test t in isolation (Step 1). If t always fails, it is categorized as OD Brit. If t sometimes fails but not always, it is categorized as NOD. Lastly, if t never fails, we run NonDex on t with various random seeds (Step 2). If t fails at least once, we proceed to rerun NonDex but now ensuring that NonDex uses the same seed for all of the reruns (Step 3). This seed is arbitrarily chosen from all seeds with which t failed in Step 2. Note that unless a seed is specified, NonDex, by default, generates a new random seed for every run that it does. If t always fails when we rerun NonDex, t is categorized as ID; otherwise, it is categorized as NOD. Back to Step 2, if t never fails, we proceed to run OBO with t (Step 4). If t fails at least once in Step 4, we proceed to run any one failing pair (i.e., $\langle t', t \rangle$) many times (Step 5). We arbitrarily choose any pair that failed in Step 4. If t always fails in Step 5, t is categorized as OD Vic. If t sometimes passes in Step 5, it is categorized as NOD. Back to Step 4, if t never fails, t is categorized as *Unknown-dependent* (UD); essentially, a test that was detected as potentially flaky by iDFlakies or NonDex, yet we are unable to reproduce the flaky-test failure using either Isolation, NonDex, or OBO.

Our use of the targeted approaches could not confirm these UD tests as flaky, and hence, we are not able to automatically categorize them further. When we investigate some of these tests, we find that one reason for these tests is because they are OD Vics that require more than one test to pollute the state, e.g., a test t that passes both in $\langle t', t \rangle$ and $\langle t'', t \rangle$ but fails in $\langle t'', t', t \rangle$. Our prior work [186] has found that such cases are rather uncommon (only 3% of all OD Vics require multiple tests to fail). Nevertheless, our inspection of some UD tests finds that three OD Vic tests from `apache/incubator-dubbo`'s `RegistryProtocolTest` class fail only when they are run following two polluters. Our OBO approach misses these tests (as it searches for polluters but tries only one at a time), resulting in these OD Vic tests being categorized as UD tests.

Another reason for UD tests is because iDFlakies uses "testrunner", a custom Maven plugin to run tests [202] so that it can control the order of all test methods, while all of our targeted approaches use Maven Surefire [132] to run tests. Differences in these two plugins and the JUnit versions supported by them cause some UD tests. Specifically, the testrunner plugin uses only JUnit version 4.12 to run tests, while all of our targeted approaches use Maven Surefire version 3.0.0-M6 (with our changes to support the OBO approach added), which uses JUnit version 4.13 to run tests. Finally, besides the differences in the way tests are run, the tests may be flaky but rather hard to reproduce, or there may be a bug in the iDFlakies or NonDex tools that reported the tests as flaky. Some of these tests also may appear flaky due to simple deficiencies of our experimental infrastructure; running a test suite in two parallel runs (with two tools) can lead to spurious failures, e.g., if the test suite

tries to bind to a specific network port, or if one rogue test runs and fills up disk space. As we are not familiar with the evaluated projects, we could not inspect in detail all hundreds of failures, so we focus our evaluation only on the tests where we can reproduce flakiness in our Isolation, NonDex, or OBO approaches.

To detect flaky tests in their TIC, we follow the same procedure shown in Figure 4.3 as detailed above, but on the tests' TICs instead of their iDFlakies-commits. Specifically, if a test in its TIC fails at least once in steps 1, 2, or 4, then we conclude that the test is flaky in its TIC.

4.2.3 RQ3: When Should One Run Flaky-Test Detectors?

The main goal of RQ3 is to study whether some code change characteristics could help one identify FICs. After identifying the FIC as described in Section 4.2.1, we further inspect which change from the FIC actually introduced flakiness. Commits that introduce flakiness in the existing tests may have a variety of changes—in the test code itself (`src/test` in Maven projects), in the code under test (`src/main` in Maven projects), or in the library dependencies (specified in `pom.xml` files in Maven projects). For this RQ, we want to identify the precise flakiness-introducing change. We compare the files changed in the flakiness-introducing change with those changed by other commits in-between the introduction of the flaky test (TIC) and the first commit in which the test is flaky (FIC). Doing so allows us to compare the changes of FICs with other commits, and to understand if there are simple suggestions that developers could follow to guide when they should run detectors. However, for commits with many changes (e.g., a refactoring commit coupled with some flakiness introducing change), this automated approach does not provide much detail into the precise change that introduced the flakiness. In theory, we could use delta debugging [229] to automatically identify the change (or, more precisely, a set of changes), but in practice, there are no readily available tools for Java and Maven projects. Therefore, we use a combination of manual delta debugging, coupled with the identification of the root cause of flakiness, to identify the exact change(s) in the commit that introduce the flakiness. Specifically, we manually inspect the code changes for each FIC to understand the semantic meaning of the changes and how such changes cause the test to be flaky. We also study the syntactic code changes, such as the location in which the changes were that cause the test to be flaky.

Obtaining Time Between TICs and FICs. Another goal of RQ3 is to understand whether detectors, which often have a configurable number of times in which they run, should be scaled to run more or less for specific tests as the number of commits or the

number of days between the current commit and the test's TIC increases. For example, if a test is added and one uses isolation with 100 runs for this test now, should one continue to run isolation with 100 runs for this test on later commits? To study this question, we find the number of commits and days between the commits of the tests whose TIC is not the same as its FIC. We obtain the number of commits by counting the commits on the ancestry path of the two commits and obtain the time difference by taking the difference of the timestamps in the commits' metadata.

4.3 STUDY RESULTS

4.3.1 RQ1: How Effective Are Flaky-Test Detectors if Run Only When Tests Are Introduced?

Table 4.2 shows an overview of the number and category of flaky tests that our experimental procedure (described in Section 4.2.1 and Section 4.2.2) detected in each project's iDFlakies-commit, as well as in each test's respective TIC. Overall, our procedure detected 684 potentially flaky tests in 55 projects. These high numbers of tests and projects show that the problem of flaky tests is widespread, even among well-tested projects developed by open-source communities and companies, e.g., Apache, Google, Spotify, or VMWare. The high number and diversity of projects also increase the generalizability of our results, as we hope that these projects are a representative sample of all projects, particularly ones whose primary language is Java. The number of flaky tests per project ranges from 1 (for several projects) to 104 (for `apache/incubator-dubbo`). Of 684 tests, our procedure for categorizing flaky tests (described in Section 4.2.2) confirmed 432 as flaky.

For each of the 432 flaky tests that our procedure detected and confirmed on the iDFlakies-commit, we found the first commit that introduced that test (TIC) as described in Section 4.2.1, and we then attempted to compile that revision of the module under test and to run that test. We did not attempt to run UD tests on their TIC: if we could not reproduce their flakiness even on the iDFlakies-commit, it is unclear what conclusions we would be able to reach by attempting to reproduce the flakiness on a prior commit. We succeeded in compiling the TIC and running the test for 245 flaky tests. Unfortunately, many other tests could not compile for a variety of reasons; in many cases, the Maven build system reported failures downloading dependencies that were no longer available (e.g., old `SNAPSHOT` versions that were not intended for long-term use, or dependencies hosted on websites that are no longer active), and in some cases, the problem was that the version of the Java programming language required on the TIC is different than that required on the iDFlakies-commit. While

Table 4.2: Flaky tests detected in each project of our study. After detecting and categorizing flaky tests in the iDFlakies-commit, we find the first commit that introduced each of those tests (TIC), try to compile and run each test, and then determine if each test is flaky or not. Due to difficulties in compiling old versions of projects, we are able to run only 245 of the total 432 (=684-252) confirmed flaky tests on their TIC.

	Flaky Tests (on iDFlakies-commit)						**Flaky Tests Detected/Run (TIC)**				
Project	Total	OD Vic	OD Brit	ND	ID	UD	Total	OD Vic	OD Brit	ND	ID
activiti/activiti	23	12	8		1	2	20/21	12/12	8/8		0/1
alibaba/fastjson	34			1	33		28/33			0/1	28/32
apache/hadoop	27	3		1	9	14	3/3	1/1			2/2
apache/hbase	5			1	2	2	0/1				0/1
apache/incubator-dubbo	104	24	1	1	1	77	6/10	5/9		1/1	
apache/struts	10	4		4	2		1/1				1/1
apereo/java-cas-client	1			1			1/1			1/1	
c2mon/c2mon	5			2	1	2					
codingchili/excelastic	1					1					
ctco/cukes	1	1					1/1	1/1			
davidmoten/rxjava2-extras	2			1		1	1/1			1/1	
doanduyhai/achilles	29	1	1	1	1	25	2/2			1/1	1/1
dropwizard/dropwizard	14	1			12	1	6/13	0/1			6/12
eclipse-ee4j/tyrus	8			2	2	4	4/4			2/2	2/2
elasticjob/elastic-job-lite	9	5		2	2		6/9	4/5		2/2	0/2
espertechinc/esper	35			4	23	8	0/1				0/1
feroult/yawp	1			1			0/1			0/1	
fhoeben/hsac-fitnesse-fixtures	11		1		10		11/11		1/1		10/10
flaxsearch/luwak	2			2							
fluent/fluent-logger-java	6			2	1	3	2/3			2/2	0/1
fromage/redpipe	1				1		1/1				1/1
google/jimfs	1					1					
hexagonframework/spring-data-ebean	1		1				1/1		1/1		
javadelight/delight-nashorn-sandbox	2			2			2/2			2/2	
jfree/jfreechart	1	1									
jhipster/jhipster-registry	2	1			1		2/2	1/1			1/1
kagkarlsson/db-scheduler	10			1		9	1/1			1/1	
kevinsawicki/http-request	28	28					0/1	0/1			
ktuukkan/marine-api	12	12					12/12	12/12			
logzio/sawmill	1					1					
looly/hutool	1				1		1/1				1/1
nationalsecurityagency/timely	8				4	4	3/3				3/3
openpojo/openpojo	24	5			2	17	4/5	4/5			
orbit/orbit	5				5		3/5				3/5
oryxproject/oryx	1				1		1/1				1/1
pholser/junit-quickcheck	13				13		12/12				12/12
pippo-java/pippo	2					2					
querydsl/querydsl	7			5	2		2/3			2/2	0/1
ripe-ncc/whois	4				4		4/4				4/4
sonatype-...-community/nexus-...-helm	2			1	1		2/2			1/1	1/1
spinn3r/noxy	4				2	2	0/1				0/1
spotify/helios	2			2			0/2			0/2	
spring-projects/spring-boot	15	3	1	1	7	3	1/1				1/1
spring-projects/spring-data-envers	2	2									
spring-projects/spring-ws	7	2			5						
tbsalling/aismessages	2	2					2/2	2/2			
tools4j/unix4j	1	1					0/1	0/1			
tootallnate/java-websocket	7			6		1	6/6			6/6	
undertow-io/undertow	11			4	1	6	3/3			3/3	
vmware/admiral	58			12	9	37	5/6			2/2	3/4
wikidata/wikidata-toolkit	5	2	3				2/5	2/2	0/3		
wildfly/wildfly	80	43	7	1	25	4	16/39	3/25	5/6		8/8
wro4j/wro4j	12			3	2	7	4/4			2/2	2/2
wso2/carbon-apimgt	4				4		1/1				1/1
zalando/riptide	20	2				18	1/2	1/2			
Total	684	155	23	64	190	252	184/245	48/80	15/19	29/33	92/113

we could extend our experiment to search for the first *compilable* commit (after the test's TIC) and then run the test on that commit, it would not precisely capture some of the key questions that we consider, e.g., should one run flaky-tests detectors on new vs. modified vs. unchanged tests.

Table 4.2 (right-hand side) summarizes the analysis results for the 245 flaky tests that were runnable on their respective TIC. Of the 245 flaky tests, 184 tests (75%) are detected as flaky tests on their TIC, but 61 tests (25%) are not detected as flaky tests on their TIC. The large fraction of flaky tests detected as flaky in their TIC indicates that it is beneficial to run flaky-test detectors for tests just added in a test suite. In fact, doing so could detect 75% of the flaky tests in our study exactly when it is the best for the tests to be fixed (in their TIC). Nevertheless, there is still a relatively large fraction (25%) of flaky tests that are not detected as flaky on their TIC, which raises the importance of further studying these cases in depth.

We first consider how the fraction of tests being flaky or not on their TIC varies across the projects. The fraction ranges from 0% (no test detected as flaky on their TIC) to 100% (all tests detected as flaky on their TIC) across the projects. For the tests that we could compile on their TICs, we find that 25 (54%) of the projects detect all of the tests as flaky on their TICs. The largest number is for `ktuukkan/marine-api` and `pholser/junit-quickcheck` where all 12 tests that are flaky in the iDFlakies-commit of these two projects are also flaky in their respective TIC. (Note that the TIC for different tests can differ, and in fact, it does differ among these 12 tests for both projects.) The smallest number is for several projects with only one flaky test in the iDFlakies-commit, where that one test is also detected in its TIC. Overall, there are 21 projects that have at least one test not detected in its TIC. 14 of those projects have a mix of tests detected and not detected in their TIC. An outlier that stands out is the project `wildfly/wildfly`, where 23 out of 39 tests are not detected on their TIC. As we describe in Section 4.4.1, this outlier is largely because there are 22 OD Vics in this project that are all not flaky when added, but all become flaky later due to one change. The remaining 7 projects with at least one test not detected actually have none of their tests detected in their TIC. Overall, our results show that many projects have tests that become flaky some time after the test is introduced (TIC), so it is worthwhile to understand when and how flakiness is introduced (FIC) as we do in Section 4.3.3 and Section 4.4.

4.3.2 RQ2: How Do Flaky-Test Categories Affect the Effectiveness of Running Flaky-Test Detectors Only When Tests Are Introduced?

As the different state-of-the-art flaky-test detectors used in our study target different

categories of flaky tests, a natural question to ask is whether all detectors can be equally successful in detecting flaky tests in their TIC. If some categories of flaky tests tend to not be flaky in their TIC, but become flaky only later, then running the detector only on the TIC may provide a false sense of safety, and distributing the efforts for flaky-test detection across multiple commits may be more effective.

To better understand this problem, our experimental procedure (as illustrated in Figure 4.3 and described in Section 4.2.2) categorizes the flaky tests we study into four different categories. In total, we were able to categorize 432 tests, the majority of 684 potentially flaky tests, as definitely flaky tests, with a reproducible category. The remaining 252 flaky tests are categorized as unknown-dependent (UD): our experiments with iDFlakies or NonDex reported the tests as both passing and failing in various runs of the test suite, but it was not possible to ever reproduce that flakiness outside of the full test suite, in our Isolation, NonDex, or OBO approaches. More details about these 252 UD tests are in Section 4.2.2.

Considering the category of flaky tests, we find a great diversity across the projects and categories. In particular, of the 432 flaky tests that our procedure categorized in the iDFlakies-commit, 155 tests were categorized as order-dependent victim (OD Vic), 23 tests were categorized as order-dependent brittle (OD Brit), 64 tests were categorized as non-deterministic (NOD), and 190 were categorized as implementation-dependent (ID). Recall that order-dependent flaky tests are deterministic in their outcome when run in isolation from the test suite (and thus likely deterministic modulo the tests that have run prior to that test in the test suite), and implementation-dependent flaky tests are deterministic in their outcome when run in isolation for a specific seed. A detailed breakdown of each project's number of categorized tests is in Table 4.2.

The results of our categorization show that across all projects, OD Vics are less likely to be flaky in the TIC (60%) than the overall average of 75%, while the other three categories tend to be flaky more than average (79% of OD Brits, 88% of NODs, and 81% of IDs). Our finding that OD Vics tend to be less likely to be flaky when they are added is likely because OD Vics, unlike the other three categories, rely on another test to fail. Therefore, when OD Vics are added without the other test to make them fail, they cannot be detected when added. In contrast, our results suggest that OD Brits, NODs, and IDs, which are all tests that, when each is run by itself, can result in a flaky-test failure, do have a high likelihood to be detected in their TICs.

4.3.3 RQ3: When Should One Run Flaky-Test Detectors?

To better understand when a test becomes flaky, we analyze in detail (mostly manually

Table 4.3: Flaky tests not detected as flaky on their Test-introducing commit (TIC) and the information about their Flakiness-introducing commit (FIC).

Project	Test	Category	FIC Files Changed				Commits from TIC-FIC Changing					Days TIC-FIC
			TC	TS	CUT	B	TC	TS	CUT	B	Any	
activiti/activiti	testErrorC...	ID	X	X	X		6	1,784	2,600	642	3,294	1,177
alibaba/fastjson	test_1	ID	X	X	X		5	498	607	140	770	1,217
	test_date...	ND	X	X			1	1	1		1	2
	test_for...	ID	X	X			1	16	13	3	20	7
	test_list...	ID	X	X	X		1	3	4		4	1
	test_rese...	ID	X	X	X	X		1			1	1
apache/hbase	testConcur...	ID		X		X	2	412	525	74	726	122
apache/incubator-dubbo	testBindin...	OD-Vic		X	X		1	20	29	7	38	22
	testClearR...	OD-Vic		X	X			20	29	7	38	22
	testGetInv...	OD-Vic		X	X			7	4	3	16	9
	testGetInv...	OD-Vic		X	X			7	4	3	16	9
dropwizard/dropwizard	customJson...	ID	X	X	X	X	3	206	221	226	554	582
	customJson...	ID	X	X	X	X	3	206	221	226	554	582
	customJson...	ID	X	X	X	X	3	206	221	226	554	582
	printsDidY...	ID	X	X	X		23	234	257	179	552	379
	testLogbac...	OD-Vic		X			8	664	754	539	1,615	1,143
	testPretty...	ID	X	X	X	X	1	73	75	99	197	171
	testPretty...	ID	X	X	X	X	1	73	75	99	197	171
elasticjob/elastic-job-lite	assertExec...	ID	X	X	X		6	130	166	32	225	82
	assertPers...	OD-Vic	X	X	X		3	52	61	29	116	148
	assertUpda...	ID		X	X		4	78	118	47	145	72
espertechinc/esper	testRegres...	ID				X	15	128	227	32	254	650
feroult/yawp	testFlowDr...	ND			X		1	2	4		4	0
fluent/fluent-logger-java	testReconn...	ID		X	X		10	25	33	10	57	740
kevinsawicki/http-request	basicProxy...	OD-Vic	X	X	X		3	8	16	10	38	28
openpojo/openpojo	shouldSkip...	OD-Vic			X			10	11	5	21	20
orbit/orbit	testConstr...	ID			X	X	3	23	76	71	143	408
	testDefaul...	ID			X	X	3	23	76	71	143	408
querydsl/querydsl	execute2	ID			X		3	127	142	61	206	188
spinn3r/noxy	testBulkCl...	ID	X	X	X							0
spotify/helios	testUndepl...	ND	X	X			20	76	116	17	154	160
	verifySupe...	ND	X	X	X		42	652	1,375	415	1,924	830
tools4j/unix4j	find_file...	OD-Vic	X	X	X							0
vmware/admiral	testGetKub...	ID	X	X	X		1	29	50	8	99	36
wikidata/wikidata-toolkit	testMwDail...	OD-Brit	X	X	X		30	243	447	93	522	475
	testMwMost...	OD-Brit	X	X	X		30	243	447	93	522	475
	testMwRece...	OD-Brit	X	X	X		30	243	447	93	522	475
wildfly/wildfly	testBindAn...	OD-Vic			X	X	30	9,929	17,810	7,301	23,448	2,648
	testBindAn...	OD-Vic			X	X	30	9,929	17,810	7,301	23,448	2,648
	testBindRe...	OD-Vic			X	X	82	7,342	11,223	5,732	15,000	2,048
	testCompos...	OD-Vic			X	X	51	1,228	1,526	1,056	2,273	333
	testCompos...	OD-Vic			X	X	51	1,228	1,526	1,056	2,273	333
	testCreate...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testFireMu...	OD-Vic			X	X	6	10,058	18,225	7,413	23,933	2,717
	testFireOb...	OD-Vic			X	X	6	10,058	18,225	7,413	23,933	2,717
	testInitia...	OD-Vic			X	X	10	10,061	18,228	7,413	23,936	2,717
	testJavaCo...	OD-Vic			X	X	10	10,061	18,228	7,413	23,936	2,717
	testListBi...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testListBi...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testListBi...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testListNa...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testListWi...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testList	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testLookup...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testLookup...	OD-Vic			X	X	95	10,061	18,228	7,413	23,936	2,717
	testOnlyEx...	OD-Vic			X	X	75	5,575	8,017	4,912	10,816	1,554
	testPermis...	OD-Brit			X	X	70	6,332	9,398	5,344	12,651	1,764
	testRebind...	OD-Vic			X	X	82	7,342	11,223	5,732	15,000	2,048
	testReject...	OD-Vic			X	X	23	596	651	512	1,056	220
	testReject...	OD-Vic			X	X	23	596	651	512	1,056	220
zalando/riptide	shouldReco...	OD-Vic	X	X	X		2	7	8	7	12	1

with some automated tool support) *all* 61 tests that are not flaky on their TIC. In Section 4.4, we describe in detail how some of these tests become flaky in their FIC. Analyzing each test takes a few hours to first find the appropriate commit that introduces flakiness (FIC), and then to comprehend which exact part of that commit causes flakiness.

TIC-FIC Characteristic Differences for All Flaky-Test Categories. Table 4.3 shows the results of the analysis of what files the FIC changed (and where the change was that introduced the flakiness). In particular, we distinguish the following types of commits: (1) the commit changed the code of the test itself (note that this need not be just the test method body but can also include modifying the `@Before` or `@After` parts in the test class or its superclasses), (2) the commit did not change the test itself but did change other tests in the test suite (which is relevant for studying OD Vic cases), (3) the commit did not change any test code but did change the code under test, and (4) the commit did not change any source code of the project under test but did change the build configuration, for instance, changing dependencies in a `pom.xml`. Distinguishing these types of commits is important to determine what strategy one could use to run flaky-test detectors on various commits. Running such detectors is generally costly (i.e., they require multiple runs of the entire test suite or at least some tests, using various random seeds or other causes of "noise" [22]), so projects typically do not run these detectors on all tests for each and every commit. For example, Mozilla runs its "test verification" [201] only on tests newly added or modified in a commit.

As described in Section 4.3.1, we find that 75% of flaky tests in our study are detected as flaky on their TIC. From column "TC" (Test Class) in Table 4.3, we further find that 24 of 61 (39%) tests not flaky on their TIC become flaky in a commit that changes the code of the class containing the flaky test. Combining these 24 tests and the 184 tests that are flaky when they are added, we find that 208 of 245 (85%) tests can be detected by running flaky-test detectors on newly added or existing, but directly modified tests. However, it still leaves 15% of flaky tests that become flaky due to changes *not* being directly in the test class itself but rather elsewhere in the test suite, code under test, or library dependencies. As a result, if the goal is to automatically identify flakiness soon after it is introduced (or ideally right when it is introduced), it is necessary to do more than simply running flaky-test detectors on newly added or directly modified tests.

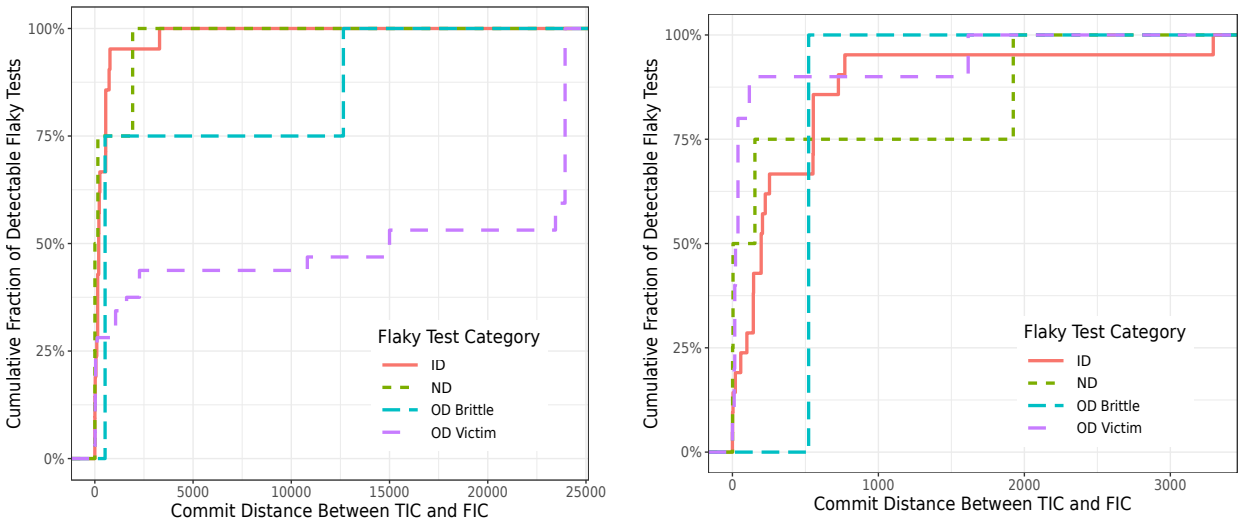
A straight-forward approach would be to run detectors on all tests and limit detector runs to the proximity of the TIC, or perhaps to run detectors less as the distance to the TIC increases. To assess the impact if one were to do so, we investigate when these tests become flaky following the methodology outlined in Section 4.2.1. Once we know the FIC for a test, we analyze the *commit distance* between its TIC and FIC, i.e., the number of commits and

days (Section 4.2.3) between the introduction of a test and when it becomes flaky.

When we analyze the commit distance across all 61 flaky tests that we detected in commits after the TIC, we find a high average distance of 7089 commits between TICs and FICs with the median distance being lower at 554 commits. Running flaky-test detectors for such a large number of consecutive commits is likely prohibitively expensive for most organizations. Therefore, we conclude that (1) flaky tests are often flaky when they are added and (2) flaky tests that are not flaky when added typically do not become flaky for many commits after their TIC.

As developers at some organizations, such as Mozilla [201] and Netflix [149], run detectors on newly added or modified tests, we next explore how long it takes a flaky test class to be changed if the flaky test class was not changed in the FIC. Specifically, 39% (24 out of 61) of the FICs included changes to the test class containing the flaky test; the other 61% (37 out of 61) did not. Hence, only running flaky-test detectors on tests that are changed could not have detected the FICs for 37 tests immediately. To understand the delay to detect these 37 tests if detectors were run only on test classes that are changed, we search for such changes in between the FIC and iDFlakies-commit. We find that changes to the test class containing the flaky tests happened for only 8 of the 37 tests. For these 8 tests, the commits are a median of 62 commits and 22 days after the FIC. The remaining 29 tests do not have any changes to the test class containing the flaky test after the FIC, so these tests could not be detected if detectors were run on only modified test classes even for all commits. If detectors are run on all tests for all commits that change test code, the remaining 29 flaky tests would be detected in the median of only 3 commits or 3.6 days after the FIC. However, because most commits have changes to some test code, as shown in Table 4.3, running flaky-test detectors on all tests whenever tests are changed would still be prohibitively expensive. Hence, we suggest that detectors should be run when tests are added and later detectors may be suspended for a large range of commits to achieve a good detection-to-cost ratio. The range of commits depends on developers' budget for running detectors. Excluding `wildfly/wildfly` tests (being 38% of tests), we find medians of 144 commits and 154 days between TIC and FIC. Thus, one may consider running detectors periodically, say, every 150 commits.

TIC-FIC Characteristic Differences for Different Flaky-Test Categories. We next explore how the characteristic differences of TICs and FICs may vary depending on the category of the flaky test. Either on the test's TIC or after a change to its test class, our results show that 65% of OD Vics (48 from TIC + 4 from modifying test class / 80 runnable on TIC), 95% of OD Brits (15 from TIC + 3 from modifying test class / 19 runnable on



(a) Including wildfly/wildfly.

(b) Excluding wildfly/wildfly.

Figure 4.4: Flaky-test detectability over commit distance between TIC and FIC.

TIC), 97% of NODs (29 from TIC + 3 from modifying test class / 33 runnable on TIC), and 94% of IDs (92 from TIC + 14 from modifying test class / 113 runnable on TIC) can be detected. It is promising that the majority of OD Brit, NOD, and ID tests become flaky right after their test code has been added or changed, which shows that how Mozilla [201] and Netflix [149] handle new or modified tests is a good strategy for detecting these categories of flaky tests. However, the comparatively low ratio for OD Vic tests (65%) indicates that this strategy may miss a nontrivial fraction of other flaky tests. (If `wildfly/wildfly` with its many undetected tests on TIC is ignored, being an outlier, the percentage of tests detected becomes 90%, which does better justify the strategy of running flaky-test detectors when tests are added or modified.)

Figure 4.4 shows the cumulative distribution functions of flaky-test detectability over commit distance (for any type of change) for the four different flaky-test categories targeted by the detectors in our study. While Figure 4.4a includes all data, Figure 4.4b excludes data from the `wildfly/wildfly` project as it is an outlier due to the extreme commit and time spans between TIC and FIC, and the large fraction of flaky tests associated with one identical TIC-FIC pair. From Figure 4.4a we can see that flaky tests of *ID* type have a much lower commit distance (median 201.5 commits corresponding to a median time difference of 188 days) between TIC and FIC than flaky tests of type *OD Vic* (median 15000 commits, 5.6 years). According to this finding based on our unfiltered data, running detectors for ID flaky tests (e.g., NonDex) would be most effective when concentrated on commits that are relatively close to the TIC, whereas OD Vic detectors (e.g., iDFlakies) are better applied more sparsely. However, if the ID, OD Brit, and OD Vic tests of `wildfly/wildfly` are

ignored, the conclusion changes: while the median numbers for ID tests change only slightly, the median commit distance for OD Vic tests changes to only 38 commits and the median time span to 21 days. For OD Brit tests, we cannot derive a clear tendency, either; Figure 4.4a suggests that the commit distance lies between that for ID and OD Vic tests, but the observation is based on only four data points. 75% of the NOD tests have an even lower commit distance than ID tests. However, the observation is again based on just four data points, and no robust detectors for NOD tests exist, so this result may not generalize and would be difficult to reproduce in repetitions of our study. For these reasons, we do not draw conclusions on OD Brit and NOD tests' commit distances. Nonetheless, our results indicate that efforts for flaky-test detection may be most effective shortly after a test's introduction (e.g., about 20 days for OD Vic test and 180 days for ID tests).

4.4 CASE STUDIES OF TESTS NOT FLAKY AT TEST-INTRODUCING COMMIT

We inspected all tests that are not flaky at their TIC and confirmed the FIC for each test. We next present more details for a number of these tests. We select a diverse set of cases from various categories of flaky tests and from various projects. These cases show that the causes of flakiness, both where the flakiness is and what kind of change introduces it, are rather diverse, so one cannot easily build a general technique to detect all of these causes. For each case, we highlight (in bold) the specific reason for why we selected that case.

4.4.1 Order-Dependent Victim

Victim added before a polluter. Consider the test `HttpRequestTest.basicProxyAuthentication` from the project `kevinsawicki/http-request`. The test is a victim that is added in the commit bf07c2f, but this test does not become flaky until cb9e021. In the FIC, the changes include adding the polluter, adding another test, and some changes to the code under test. The polluter test (`HttpRequestTest.customConnectionFactory`) would call `setConnectionFactory` with a customized `ConnectionFactory`, which consequently causes the victim to fail because the `ConnectionFactory` set by the polluter is not the `ConnectionFactory` that the victim expects.

Ignored test. Consider the test `FindFileTimeDependentTest.find_fileCreatedBeforeNow` from the project `tools4j/unix4j`. In the commit 1c9524d, the test is added, in the sense that the test method is added to the test class. However, the test is added with an `@Ignore` annotation that instructs JUnit not to run the test. This test was committed into the project repository before it was finished, and to indicate that it was not ready to use, the developer

marked it with an `@Ignore` annotation. Two commits later, in the commit dfc4c77, the test was completed, and the `@Ignore` annotation was removed, so then the test becomes flaky. Hence, this test may even be considered flaky when it was first "added" based on how one defines added. Strictly speaking, the test was not flaky in its TIC, based on how we defined the TIC (to be able to objectively find it by mining Git repositories without running all of the tests in all of the commits that we consider).

A non-flaky test becomes a brittle and a polluter. Consider the 22 tests from the project `wildfly/wildfly` that are OD Vics in iDFlakies-commit but not flaky on their TIC. All 22 of these tests become victims for the same reason, namely, they all fail when another test fails. Specifically, the test `WritableServiceBasedNamingStoreTestCase.testPermissions` becomes OD Brit in the commit c22e231 due to a change in a dependency. More details about this OD Brit test are in Section 4.4.2. When this test is run without its state-setter, not only does it fail itself but it also pollutes the state such that 22 OD Vic tests that run after it also fails! That is, if `testPermissions` fails as an OD Brit, then it does not run some clean up code for the `WildFlySecurityManager` class, resulting in that class entering a corrupt state that causes the OD Vics to fail.

4.4.2 Order-Dependent Brittle

Flakiness introduced due to a dependency change. Consider the test `WritableService-BasedNamingStoreTestCase.testPermissions` from the project `wildfly/wildfly`. This test is added in the commit f7a03d7 but becomes OD Brit in the commit c22e231. The FIC changes are relatively small and only modify (1) one dependency (in `pom.xml`) and (2) one file (CredentialSourceDependency.java). The file itself is in the code under test, not in the test code, and even in a module (`clustering`) different from the test class' module (`naming`); however, the test could still be flaky due to the file change. Interestingly enough, our inspection shows that the flakiness is due to the dependency change and *not* due to the file change. This example illustrates that even if a detector ran all tests when the *source* code of a project changes, it would still miss some flakiness introduced due to dependency changes.

Precisely identifying flakiness cause and what part of code it belongs to. Consider the test `MwDumpFileProcessingTest.testMwDailyDumpFileProcessing` from the project `wikidata/wikidata-toolkit`. This case illustrates some interesting points in understanding what changes introduce flakiness and what flaky-test detection strategy could detect that flakiness. This test is added in the commit 1c192a3 and is not flaky then. The test becomes OD Brit in the commit f7cb408. (As a side note, this commit was found automatically by `git bisect`: confirming OD Brit tests is the easiest of all categories, because it requires just

running one test in isolation by itself with no other tool but Maven.) The test does not fail when run in the test suite, because the test suite contains a "state-setter" [186] that happens to run before the brittle, but the test does fail when run by itself.

The changes in the FIC are relatively large, modifying 15 files across 4 modules, and also deleting 1 file and adding 1 file, but the majority of the changes in the commit are merely for renaming or refactoring existing files. Our careful inspection shows that the relevant parts of the commit are (1) changes in the test class and test method itself because of some API change in a class that the test method calls, and (2) changes in another class that actually introduces the flakiness for this test. Specifically, the commit happens to modify the file MwDumpFileProcessingTest.java that contains the test class with the test method of interest. In fact, the body of the test method itself is modified but mostly for refactoring some method calls. If one used a detector that runs all modified tests in isolation, one would detect this test immediately in the FIC. However, the change of the test code in this FIC is *not* what introduces the flakiness. In fact, the commit could have been split into two: one that performs the refactorings and the other that makes the behavior change, which happens to be a real flakiness-introducing change. In such a scenario, the first hypothetical commit would change the test but not make it flaky, and the second hypothetical commit would be the one that actually introduces flakiness; hence, using a detector that runs all modified tests in isolation would *not* run this test in the second commit and would *not* detect this test in its (hypothetical, second) FIC.

The relevant change that introduces flakiness is in the file MockDirectoryManager.java, which is not even in the same module as the brittle's test class. The change makes some (mocked) directory read-only by default, hence the test of interest fails when it attempts to write to the directory. An interesting aspect is how one should even label where this change is for our Table 4.3. The change is in `src/main` and not `src/test`, so it could be considered to be in the code under test, but the change is in a class that is used only for testing: the module is `wdtk-testing`, which provides utility classes for testing other modules in the project, and the class name suggests that it is for mock testing. If we had a hypothetical scenario as described in the previous paragraph, this could create issues for labeling the second commit that would only change this file. As the actual FIC has many changes due to the renaming and refactoring, we label the changes of the FIC as it is, namely, that it changes the test class, test suite, and code under test of this flaky test.

4.4.3 Non-Deterministic

Flakiness introduced due to a change in the test code and ability to precisely

measure flakiness rate. Consider the test `DateParseTest9.test_dates_different_timeZones` from the project `alibaba/fastjson`. This test is added in 8061e09. The test becomes flaky in the commit d296511. The test code is changed, so a flaky-test detection strategy that runs modified tests could detect this flaky test. Interestingly, this test is an NOD flaky test because it depends on the timezone in which the test is run. In the commit ec17139 (about three hours after becoming flaky), a change is made claiming to "fix" the test by making it no longer depend on the timezone. However, the fix does not fully remove the flakiness because it changes the test to not depend on the timezone in which the test is run but instead to depend on a random seed (which itself depends on time) to set a timezone. In fact, this test is an unusual case where we can precisely measure the probability in which the test fails. This test's outcome depends on the length of an array from which the random number picks one index, and the test fails for some indexes but passes for others. On Java version 1.8.0_25-b17, the array has size 613, and 67 indexes fail, so if indexes are chosen uniformly, the probability of failure is 10.9%.

Flakiness due to Concurrency and Async Wait. Two other ND tests which do not fail on their TIC are `FlowDropsTest.testFlowDropsToSameSink` from the project `feroult/yawp` and `SupervisorTest.verifySupervisorStartsAndStopsDockerContainer` from the project `spotify/helios`. In commits where these tests fail, they fail in only 1-2% of runs. We check the FICs of these tests by confirming with 500 runs that the tests do fail on the FICs, while the tests do not fail in the parent commits of the FICs. If a test fails in about 1% of runs, and all runs are independent, then the probability that it fails at least once in 500 runs goes up to 99.3% ($1 - 0.99^{500}$).

For the `testFlowDropsToSameSink` test, the flakiness is due to Concurrency (as defined by Luo et al. [126]), namely, the randomness of thread scheduling in a library dependency. The test at some point asserts that all objects are removed from a list. In the FIC (abae178), only code under test (CUT) is changed, and based on the FIC's change and commit message, some asynchronous code is introduced to improve performance. The introduced asynchronous code may improve performance, but it also causes the `testFlowDropsToSameSink` test to be flaky because the removal of objects from the list may no longer happen before the assertion (depending on the scheduling of threads).

For the `verifySupervisorStartsAndStopsDockerContainer` test, the flakiness is due to Async Wait (as defined by Luo et al. [126]). The code change in the FIC (0232600) shows that flakiness is introduced in both CUT and the test itself. In the parent of the FIC, the test starts a Docker container, does some computation in the container, and stops the container before checking whether the container is stopped. In the FIC, the stopping of the container is changed to be asynchronous with a timeout. In most runs of this test, the stopping of the

container asynchronously can complete before the check, but the check can fail depending on the load of the machine running the test.

4.4.4 Implementation-Dependent

Resolving compilation problems to identify precise FIC. Consider the test `KubeConfigContentServiceTest.testGetKubeConfigWithBearerToken` from `vmware/admiral`. We use this example not only to discuss how flakiness was introduced but also to illustrate some challenges in compiling older versions of projects and searching for the FIC. This test is added in the commit 0d6718d and is not flaky right then. Using `git bisect`, with some manual inspection, we confirmed that the test is still not flaky at 2a96edc but is definitely flaky at 6ccaa16. There are 39 commits in the range 2a96edc..6ccaa16, but the project does not compile for any of them. For this case, we invested extra effort to identify a precise commit in that range that introduced the flakiness.

One reason that the project does not compile is that one of the required modules (`admiral-photon`) fails with an unmet dependency, `com.vmware.xenon:xenon-common:jar:1.5.4_9-SNAPSHOT`. While we cannot find the compiled `.jar` file or source code for such missing dependencies, fortunately, we could find the entire source history for the Xenon project in `vmware-archive/xenon` on GitHub. We studied the dates in Xenon commits between versions `1.5.4_9-SNAPSHOT` and `1.5.4_9`, and the relationship of these dates with the dates of `vmware/admiral` in the range 2a96edc..6ccaa16. The test class of interest was itself changed in two commits in that range, so we focused on those two commits. Fortunately, we found that both of those commits can use `1.5.4_9` instead of `1.5.4_9-SNAPSHOT`, thereby allowing us to restore the same code that the developers of `vmware/admiral` had when they built the code. We were then able to compile the code, run the test, and confirm that the flakiness starts from the commit 75b53f3, with its parent, 934e0a7, not failing with NonDex. This example is relatively simple as a change in the test class introduced flakiness, and the category of flakiness is the same in FIC as in the iDFlakies-commit. However, in terms of the "archaeological" work needed to find the FIC, this example illustrates some of the challenges.

Adding an assertion makes flakiness manifest. Consider the test `ZKTest.testBulkClusterJoining` from the project `spinn3r/noxy`. The test is added in 7029534. While it does *not* fail by itself or with NonDex, interestingly enough, it does produce different values that it prints on the standard output. The very next commit, b26d669, adds an assertion for the value printed on the standard output, and the test then fails with NonDex (but does not fail without NonDex). One could argue here that flakiness existed even in the TIC but simply did not show up in test failures. Zhang et al. [237] call this phenomenon *manifest*

flakiness and argue that one should report only such cases where a test fails. In contrast, Huo and Clause [84] and Gyori et al. [73] argue that one could report even tests that *may* become flaky in the future. Indeed, if a goal is to detect (undesirable) flakiness as soon as it is introduced, one would want to have techniques that report potential flakiness, at the expense of some false alarms. For example, this specific example test could have been found at TIC by reporting that the test prints different values with NonDex.

Flakiness introduced due to a change in the code under test. Consider the test `ConstructionTest.testConstruction` from the project `orbit/orbit`. This test is added in ebcc9ef. The test becomes ID in the commit 553f955. The change is in the code under test (not in the test code), specifically in the file Stage.java. The change replaces an invocation of a constructor, `new JGroupsClusterPeer()`, with an invocation through reflection, `JGroupsClusterPeer.class.getConstructors()[0].newInstance()`. More precisely, the class object is created from a string to avoid direct compilation dependency on the class `JGroupsClusterPeer`. The class `JGroupsClusterPeer` has two constructors, and the specification of `getConstructors()` does not specify in what order they should be returned, so indexing at offset `[0]` can return the other constructor (with one argument), not the no-argument constructor. At a glance, one may expect that `newInstance()` could not be invoked on the other constructor, so it would raise an exception right then. However, the constructor actually gets invoked with the `null` value for the argument (which is stored in a field and then leads to an exception much later when the field is referenced). This example illustrates how a change in the code under test can introduce flakiness and also how some of these cases can be challenging to analyze and debug.

4.5 DISCUSSION

Our study has a variety of implications for researchers and developers alike. By detecting and then categorizing flaky tests as order-dependent brittle, order-dependent victim, implementation-dependent, non-deterministic, and unknown-dependent (Section 4.3.2), we provide guidance to researchers creating new flaky-test detectors. In particular, we found that 252 of the 684 flaky tests that we detected were *unknown-dependent*, and could not be easily categorized outside of the test suite. This result shows the need for the community to develop new flaky-test detectors that do not focus on order or implementation dependence.

Our study provides empirical evidence to support the adoption of techniques used in proprietary software development for finding flaky tests—specifically, because 85% of flaky tests are flaky when added or directly modified, developers can extensively run flaky-test detectors on added or modified tests. However, we found that this result varies across

projects: in some projects, all flaky tests were flaky on their TIC, and in others, none were. Hence, researchers studying the applicability of flaky-test tools should consider a diverse set of projects. We carefully investigated all cases where tests were *not* flaky on their TIC, and found a range of explanations for why these tests become flaky later on. Because no common, simple explanation for why tests become flaky later exists, we suggest that developers run flaky-test detectors not only when tests are added or modified, but also with a minimum regular frequency (e.g., monthly). We make our entire dataset publicly available (including the category of each flaky test, its TIC, and its FIC), so future research can use this data for evaluations and to better detect and categorize these tests [1].

4.5.1 Threats to Validity

We next discuss threats to validity of our study following the classification by Wohlin et al. [221].

Conclusion Validity. While the main conclusions of our study do not rely on inferential statistics, we do apply such methods for some of our work. For the correlation analysis of flaky test numbers and project characteristics, we chose Kendall's τ, as it is non-parametric and more robust than Spearman's ρ [27]. We conduct an equality of proportions test to determine whether the number of flaky tests increases proportionally with various project characteristics. To determine whether the number of flaky tests grows under- or over-proportionally with characteristics, we investigate scatter plots (Figure 4.1) and find that the results from investigating the plots confirm the validity of the proportionality test's conclusion.

Internal Validity. Flaky tests are non-deterministic by nature, and hence create a number of potential threats to the validity of our conclusions. We believe that we have identified and taken steps to mitigate many of these concerns. For instance, it is possible that some flaky tests were not detected by iDFlakies or NonDex on the iDFlakies-commit. We ran the detectors following their recommended configurations, but acknowledge that these detectors do not guarantee the detection of every flaky test. Furthermore, the flaky tests that we study come from only two detectors. Unfortunately, there are no other publicly available flaky-test detectors that are robust enough to run on our set of Java projects, and hence, we could not extend and compare our results with those other detectors. Nevertheless, our primary goal is to determine when flaky tests become flaky, so even if we studied only a fraction of all flaky tests in our projects, we believe that this selection can be representative so our results may generalize to the other flaky tests in these projects.

To determine when each flaky test first becomes flaky, we face the risk that it might be flaky on an earlier commit than observed in our experiments. This issue can, again, be due to the inherent non-determinism of flaky tests. We alleviate this concern by categorizing flaky tests into categories that allow us to more precisely reproduce them. For example, for every test that we categorize as an OD Vic (i.e., a specific "polluter" test running before the victim causes the victim to fail) we also find which tests pollute the shared state. With the polluter, we can deterministically reproduce the failure of the victim when we run it after the polluter. Note that running OBO for detecting OD Vics can miss tests whose failures depend on *multiple* other tests. Such cases have previously been found to be rare though [186]. The NonDex tool similarly allows for a rather consistent reproduction of ID flaky-test failures. Moreover, we manually inspected all of the cases where a test was not found flaky on the TIC, providing even greater confidence in our identification of FICs.

The infrastructure that we built to locate TICs may also contain faults that could have affected our results. To mitigate this threat, our infrastructure outputs all commits that it considers to be a potential TIC of a flaky test. The dissertation author and a collaborator randomly sampled the logs of these tests to confirm the TICs of the sampled tests. The remaining parts of our experiments largely involve using Maven to run tests in isolation and a simple, yet effective custom Maven Surefire to run tests in OBO. We do rely on iDFlakies and NonDex to help us detect flaky tests, and these detectors themselves may also have faults that could have impacted our results. We attempt to mitigate this threat by analyzing a sample of the logs produced by Maven, iDFlakies, and NonDex, and automatically confirming flaky tests detected by the latter two using the procedure in Figure 4.3.

Due to the fact that flaky tests non-deterministically pass or fail, our results may not be easily reproducible, especially flaky-test failures. We attempt to mitigate this threat by rerunning every potential flaky test detected by iDFlakies and NonDex, and categorizing tests as UD when they are not reproducible to prevent such tests from affecting the results of our research questions. By default, tools such as iDFlakies also reruns a suspected flaky test in its failing and passing order before outputting its category. Nevertheless, it may still be possible that the tests found to be OD from iDFlakies or our reruns are actually not OD tests. Note that the inverse would not be possible because once a test is observed to pass and fail in one order, it cannot be an OD test.

Construct and External Validity. The results from our study may not generalize to a larger population of projects, flaky tests, or flaky-test detectors. As described in Section 4.1.2, we attempt to mitigate this threat by selecting a large and wide variety of projects to evaluate. Our projects are obtained from iDFlakies, therefore our results may be biased

the same way that our previous results may be biased [108] (e.g., finding OD tests more frequent than they really are). However, the diversity of our results (e.g., high numbers of both OD and ID tests) between different projects do suggest that our sample of projects and tests may be representative of other projects. We acknowledge the limitation of our presented results to the chosen detectors, but we are not aware of other publicly available detectors that work for the large corpus of projects that we studied and offer the degree of automation that is mandated by a large scale study like ours.

4.6 SUMMARY

Our study has first identified a large corpus of flaky tests, then categorized them by the category of flakiness, and finally traced those tests back in time to determine when they first become flaky. We have detected 245 flaky tests that we could compile and run on their Test-introducing commit (TIC). We find that 75% of these tests are flaky when added, indicating that there can be substantial value for developers to run flaky-test detectors specifically on newly added tests. We also find that 85% of flaky tests are flaky when added or directly modified, confirming the benefits of the approach taken by organizations such as Mozilla [201] and Netflix [149]. However, the remaining 15% of flaky tests become flaky due to other changes, suggesting the need for future work to better detect such tests.

One option would, on every revision, run detectors on the *entire test suite* (not just on the newly added or directly modified tests) using a lot of randomization, e.g., a combination of iDFlakies to randomize the test order, NonDex to randomize the choices of non-deterministic libraries, and even more "chaos" mode tools to randomize other choices, e.g., the thread schedules [22]. The test suite could be run only once, thus taking about the same amount of time as if not using any randomization, but increasing the chance to detect some flaky tests soon after they become flaky. While such an approach could indeed detect flaky tests, it would make debugging of test failures more difficult: any test failure could be either due to the recent code changes or due to the randomizations. The question of whether a test suite should be by default run in a random order (similar to iDFlakies) led to a lengthy discussion for the RSpec testing tool for Ruby [172], with both sides passionately arguing why their position is the "right" default. In the end, the decision was to not have randomization by default, decreasing the chance to detect flaky tests. In the future, we expect similar debates to be reopened, and our study provides motivation for understanding the trade-off for what to run at which points in development. To further improve on the early detection of flaky tests, we need more advanced ways to determine when flaky tests become flaky, so we can use the various detectors efficiently. We may also need to adopt proactive techniques that

find potentially flaky tests even if they cannot yet manifest in flaky-test failures. To help with future research, we make our dataset of flaky tests with labeled TICs and FICs publicly available [1].

CHAPTER 5: [CHARACTERIZATING] ROOT CAUSING FLAKY TESTS IN A LARGE-SCALE INDUSTRIAL SETTING

This chapter presents RootFinder, a technique developed with Microsoft collaborators to automatically root cause flaky-test failures. Section 5.1 presents some background on Microsoft's build and test system. Section 5.2 presents our end-to-end framework, which includes RootFinder, to help developers root-cause flaky tests. Section 5.3 presents our case study from applying the framework at Microsoft, and Section 5.4 presents the threats to validity of our work. Finally, Section 5.5 presents some open research challenges, and Section 5.6 concludes this chapter.

5.1 BACKGROUND ON MICROSOFT'S BUILD AND TEST SYSTEM

Microsoft uses a modern build and test service system in the cloud, called *CloudBuild* [49]. Similar to other systems such as Bazel [14] from Google and Buck [21] from Facebook, CloudBuild builds code and runs tests incrementally and in a distributed manner. When CloudBuild receives a build request with code changes, CloudBuild identifies all modules that are impacted by the changes. CloudBuild then executes the tests only in those impacted modules, and skips the remaining modules' tests, because none of their dependencies changed. Note that, within a module, CloudBuild always executes all tests in the same order (sorted alphabetically). CloudBuild reruns failing tests once to see if the retry passes or not. If the retry passes, then the test is identified as flaky and would not prevent changes from being merged. If the retry fails, then the test is identified as a regression and the changes cannot be merged. As of 2019, CloudBuild was used by ≈1200 projects inside Microsoft and executes ≈350 million unit tests per day across all projects.

5.2 END-TO-END FRAMEWORK

We next present our framework for identifying the root causes of flaky test failures. The framework consists of three main steps. Figure 5.1 shows an overview of our framework. Our framework takes as input a flaky test and its dependencies (i.e., test binary, its dependent source binaries, and relevant test data), so that the test can be executed on any machine.

Instrument tests and dependencies. Our framework first produces instrumented versions of the flaky test and all of its dependencies using an instrumentation framework, called Torch [91, 125]. The instrumentation helps us log various runtime properties of the test

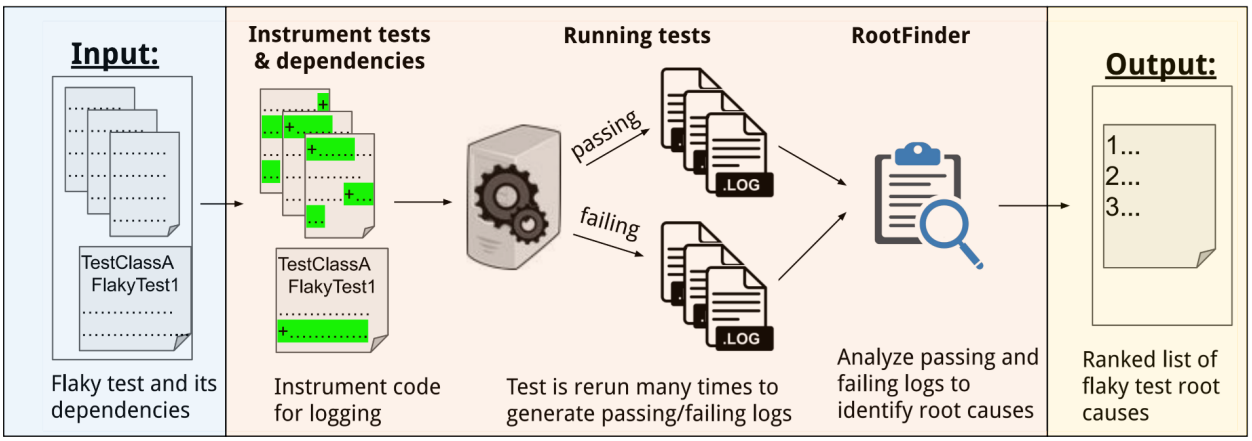


Figure 5.1: Overview of our framework to root cause flaky tests.

execution. Note that except for the overhead from logging runtime properties, the instrumented binaries retain the same functionalities as the original binaries, and therefore, tests can seamlessly run on Torch-instrumented binaries.

Running tests. Using the instrumented version, we next run the flaky test **100** times on a local machine in an attempt to produce logs for both passing and failing executions. The logs generated by Torch contain various runtime properties at different execution points. More details are described in Section 5.2.1. We run the test 100 times as doing so represents a good trade off between obtaining logs for both passing and failing executions, and the time spent on test runs. Note that these 100 runs are done offline—at a later time instead of on CloudBuild machines as developers are making code changes and waiting for test results. Our runs are offline because running the test 100 times under instrumentation can be expensive, and we do not want to increase the build times for the developers.

RootFinder. As tests may exercise many methods during their execution, the logs produced from the previous step can be rather large. To help developers comprehend these logs, we present a tool called RootFinder that can automatically analyze these logs to highlight the differences between passing and failing executions. These differences can be used to help provide insights into why a test is flaky. More details are described in Section 5.2.2.

5.2.1 Torch Instrumentation

Torch [91, 125] is an extensible instrumentation framework for .NET binaries. Torch takes a .NET binary and a set of target APIs, and instruments each target API call in the binary. What API calls and how the API calls are instrumented depends on what Torch *instrumentation plugin* is used. For instance, for profiling one may instrument a binary to track latencies of certain APIs. By default, Torch comes with plugins for profiling, logging,

```
WebClient client = new WebClient();
String data = client.DownloadString(url);
```

(a) Original code

```
WebClient client = new WebClient();
TorchInfo ti = Torch.GetInstrumentationInfo();
String data = Torch_DownloadString(ti, client, url);
```

(b) Instrumented code

```
public static String Torch_DownloadString(TorchInfo ti, WebClient client, String url) {
  String returnValue = null;
  var context = Torch.OnStart(ti, client, url);
  try {
    returnValue = client.DownloadString(url);
  } catch (Exception exception) {
    Torch.OnException(exception, context);
    throw exception;
  } finally {
    Torch.OnEnd(returnValue, context);
  }
  return returnValue;
}
```

(c) Proxy method

Figure 5.2: Torch instrumentation example.

fault injection, concurrency testing, thread schedule fuzzing, etc. One can extend these plugins or write new plugins to suit one's instrumentation goals.

During instrumentation, Torch replaces each API call with an automatically generated proxy call, as shown in Figure 5.2. Note that Torch does not instrument the implementation of a target API; only the call to the API is instrumented. The proxy generated by Torch calls the original API; in addition, as shown in Figure 5.2(c), it calls three Torch callbacks—(1) `OnStart`, called immediately before calling the original API, (2) `OnEnd`, called immediately after the original API returns, and (3) `OnException`, called when the original API throws an exception. `OnStart` returns a context that is passed to `OnEnd` and `OnException`; the context provides information about the API call that invoked the Torch callbacks.

For identifying the root causes of flaky tests, we use Torch's logging plugin to track and log various runtime properties. We find that some APIs will behave differently in passing and failing executions, and analyzing the differences will provide us insights on the root causes of flakiness. Because such APIs are not known beforehand, we opt for logging information for *all* API calls. Specifically, we log the following properties for all API calls:

1. Call information, including signature of the API, and its caller API (the API calling

the instrumented API). We also track the location of the API call in the binary and source code.

2. Timestamp at each call. Timestamps at `OnStart` and `OnEnd` give the latency of the API call.

3. Return value at `OnEnd` and exception at `OnException`, if any.

4. A unique ID of the receiver object of the API. This information helps identify APIs that are operating on the same object and can help identify root causes such as potential concurrency issues.

5. IDs of the process and thread executing the API.

6. ID of the parent thread, i.e., the thread that spawned the thread executing the API. The information is important to understand dependencies of different threads and their activities [125].

It is important for the instrumentation to have a small runtime overhead. Excessive overhead can change runtime behavior and remove existing flakiness or introduce new flakiness. The overhead comes from two different sources. First, computing some of the runtime properties can be expensive. For example, finding signatures of an API and its object type through reflection, or finding the parent API through stack trace can be expensive. Torch avoids this cost by computing these static properties during instrumentation and passing them to the `OnStart` callback as static parameters (as a `TorchInfo` object in Line 3 of Figure 5.2b). Second, due to our collection of runtime information for all APIs, the size of the logs our framework generates can be prohibitively large. To avoid the large space usage, Torch compresses the logs in memory and asynchronously writes them to disk[1].

5.2.2 RootFinder

We develop a simple tool called RootFinder to parse Torch logs of passing and failing executions to identify potential root causes of certain categories of flaky tests. At a high level, RootFinder takes as input a method name that is likely to be the cause of the flakiness, Torch logs of passing runs, and Torch logs of failing runs. Examples of input method names that may be of interest include methods that return non-deterministic values such as

[1]We also experimented with more lightweight Event Tracing for Windows (ETW) logging [50]; however, at a high logging event rate, ETW may skip logging randomly chosen events. We observed a high loss rate, and hence did not have Torch use ETW for logging.

`System.Random.Next` (returns a non-negative random integer) or `System.DateTime.Now` (returns a DateTime object representing the current date and time). By default, our framework uses RootFinder with a predefined set of non-deterministic method calls, but developers can also add or remove method calls as needed.

RootFinder works in two steps. In the first step, RootFinder processes each log file independently and evaluates a set of *predicates* at each line of the log file. The predicates, similar to the ones used in statistical debugging [118], determine if the behavior of the callee method in a log line is "interesting" (several example predicates will be given shortly). The outputs of the predicates are written to a *predicate file*. Each line in the predicate file contains the following information about a predicate: (1) the *epoch* of the predicate, (2) the name of the predicate, and (3) the value of the predicate at its epoch. We currently consider predicates that are *local* to specific code locations, and therefore use epochs that can identify partial orders of predicates evaluated at the same code location. More specifically, the epoch is given by a concatenation of the unique code location of the method call[2], current thread ID, and a monotonically increasing sequence number that is incremented every time the method is called at the current location. For instance, in Figure 5.4, the method `Random.Next()` at unique location `9` is called multiple times (e.g., perhaps the line is in a loop or is called by multiple threads)—once in log line `2` and again in log line `5`. Assuming that both calls are executed in the same thread with ID 10, the first call has the epoch `9:10:1`, the second call has the epoch `9:10:2`, and so on. Partial orders of the epochs can be derived from their IDs along with the threads' parent-child relationship, which Torch dynamically tracks and logs.

Predicates: A predicate evaluates the state of the method call at the current epoch. RootFinder currently implements the following boolean predicates:

- *Relative:* This predicate is true if the return value of a specific line number and thread ID pair is always the same. This predicate is useful to identify if a method is always returning the same value in repeated calls.
- *Absolute:* This predicate is true if the return value of the current epoch matches a given value. This predicate is useful to check if a method returns an error value (e.g., `null` or an error code).
- *Exception:* This predicate is true if the method throws an exception.

[2] *Unique code location* uniquely identifies the location of a method call in the code. An example is the name of the program source/binary file plus the line number/binary offset of the method call within the file. For simplicity we use `Source#` as the unique location.

- *Order:* This predicate is true if an ordering of method calls matches a given list of methods and optionally, whether a specified amount of time occurred between the methods. This predicate is useful to identify thread interleavings.
- *Slow:* This predicate is true if the method call takes more than a specified amount of time. (The threshold can be determined based on domain knowledge of the called method, or by analyzing latencies of passing test runs.)
- *Fast:* This predicate is true if the method call takes less than a specified amount of time.

After the predicate files are generated, RootFinder compares all predicate files (from passing and failing runs) to identify ones that are true/false in all passing executions, but are the contrary in all failing executions. Intuitively, these predicates are strongly correlated to test failures and hence are useful to understand the underlying root cause of failures. Specifically, RootFinder labels each predicate in the predicate files with one of the following categories:

(1) Inconsistent-in-passing: Such a predicate either appears in only a subset of all passing test runs or appears in all passing runs but with both true and false values. The log line corresponding to such a predicate is likely irrelevant as to why a test is flaky. This is because whether the predicate was true or false did not affect the outcome of the test runs (i.e., they always passed).

(2) Inconsistent-in-failing: Such a predicate either appears in only a subset of all failing test runs or appears in all failing runs but with both true and false values. As is the case of the previous category, this predicate is also likely irrelevant as to why a test is flaky.

(3) Consistent-and-matching: Such a predicate appears in all passing and failing runs and with the same value. This predicate is also likely irrelevant as to why the test is flaky as it did not affect the final outcome of the test runs.

(4) Consistent-but-different: Such a predicate either (I) appears only in passing or only in failing runs, or (II) is true in all passing runs but false in all failing runs (or vice versa). (I) indicates that executions of a passing and a failing run diverge before the epoch where the predicate was evaluated (which is why the predicate appears in one set and not the other), while (II) indicates how a method consistently behaves differently in the passing and failing runs. This predicate is *highly likely* to explain why a test is flaky because it precisely shows how passing and failing test runs differ.

By default, the predicates outputted by RootFinder are sorted so that the predicates that are most likely to explain why a test is flaky are shown first (i.e., Consistent-but-different

```
class TestAlertTest {
  void TestUnhandledItemsWithFilters() {
    TestAlert ta1 = CreateTestAlert();
    TestAlert ta2 = CreateTestAlert();
    ...
    Assert.AreNotEquals(ta1.TestID, ta2.TestID);
  }
  TestAlert CreateTestAlert() {
    int id = new Random().Next();
    ...
    return new TestAlert(TestID = id, ...);
  }
}
```

Figure 5.3: Test method from a Microsoft product's test suite.

predicates). Once the categories are sorted, RootFinder then sorts the predicates within each category so that the predicates with the lowest log line numbers are outputted before the ones with higher numbers. As presented in Section 5.3.2, our case studies find that sorting predicates as described enables RootFinder to effectively output useful predicates.

The predicates outputted by RootFinder can aid the debugging efforts of nine out of ten categories of flaky tests mentioned in prior work [126]. More specifically, for categories such as Network, Time, IO, Randomness, Floating Point Operations, Test Order Dependency, and Unordered Collections, RootFinder can directly compare the return value of failing and passing Torch logs to identify predicates that are highly likely to explain why the test is flaky. For the Async Wait and Concurrency categories, our framework currently relies on Torch's ability to first fuzz delays, and then for RootFinder to identify latency-related predicates, such as Fast and Slow, to help developers root cause those categories. For the remaining category, Resource Leak, our framework can eventually be extended to include memory leak detection tools [135, 210], but we do not find many flaky tests in this category. More details about the categories of flaky tests at Microsoft are in Section 8.3.5.

An Example. Figure 5.3 shows the simplified version of a test from a proprietary product at Microsoft. `TestUnhandledItemsWithFilters` is flaky because `new Random().Next()` may return the same value if the two calls are invoked close together. Specifically, if a `Random` object is instantiated without a seed, the object will use the current system time as the default seed. Therefore, two `Random` objects instantiated without a seed and close together will be initialized with the same seed and return the same sequences of random numbers.

Figures 5.4 and 5.5 show a fragment of Torch logs from passing test runs and from failing test runs, respectively. To keep this example simple, we omit information from Torch that is irrelevant to this example, such as ID of thread executing API and ID of parent thread. As

Line #	Source #	Seq #	Caller name	Callee name	Timestamp	Return Value
1	3	1	TestAlertTest.TUIWF()	TestAlertTest.CreateTestAlert()	1	TestAlert#1
2	9	1	TestAlertTest.CreateTestAlert()	Random.Next()	2	14
3	11	1	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	2	null
4	4	1	TestAlertTest. TUIWF()	TestAlertTest.CreateTestAlert()	3	TestAlert#2
5	9	2	TestAlertTest.CreateTestAlert()	Random.Next()	3	16
6	11	2	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	3	null
...	...	...	...	...	...	...
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	20	null
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	17	null
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	28	null

Figure 5.4: Torch logs for passing test runs of the test in Figure 5.3. `TUIWF` is `TestUnhandledItemsWithFilters`. Thread ID is the same for all lines.

Line #	Source #	Seq #	Caller name	Callee name	Timestamp	Return Value
1	3	1	TestAlertTest.TUIWF()	TestAlertTest.CreateTestAlert()	1	TestAlert#1
2	9	1	TestAlertTest.CreateTestAlert()	Random.Next()	2	21
3	11	1	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	2	null
4	4	1	TestAlertTest. TUIWF()	TestAlertTest.CreateTestAlert()	2	TestAlert#2
5	9	2	TestAlertTest.CreateTestAlert()	Random.Next()	2	21
6	11	2	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	2	null
...	...	...	...	...	...	...
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	18	N/A
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	24	N/A
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	16	N/A

Figure 5.5: Torch logs for failing test runs of the test in Figure 5.3. `TUIWF` is `TestUnhandledItemsWithFilters`. Thread ID is the same for all lines.

shown on Line 2 and Line 5 of Figure 5.5, the `StartTime` values of the `Random.Next()` calls are the same in each failing log, therefore the return values for both calls to `Random.Next()` are the same within each failing log (e.g., `21`, `17`, and `5`). In the passing logs, such as the ones depicted in Figure 5.4, we can see that on Line 2 and Line 5, the `StartTime` values are different and consequently, the return values of `Random.Next()` are also different. Recall that the assertion on Line 8 of Figure 5.3 passes if the return values of `Random.Next()` are different and fails otherwise.

RootFinder can help narrow down the the root cause of `TestUnhandledItemsWithFilters`. In step 1 of RootFinder's processing, RootFinder converts the passing logs in Figure 5.4 into predicate files containing the predicate `(9:2, Relative, False)`. This predicate means that the method `Random.Next()` in `Source#` 9 and `Seq#` 2 returns a value that is different from the return value of the immediate previous call of the same method at the same `Source#`. Similarly, it converts the failing logs in Figure 5.5 into predicate files containing the predicate `(9:2, Relative, True)`. In step 2, RootFinder compares the predicates across passing and

failing runs, and identifies the `(9:2, Relative)` predicate as *Consistent-but-different*. This predicate quickly points to a root cause (or a symptom that is strongly correlated to the root cause) of the flakiness, as well as its code location (e.g., `9:2`). When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, 2 seconds to run and outputted 408 predicates in total. The predicate containing the root cause was ranked first.

5.3 CASE STUDIES

Table 5.1: Characteristics of the specific flaky test examples in Section 5.3.2 and of all 44 flaky tests in our dataset.

	Duration / test	**% of failed executions/test**	**# of method calls/test**	**# of unique method calls/test**	**# of threads / test**	**# of objects / test**
Specific examples in Section 5.3.2						
Time	1s	29%	4.7k	463	3	1151
Concurrency	10s	1%	18k	882	8	8200
Async Wait	2s	1%	0.2k	90	4	62
Resource Leak	4s	1%	0.6k	218	6	246
All 44 flaky tests						
Median	5s	6%	2.5k	248	3	637
Average	45s	28%	335k	335	5	55418

We next present the results of applying our framework on large projects that use CloudBuild. To ensure that our results are not biased due to a single project, we collect all distinct flaky tests recorded during a day in our production environment. Among these flaky tests, we identified the tests that are compatible with the Torch instrumentation framework. More specifically, CloudBuild supports unit tests written in both managed (such as C#) and unmanaged (such as C++) code [129]. Also, CloudBuild supports tests written for various test frameworks such as MSTest [209], NUnit [153], and xUnit [225]. Our current implementation of the instrumentation framework supports only unsigned (binaries that do not include digital signatures) and managed code, and is also tailored for tests that run using the MSTest framework.

Overall, we collected **315** flaky tests that matched the described criteria. Our collected flaky tests belong to different projects that provide services for both internal and external customers, and also fall into different categories, such as database, networking, and security. Among these flaky tests, we were only able to reproduce flakiness, i.e., produce logs for both passing and failing executions in 100 runs for **44** tests. Our findings here that only 44 out of 315 flaky tests are reproducible with Torch instrumentation suggests that improvements to reproducing flakiness, particularly with instrumentation, can be highly impactful.

5.3.1 Study Dataset

We use a dataset consisting of 44 flaky tests. These tests belong to 22 software projects from 18 Microsoft internal/external products and services. For each test, the dataset contains 100 execution traces, some of which are from failed executions. Each trace file consists of a sequence of records containing various runtime information about all executed methods as described in Section 5.2.1.

Table 5.1 shows some characteristics about the tests and traces. The characteristics show the overall complexities of the tests. The average run duration of the tests is nontrivial (45s), even though they are all unit tests. Each test runs a large number of methods (335k total methods/test and 335 unique methods/test), mostly because a tested component often depends on many underlying components. Furthermore, 80% of the tests use more than one thread and on average, each test runs on 5 threads and operates on 55418 objects. In short, these tests produce massive runtime logs, which can be extremely challenging to analyze.

Each runtime log in our dataset contains a wealth of information. For example, each log contains all of the methods executed by the test, and the latencies, return values, thread IDs, etc. of the executed methods. We make our dataset of these runtime logs publicly available [168] to help researchers build better solutions to the flaky-test problem and understand the runtime behavior of flaky tests in a production system. Our dataset is anonymized so that sensitive strings (such as method names containing Microsoft product names) are replaced with hash values.

5.3.2 Case Studies of Finding Root Causes

In this section, we provide in-depth examples of flaky tests and how RootFinder assisted developers with debugging these particular flaky tests. The remainder of this section presents three examples of flaky tests that our framework can find root causes for and one example that our framework cannot. All examples are anonymized and simplified as needed.

Time. We find some tests to be flaky due to improper use of APIs dealing with time. These flaky tests rely on the system time, which introduces non-deterministic failures, e.g., a test may fail when run in different time zones.

Figure 5.6 shows a simplified version of a test case, which ensures that a service and its replica return the same response to a particular message. Specifically, the assertion on Line 6 occasionally fails. The failures are because calls to `Service` and `ServiceReplica`'s `SendAndGetResponse` may or may not use the same timestamps. If the invocations of `Send-`

```
1 [TestMethod]
2 public void TestReplicaService() {
3   ... // initialize payload
4   byte[] response = Service.SendAndGetResponse(payload);
5   byte[] replicaResponse = ServiceReplica.SendAndGetResponse(payload);
6   Assert.AreEqual(response, replicaResponse);
7 }
8 public class Service : NetworkService {
9   ... // initialize base
10  public byte[] SendAndGetResponse(Request req) {
11    ...
12    DateTime currentTime = DateTime.UtcNow;
13    Message message = new Message(req, currentTime);
14    return base.SendAndGet(message.Serialize());
15  }
16 }
```

Figure 5.6: A test that is flaky due to getting the system time.

`AndGetResponse` on Lines 4 and 5 happen within a short window of time, the timestamps produced by `DateTime.UtcNow` on Line 12 can be the same due to the limited granularity of the system timer. The granularity is seconds by default. If the timestamps are the same, then the test passes; otherwise, it fails. We find many flaky tests at Microsoft exhibiting similar behavior. This example fails 29% of the time in our experiments, but we also find other tests exhibiting a similar root cause failing up to 88% of the time.

A useful predicate for this example should indicate that the timestamp in `SendAndGet-Response` (Line 12) when invoked by Line 5 is always the same as the timestamp when invoked by Line 4 in the passing logs, but the timestamps are always different in the failing logs. When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, 11 seconds to run and outputted 1163 predicates in total. The useful predicate was ranked at 81. In practice, when developers used RootFinder on this example, the developers were able to input suspicious method names to quickly find the useful predicate in a few minutes.

Concurrency. A flaky test's root cause is concurrency when the test can pass or fail due to different threads interacting in a non-deterministic manner (e.g., data races, deadlocks).

Figure 5.7 shows an example of a test that is flaky due to concurrency issues. The `TestDirtyResource` method tests that a manager (created in Line 4) of a cluster properly recycles used resources. Once the manager is created, it is setup by adding a machine to it, that is in use, or "dirty" (Lines 5–6), along with more setup code (omitted for brevity). The test then repeatedly creates and sends requests to the cluster to execute jobs (Lines 9–12) and makes sure that the response obtained on Line 12 is correct (Lines 13–16). Finally, the

```
1  [TestMethod]
2  public async Task TestDirtyResource() {
3    ...
4    using (var emptyPoolManager = CreatePoolManager(...)) {
5      Resource pm = ResourceUtils.CreatePhysicalMachine(...);
6      await emptyPoolManager.AddOrUpdateResourceAsync(pm, HeartbeatStatus.InUse);
7      ...
8      for (int i = 0; i < 5; i++) {
9        var sessionId = Guid.NewGuid().ToString();
10       var request = new ResourceAllocateRequest(pm);
11       await emptyPoolManager.PreAllocateResourcesAsync(...);
12       var response = (await QueryAllocate(...)).FirstOrDefault();
13       Assert.IsNotNull(response);
14       Assert.AreEqual(request.Id, response.RequestId);
15       Assert.IsNotNull(response.ResourceId);
16       Assert.AreEqual(pm.ResourceId, response.ResourceId);
17       await emptyPoolManager.ReleaseResourcesAsync(response.ResourceId);
18       await emptyPoolManager.HeartbeatResourceAsync(pm, HeartbeatStatus.Ready);
19       await emptyPoolManager.ProcessResourcesHeartbeats(CancellationToken.None);
20     }
21   }
22 }
```

Figure 5.7: A test that is flaky due to concurrency.

resources used by the cluster to process the request are released (Line 17), and the newly freed resource heartbeats its status to the manager (Line 18), and the manager processes it (Line 19). We observed in our experiments that the assertion on Line 13 fails occasionally.

Upon investigating the failure, we find that the creation of the resource manager (Line 4) also starts a background task that periodically marks resources as unavailable in case T milliseconds (ms) has elapsed since the last heartbeat was processed. This test fails when this background task runs T ms after another thread has processed Line 19 but before Line 12, because the background task would mark the cluster resource as unavailable, which would then cause the subsequent resource allocation request made using `QueryAllocate` on Line 12 to return `null`. The null value will then cause the assertion on Line 13 to fail. This example demonstrates a subtle flakiness condition that only manifests in a particular thread interleaving, and moreover, only if the interleaving follows very precise timing constraints.

A useful predicate for this example should indicate that the background task from Line 4 always ran after Line 19 and it always runs T ms or longer after Line 19. When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, 126 seconds to run and outputted 127187 predicates total. The useful predicate was ranked at 3231. RootFinder did not rank the root cause predicate highly, because this flaky test's root cause is not only due to thread interleavings, but also the timing of the interleavings.

```
1 DatabaseProvider ssp;
2 String currentDirectory = ...;
3 [TestMethod]
4 public void ResourceAllocation() {
5   ssp = new DatabaseProvider(currentDirectory);
6   ...
7 }
8 [TestCleanup]
9 public void TestCleanup() {
10  ClearConnections();
11  if (File.Exists(dbPath)) {
12    File.Delete(dbPath);
13  }
14  ...
15 }
```

Figure 5.8: A test that is flaky due to resource leaks.

Async Wait. Tests are flaky due to the Async-Wait issues when the test makes an asynchronous call and does not properly wait for the result of the call to become available before using it. Depending on whether the asynchronous call was able to finish at the right time or not, such flaky test may pass or fail. Section 1.2.2 presents an example of an Async-Wait flaky test. A useful predicate for this example should indicate that the task on Line 4 always took longer in the failing runs. When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, one second to run and outputted 868 predicates in total. The useful predicate was ranked at 17.

Resource Leak. A flaky test's root cause is Resource Leak when the test passes or fails because the application does not properly acquire or release its resources, e.g., locks on files.

Figure 5.8 shows an example of a Resource Leak flaky test. `ResourceAllocation` checks whether the necessary resources are properly allocated for an application. The application internally uses a third-party database to store some information. The `TestCleanup` method is executed after every test to delete the database, so that it may be re-initialized before the next test. Although Line 12 tries to close the connection to the database, the third-party library requires the garbage collector to run prior to this step to release the file handle to the database file. If the garbage collector does not run first, then the file resource is held, causing the subsequent attempts to delete the file on Line 12 to throw an exception. Since our framework relies on Torch to capture information relevant to the root cause of the flaky test, our framework is currently unable to assist developers for Resource Leak flaky tests such as the one in this example.

5.3.3 Applying RootFinder on to Case Studies

When we apply RootFinder on the five examples described in Section 5.2.2 and Section 5.3.2, we find that the root causes are summarized as predicates in the output for four of the five examples, especially when developers provided domain knowledge to RootFinder. These predicates can assist developers in identifying what values are the same or not in the passing and failing executions, and provide them the code location that produced these values. When we present the findings of RootFinder to developers or use it ourselves, they/we are able to more quickly reproduce the failures of the flaky test.

5.4 THREATS TO VALIDITY

Internal Validity. Our threats to internal validity are concerned with our study procedure's validity. RootFinder and our framework can contain faults that impact our results and lessons learned. We attempt to mitigate this threat by having thorough code reviews and testing of RootFinder and our framework. Furthermore, we rely on various other tools in our framework, such as Torch. These tools could have faults as well and such faults could have also affected our results. To mitigate this threat, some of the logs produced by Torch and the root causes produced by RootFinder are manually analyzed and confirmed by at least two of the authors.

External Validity. Our threats to external validity are concerned with all threats unrelated to our study procedure. Our lessons learned may not apply to projects other than the ones in our study. We attempt to mitigate this threat by including a diverse range of projects in our study. Our projects are used by both internal and external customers of Microsoft, and also fall into different categories such as database, networking, security, and core services. Flaky tests are, by definition, tests that non-deterministically pass or fail with the same version of code. Due to the non-deterministic nature of these tests, the flaky tests from our dataset that we are able and unable to produce Torch logs for may not be the same if the experiment was to be repeated.

5.5 OPEN RESEARCH CHALLENGES

The results reported in this chapter are just scratching the surface of identifying the root causes of flaky tests. Here are some open questions that are left to be explored.

How to evaluate results. A major challenge is to determine the "*ground truth*" of why a test is flaky. Ultimately, this requires developers to look at each flaky test, examine the suggested candidate root causes, and then decide which root cause(s) is/are the most likely. For a large number of flaky tests, this evaluation is *expensive*. Moreover, results may vary depending on the developer's expertise and familiarity with the code being tested and with the tests being performed. How to prevent *biased evaluations* due to diversity in developer expertise is another non-trivial challenge.

What information to log. Earlier in the paper, we showed one way to log execution traces. But there are many other options. Should all method calls and returns be logged? In all the software components or only in some? If some, which ones and why? Should function input and output values be logged as well? Should intra-procedural execution information (e.g., which code instructions, blocks or branches were executed) also be recorded for a fine-grained analysis?

Logging versus analysis tradeoffs. The more data is being logged, the more computationally expensive it is to store and process all the data. The analysis itself becomes more complicated: non-essential differences may creep in if too much information is recorded, which makes it harder to identify meaningful differences. On the other hand, recording too little information may omit key events explaining the source of the test flakiness. How to strike the right balance between logging and analysis effectiveness is another key challenge in this space.

Fixed versus variable logging. Another dimension to the problem is whether the level of detail used for logging should be the same for all applications and tests, or whether it should be adjusted, automatically over time in an "iterative-refinement process", or using user-feedback. If the logging level can be adjusted, should it start *lazily*, at a high-level and be refined until meaningful differences are detected? Or, should logging proceed bottom-up, *eagerly* logging many events, then identifying irrelevant details (data noise) and eliminating those until meaningful differences are found?

Logging versus reproducibility. The more data one logs, the more intrusive the runtime instrumentation can be. As discussed earlier in this chapter, reproducing flakiness is difficult, and can become even more difficult when significant runtime slowdowns are introduced by expensive logging activities.

5.6 SUMMARY

Flaky tests are a prevalent issue plaguing large software development projects. Simply ignoring flaky tests is a dangerous practice because the failures might mask severe software

defects. To help software developers, we need better tools and processes for dealing with flaky tests. This chapter presented an effort at Microsoft to deal with flaky tests. Specifically, we discussed our work on a framework and a new tool, RootFinder, for root-causing flaky tests. We hope that the framework, tool, and case studies presented in this chapter will encourage more research on this important problem.

CHAPTER 6: [CHARACTERIZATING] UNDERSTANDING REPRODUCIBILITY AND CHARACTERISTICS OF FLAKY TESTS THROUGH TEST RERUNS IN JAVA PROJECTS

This chapter presents our in-depth study of non-deterministic tests and provides actionable guidelines to deal with flaky tests. We believe that two key challenges have limited the amount of in-depth work on non-deterministic (NOD) tests (defined in Section 1.2). The first challenge is the machine cost for rerunning tests. Many NOD tests may fail rather infrequently (e.g., once in 4000 test runs, as we observe from our experiments for this chapter) or only under specific circumstances, so it takes substantial time and reruns to observe even one failure, let alone a few failures to study when and how they occur. The second challenge is the human cost for debugging NOD tests. In contrast to order-dependent (OD) tests that fail deterministically and could be somewhat easier to reproduce and debug, NOD tests fail non-deterministically, potentially infrequently, and can take a lot of time to debug, especially for researchers unfamiliar with some open-source code that has flaky tests.

In this chapter, we perform a study that is organized around four main research questions (RQs) that aim to improve our understanding of how to rerun, detect, debug, and prioritize flaky tests. By answering these questions, we aim to provide actionable guidelines and practical suggestions for developers and researchers. To perform our study, we (re)run tests using the default test runner configured by the developers, following what developers typically do in their development practice. In other words, we do *not* use any research tools [59, 73, 84, 108, 184, 237] or emerging approaches [160, 199] that aim to detect flaky tests. While our procedure does miss some flaky tests, it (1) gives a more *realistic assessment* of the impact that flaky tests have on regression testing; (2) exposes flaky tests that can be particularly difficult to debug, because they fail rarely even under the exact same build configuration; and (3) avoids false alarms that can be produced by some of the research tools (e.g., iFixFlakies [186] reports on some false alarms reported by our work in Chapter 2).

The remainder of this chapter is organized as follows. Section 6.1 presents our research questions, and Section 6.2 presents our experimental methodology. Section 6.3 then presents our results, and Section 6.4 presents our case studies of some flaky tests. Finally, Section 6.5 presents the threats to validity, and Section 6.6 concludes this chapter.

6.1 RESEARCH QUESTIONS

To increase the understanding of the reproducibility and characteristics of flaky tests, we consider four main questions.

RQ1: What is the failure rate (defined in Section 1.2) and maximal burst length (defined in Section 1.2.3) of flaky tests across all test-suite runs?
Why it matters: A common way to separate test failures into regression failures and flaky failures is through test reruns. For example, Google [137] and Microsoft [107] report how they rerun tests after a failure to check whether the test's result would change to pass (indicating a flaky failure instead of a regression failure). The number of reruns is typically chosen ad-hoc, e.g., ten times at Google [137] and once at Microsoft as described in Section 5.1. This RQ aims to provide empirical evidence for how many reruns can suffice to separate regression failures and flaky failures.
RQ2: How do failure rates and maximal burst lengths differ across test class orders (TCOs), and how many tests are NDOI (defined in Section 1.2.2) and NDOD (defined in Section 1.2.3)?
Why it matters: Developers have a limited budget of test runs to detect NOD tests. Is it better for those runs to be used through (1) many unique test orders with fewer runs each, or (2) fewer orders with many runs each? If NDOD tests are more prevalent than NDOI tests, then (1) can be better, otherwise (2) can be better. This RQ aims to provide empirical evidence on the prevalence of these two subcategories, so that developers can better detect NOD tests.
RQ3: How likely can NOD-test failures from running the test suite be reproduced by running the NOD test in isolation?
Why it matters: When developers encounter a test failure by running a test suite, a common next step is to try reproducing a test failure by running the test in isolation. For a regression failure, the test continues to deterministically fail in isolation. For OD tests, the test also deterministically passes or fails in isolation [186], and it deterministically fails when run in the same order that produced the test failure. However, for NOD tests, it is unclear how likely test runs in isolation can reproduce the test failure from running the entire test suite, e.g., the test may fail more or less often when run in isolation than in the test suite. This RQ aims to show empirical evidence to clarify this issue.
RQ4: How do failure rates of individual flaky tests relate to failure rates of test-suite runs?
Why it matters: Whether developers can merge their recent changes to a project typically depends on whether the entire test suite passes or fails with their recent changes. RQ1 and RQ2 study the failure rate of each individual flaky test, while RQ4 aims to show how often a test-suite run has at least one failing flaky test. RQ4 is important because it provides empirical evidence on how often (1) developers encounter a test-suite run failure due to flaky tests and (2) failures of different flaky tests are symptoms of the same underlying problem, which developers can use to prioritize their debugging efforts.

6.2 EXPERIMENTAL METHODOLOGY

6.2.1 Modules Used in Our Study

To investigate our research questions, we use open-source projects from the iDFlakies dataset [85]. We obtained this dataset from our prior work described in Chapter 2. The prior work used the iDFlakies tool, which detects flaky tests by perturbing the execution order of test methods (not just test classes) across repeated test-suite runs and marking tests that both pass and fail in various runs as flaky. Due to the perturbing of test order, many tests that iDFlakies detected are OD tests, i.e., they pass or fail deterministically depending on which other tests have (not) run before them in the test suite.

The iDFlakies dataset [85] consists of Java projects that use the Maven build system [131]. Maven organizes projects around modules, so we present our analysis in terms of modules. We do not use all modules (111) from the iDFlakies dataset for our study, because we focus on NOD tests, and iDFlakies detected NOD tests in only 62 modules (and only OD tests in the other 49 modules). We focus on NOD tests because many past studies of flaky tests focused on OD tests [59, 108, 186, 237]. We find that only 48 of the 62 modules could be compiled "out-of-the-box" (i.e., Maven could not download some dependencies for the other 14 modules), and for 44 of the 48 modules we could run the test suite 4000 times (i.e., the test suite deadlocks for the other 4 modules) and easily control the order in which Maven Surefire [132] runs test classes (i.e., the module uses Maven Surefire version 2.7 or higher).

6.2.2 Configurations in Our Study

To obtain the flaky tests for our study, we run tests in two different configurations:

- *Test Suite (TSO)*: run the entire test suite using `mvn test`. The order of the tests may differ. We call one run of the test suite in any order a *test-suite run (TSR)*.
- *Isolation (ISO)*: run each individual test in its own JVM using the command `mvn test -Dtest=TestClass#testMethod`.

We run TSO and ISO 4000 times each to balance the chance to detect flaky tests and the machine time used for our experiments. In total, our experiments used 2148 hours (∼90 days) of CPU time. At the time that our experiments were done, they consisted of the largest number of runs in any published study of flaky tests. We ran our experiments on Microsoft Azure [138] using the `Standard_D11_v2` virtual machines with 2 CPUs, 14 GB of RAM, and 100 GB of hard-disk space each.

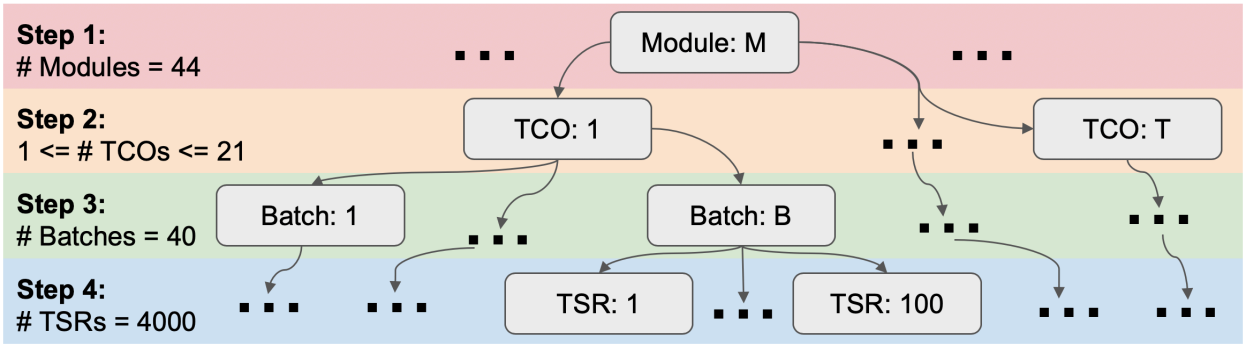


Figure 6.1: Overview of running test suites to obtain flaky tests in our study.

6.2.3 RQ1 and RQ2: Failure Rate and Burst Length

To answer RQ1 and RQ2, we obtain and compare failure rates and maximal burst lengths of flaky tests across all TCOs (RQ1) and across different TCOs (RQ2). Of the 44 modules from Section 6.2.1, we find at least one flaky test in 26 modules when they are run in TSO. Figure 6.1 shows an overview for how we run the test suites to collect our dataset and Table 6.1 shows the statistics of these 26 modules.

Test-Class Orders (TCOs). For each module, we run Maven Surefire (i.e., `mvn test`) with (1) no changes to the build configuration `pom.xml` files (except M26's `pom.xml` to make its tests run sequentially)—these runs commonly yield different TCOs needed to answer RQ2; and (2) a change to run all test classes in a sorted order (reverse alphabetical of their class names)—to ensure we get many runs of at least one order. We run each of (1) and (2) for 20 *batches*, where each batch runs exactly 100 test-suite runs (*TSRs*) in a fresh virtual machine (VM). Overall, we have exactly 40 batches and consequently, 4000 TSRs per module.

For (1), Surefire determines a TCO based on file system specific properties, and because we run each batch in its own VM, the TCOs across batches likely differ, resulting in up to 20 TCOs from (1) and 1 TCO from (2). Consequently, the number of TCOs ranges from 1 to 21, each of which is run between 1 and 40 batches. For modules with 3 or fewer test classes, we cannot obtain 21 TCOs because the maximum number is 6 (i.e., 3!). Even for modules with 4 or more test classes, we can still obtain fewer than 21 different TCOs because Surefire can return the same TCO across different VMs (which happened for two modules: M9 and M12). Table 6.1 shows the exact number of TCOs[1] that we obtained per module.

We design our experiment to use multiple batches with 100 TSRs per batch rather than

[1]The test-method order may differ even when the TCO is the same. Specifically, for modules using JUnit versions 4.11 or higher, the test-method order would be the same for any test class regardless of the TCO. From the modules in Table 6.1, only M12 and M13 use a version of JUnit that is lower than 4.11 and exhibit different test-method orders for at least one test class.

Table 6.1: Statistics for the modules where we detected some flaky test by first running entire test suites (TSO) and then running the flaky tests detected by TSO in isolation (ISO); "=" indicates same value as the cell to the left.

		# test		#	# flaky tests		TSO failure rate [%]				ISO rate [%]	
MID	**Project slug - Module**	**methods**	**classes**	**TCOs**	**TSO**	**ISO**	**TSR**	**min**	**max**	**sum**	**min**	**max**
M1	alibaba/fastjson	4464	2079	21	6	0	7.7	0.1	5.0	15.2	n/a	n/a
M2	apache/incubator-dubbo - m1	14	7	21	3	1	0.9	0.1	0.5	0.9	0.3	=
M3	- m2	66	15	21	9	0	10.6	0.1	10.0	20.7	n/a	n/a
M4	c2mon/c2mon - m1	125	18	21	1	0	<0.1	=	=	=	n/a	n/a
M5	- m2	10	2	2	2	2	1.6	0.4	1.2	1.6	1.0	1.1
M6	codingchili/excelastic	12	4	12	1	0	11.4	=	=	=	n/a	n/a
M7	davidmoten/rxjava2-extras	390	48	21	3	2	0.2	0.1	0.1	0.2	<0.1	<0.1
M8	elasticjob/elastic-job-lite	502	89	21	1	0	2.5	=	=	=	n/a	n/a
M9	espertechinc/esper	2	2	1	1	1	3.0	=	=	=	3.7	=
M10	feroult/yawp	1	1	1	1	1	1.6	=	=	=	2.4	=
M11	flaxsearch/luwak	202	37	21	2	2	1.0	0.3	0.8	1.0	58.6	68.2
M12	fluent/fluent-logger-java	18	5	17	6	0	1.8	0.1	1.8	5.2	n/a	n/a
M13	javadelight/delight-nashorn-sandbox	79	35	21	3	2	1.4	<0.1	0.7	1.4	4.2	57.1
M14	kagkarlsson/db-scheduler	51	18	21	8	1	0.3	0.1	0.3	1.1	0.9	=
M15	looly/hutool	7	2	2	1	1	0.1	=	=	=	1.5	=
M16	nationalsecurityagency/timely	144	32	21	4	4	2.8	1.3	2.8	9.5	1.3	3.0
M17	oracle/oci-java-sdk	62	8	21	1	1	<0.1	=	=	=	<0.1	=
M18	orbit/orbit	20	8	21	1	1	0.1	=	=	=	0.2	=
M19	OryxProject/oryx	92	19	21	1	1	0.8	=	=	=	1.0	=
M20	spinn3r/noxy	3	1	1	3	0	50.0	49.0	50.0	100.0	n/a	n/a
M21	square/retrofit - m1	80	15	21	5	0	1.4	<0.1	0.5	1.4	n/a	n/a
M22	- m2	297	10	21	1	1	0.1	=	=	=	0.2	=
M23	TooTallNate/Java-WebSocket	145	22	21	32	24	30.1	<0.1	5.1	46.0	<0.1	16.6
M24	wro4j/wro4j	308	64	21	9	3	1.3	0.1	0.7	2.0	0.3	0.4
M25	wso2/carbon-apimgt	2	1	1	1	1	0.1	=	=	=	0.1	=
M26	zalando/riptide	33	12	21	1	1	10.9	=	=	=	39.1	=
Total / Average		**7129**	**2554**	**415**	**107**	**50**	**5.4**	**3.2**	**4.2**	**9.1**	**6.4**	**10.9**

(1) just one batch to run all 4000 TSRs, or (2) 4000 batches, each with just one TSR. Compared to (1), our design helps with cases where a test may deadlock in a batch (i.e., we would only need to rerun one small batch). Compared to (2), our design helps by controlling machine cost and providing control over the number of TSRs for each TCO (which we use to calculate failure rates for RQ2).

To automatically determine whether a NOD flaky test is likely NDOD or NDOI for RQ2, we use (1) the ratio of TCOs that have at least one failing TSR, and (2) a statistical test of whether failure rate differences across TCOs are significant or not. As multiple batches can execute the same TCO, the number of TSRs can differ across TCOs. To exclude effects of these differing numbers, we test whether the *proportions* of test failures among test runs differ significantly across TCOs, specifically using the χ^2 test (implemented by the `prop.test` function in R). We use the resulting p-values of the test and a significance level of 0.05 to determine significance.

Flaky Tests. In total, we detected 107 flaky tests in 26 modules. Compared to iDFlakies, which detected 124 flaky tests in these 26 modules, 64 tests are in common with our 107 tests, 60 tests are detected only by iDFlakies, and 43 tests are detected only by our runs. In 18 modules where we did not detect any flaky test, iDFlakies detected 40 NOD tests.

We detect some more flaky tests than iDFlakies, because the iDFlakies study ran most test suites 100 times, while we run them 4000 times. We detect some fewer flaky tests than iDFlakies because of three reasons: (1) iDFlakies uses a custom Maven plugin that does not respect some configuration options from Surefire, e.g., `exclude` and `runOrder`; (2) iDFlakies does produce false alarms, e.g., it may even run test methods annotated with `@Ignore` [97], which should be skipped; and (3) iDFlakies can randomize not just the test-class order but also the test-method order. These features of iDFlakies have been designed to maximize the number of potentially flaky tests it can detect, while our study aims to more closely understand actual flaky tests in developers' typical test-suite runs.

6.2.4 RQ3: Reproducing TSO Failures in ISO

RQ3 evaluates how likely one can reproduce flaky-test failures observed from TSO runs by running the tests in ISO. We obtain the data for this RQ by running each test detected as flaky in TSO runs for 4000 times in ISO. Specifically, we run each of 107 tests detected from TSO runs in 40 batches with each batch running the test 100 times. We then analyze the number of flaky-test failures one can reproduce in ISO, and how the failure rates and burst lengths of these tests differ between TSO and ISO runs.

6.2.5 RQ4: Effect of Flaky Tests on TSRs

RQ4 considers how often a TSR has at least one failing flaky test. We obtain the data for this RQ from TSO runs. Table 6.1 shows the minimum, maximum, and sum of the failure rates for the flaky tests in each test suite. From these failure rates we can derive the bounds for the *TSR failure rate*, i.e., the ratio of TSRs with at least one flaky-test failure. The *minimum* TSR failure rate is the maximum failure rate across all flaky tests in the test suite—if all flaky tests are dependent, the TSR failure rate equals the maximum of the failure rates. The *maximum* TSR failure rate is the sum of the failure rates for all flaky tests in the test suite (up to 100%)—if all flaky tests are independent, the TSR failure rate equals the sum of the failure rates. We investigate the TSR failure rates observed in our experiments and compare how often these failure rates equal the minimum or maximum potential TSR failure rates.

6.3 RESULTS

We next present the results for our four research questions.

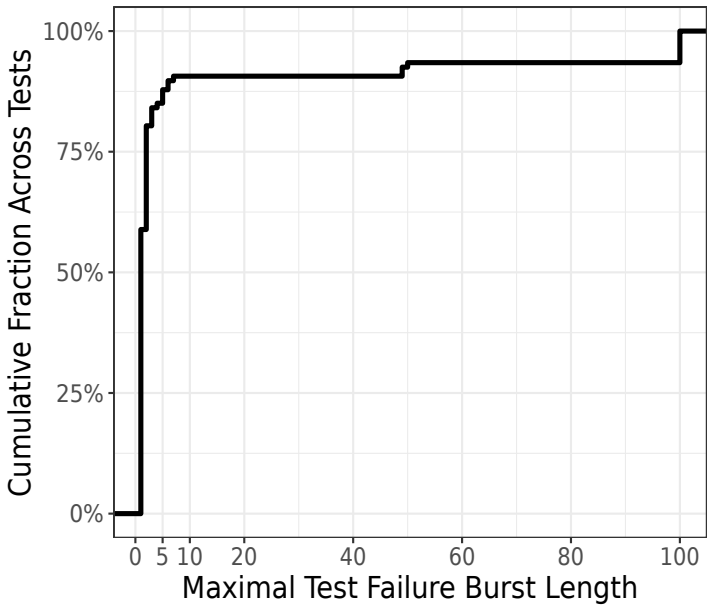


Figure 6.2: Distribution of maximal burst lengths across 107 tests for TSO.

6.3.1 RQ1: Overall Failure Rate and Burst Length

Table 6.1 shows for each module, the number of detected flaky tests (columns TSO and ISO) and the flaky tests' minimum, maximum, and sum *failure rate*—the ratio of runs in which the flaky tests fail. The failure rate indicates (1) how much of a problem the flaky test is for developers, (2) how likely it is that a developer observed it as a flaky test, and (3) how difficult it is to reproduce the flaky-test failure for debugging. For (1), the failure rate would ideally be close to 0, so that the test rarely affects developers. For (2), the failure rate would ideally be close to 50%, giving the same probability to observe both passing and failing runs, which is what dynamic flaky-test detection tools use to classify a test as flaky. For (3), the failure rate would ideally be close to 100%, so that the failures are reproducible for debugging.

For the 107 flaky tests that we detect in TSO, the (arithmetic) mean failure rate is 2.7%, with the minimum of 0.025% and the maximum of 50%. The mean is heavily affected by 7 tests (listed in Table 6.2) with failure rates of $\geq$10%; in contrast, over 85% of the flaky tests have a failure rate lower than the mean, resulting in the overall median below 0.5%. These failure rates indicate that the majority of the flaky tests detected with `mvn test` rarely affect developers; indeed, if the failure rates were rather high, the developers would have probably rewritten or removed the tests. These failure rates also indicate that the flaky tests are difficult to detect and even more difficult to debug without specialized tools.

If test failures are temporally correlated and tend to occur in consecutive reruns with large

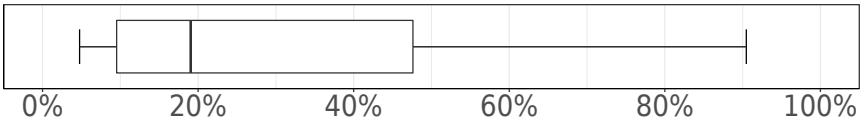


Figure 6.3: Ratios of failing TCOs for 98 tests with more than 2 TCOs.

burst lengths, they appear more deterministic to developers and may mislead them in their conclusions regarding the root cause of the failure. Large burst lengths also hamper flaky test detection, because they require more reruns before a pass result can be observed and the test is marked as flaky. However, consecutive failures of flaky tests can be beneficial for debugging, because failures can be consecutively observed after the first failure.

Figure 6.2 shows the cumulative distribution function (CDF) of *maximal* burst lengths we observed across all test reruns. The number cannot exceed 100, because each batch executed 100 TSRs. We discuss the maximal burst length, rather than the average, to obtain a *worst case estimate* of the negative impact of flaky tests. The CDF reaches a first plateau at 90.6% for a burst length of 7, which means that over 90% of flaky tests in our study failed at most 7 times consecutively. Even a burst length of 5 already has 87.8% of tests. The later increases are for two tests with a maximal burst length of 49, one of 50, and seven OD tests of 100. Thus, the commonly used number of ten reruns by Google [137] to detect flaky tests does not appear well justified: after one failure, 87.8% of tests can be found to pass in 5 reruns, with minor increases at 6 and 7 reruns. The remaining tests require many more than ten reruns.

Guideline: When rerunning tests after failures to check if they are flaky, our results suggest ≤ 5 reruns.

6.3.2 RQ2: Effect of Order on Failure Rate and Burst Length

We next consider how the previous results vary across test-class orders (TCOs) for each test. To analyze this variability, we exclude 9 tests with too few TCOs (6 tests with one TCO and 3 tests with two TCOs), giving us a total of 98 tests.

We first consider *failing TCOs*, which have at least one failing run. For each test we count the number of failing TCOs and divide it by the total number of TCOs. If the ratio of failing TCOs is low, then the distribution of failures across TCOs is not uniform. Figure 6.3 shows a boxplot for the ratio of failing TCOs with a minimum of 4.8% (1 out of 21), maximum of 90.5% (19 out of 21), median of 19%, and interquartile range of 10% to 48%. From the skew in the distribution, TCOs do affect test failures often. The failures appear in a small ratio of the TCOs for a vast majority of tests. In fact, for 75% of tests, more than half of

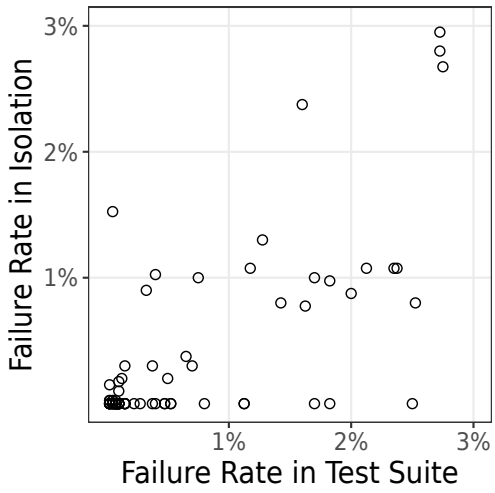


Figure 6.4: Failure rates in TSO and ISO for 80 tests that TSO detected, with both rates $\leq 3\%$; details of the other tests are in Table 6.2.

their TCOs have no failures, which would be unexpected if the failures were independent and uniformly distributed across TCOs. Our manual inspection (Section 6.4) finds some tests that are definitely OD, NDOD, and NDOI, and the ratio of failing TCOs for these categories is 4.8%–19%, 4.8%–58.3%, and 66.7%, respectively. A higher ratio indicates that a test may be NDOI.

We also conduct a χ^2 test of independence to identify significant differences in failure rates across different TCOs. Out of 98 tests, 70 have a p-value lower than 0.05. For these tests, the null hypothesis that the failure rates per TCO are the same is rejected. Their failure rates significantly differ across TCOs, and they are likely NDOD. For the remaining 28 tests, the null hypothesis cannot be rejected: while there is no clear evidence that the tests are NDOD, it does not necessarily imply that the tests are NDOI, as we show in Section 6.4.2.

We finally consider differences of the maximal burst length across TCOs for each test. We observe that 23 out of 107 tests fail in only one order. We consequently exclude these from our analysis of differences across failing TCOs. Of the remaining 84 tests, 52 have an identical maximal burst length across all TCOs, which is 1 (i.e., no consecutive failures) for 48 tests and 100 (i.e., all repetitions fail) for 4 tests. As we discuss in Section 6.4.1, the latter tests are confirmed to be OD tests by our manual inspection. Of the remaining 32 tests, the maximal burst length alternates between 1 and 2 for 21 tests and varies by small numbers (1–3, 1–4, 1–7, 2–6, 3–5, 3–6) for the remaining 11 tests.

Guideline: We find that a lot of tests may be NDOD; to detect them, it is better to run tests in more TCOs with fewer times each than in fewer TCOs with more times each.

Table 6.2: 27 tests with > 3% TSO or ISO failure rate.

MID	Test name	Failure rate [%] TSO	ISO
M1	Issue1298.test_for_issue_1	5.0	*100.0
M1	Issue1298.test_for_issue	5.0	*100.0
M1	DefaultExtJSONParser_parseArray.test_7	2.5	*100.0
M1	DefaultExtJSONParser_parseArray.test_8	2.5	*100.0
M3	PortTelnetHandlerTest.testListAllPort	10.0	0.0
M3	PortTelnetHandlerTest.testListDetail	10.0	0.0
M6	TestWriter.shouldWriteToElasticPort	11.4	0.0
M9	TestLRMovingSimMain.testSim	3.0	3.7
M11	TestParallelMatcher.testParallelSlowLog	0.3	58.6
M11	TestPartitionMatcher.testParallelSlowLog	0.8	68.2
M13	TestGetFunction.test	0.7	4.2
M13	TestMemoryLimit.test_no_abuse	0.6	57.1
M20	ZKTest.testBulkClusterJoining	49.0	0.0
M20	ZKTest.testDiscoveryListener	49.0	0.0
M20	ZKTest.testMembershipJoinAndLeave	50.0	0.0
M23	Issue256Test.runReconnectBlockingScenario9	3.2	0.5
M23	Issue256Test.runReconnectScenario0	5.1	16.5
M23	Issue256Test.runReconnectScenario1	1.6	16.5
M23	Issue256Test.runReconnectScenario2	1.2	15.9
M23	Issue256Test.runReconnectScenario3	1.6	16.1
M23	Issue256Test.runReconnectScenario4	1.6	15.8
M23	Issue256Test.runReconnectScenario5	1.6	16.6
M23	Issue256Test.runReconnectScenario6	1.9	15.3
M23	Issue256Test.runReconnectScenario7	1.9	14.9
M23	Issue256Test.runReconnectScenario8	2.7	16.3
M23	Issue256Test.runReconnectScenario9	2.3	15.7
M26	RetryAfterDelayFunctionTest.shouldRetry...DelayDate	10.9	39.1

6.3.3 RQ3: Reproducing TSO Failures in ISO

After encountering a test failure from running a test suite (TSO), developers are likely to debug the test by running it in isolation (ISO), because ISO runs faster. However, it is unclear how many flaky tests can be reproduced in ISO. Moreover, even when both TSO and ISO detect a test, the failure rates in TSO and ISO can greatly differ, as shown in Figure 6.4 and Table 6.2. The figure shows a scatterplot for the 80 tests where both failure rates are below 3%, and the table lists the actual failure rates for the other tests.

From Table 6.1 we see that ISO detected only 50 of the 107 tests that TSO detected, despite both having the same number of runs (4000). That is, ISO did not detect 57 tests by itself, but note that four tests from M1 (shown in Table 6.2) do have failing runs in ISO—in fact,

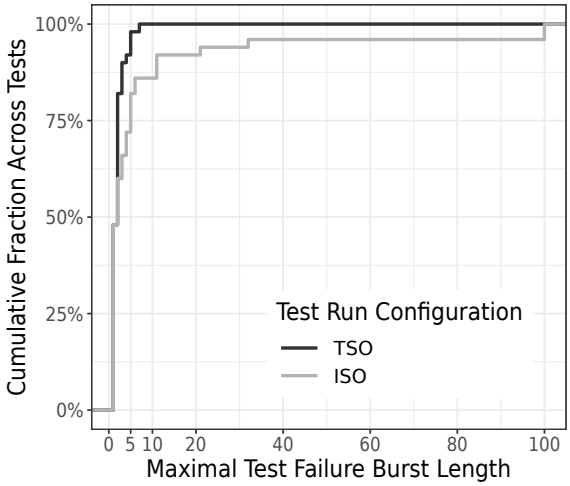


Figure 6.5: Distribution of maximal burst lengths across 50 tests for TSO & ISO.

these are OD tests that failed for all ISO runs (marked with '*' in the table)—but because they have no passing runs, we do not consider them detected in ISO. The difference in the number of detected tests already shows that reproducing passing and failing runs in TSO and ISO can greatly differ. Failure rates can also differ: of the 50 tests that ISO detected, 19 tests have their failure rates lower for ISO, 3 have it equal, and 28 have it higher. For the tests where the ISO failure rate is lower, the maximum and median difference is 2.6pp and 0.6pp, respectively. For the tests where it is higher, the maximum and median difference is 67.4pp and 7.4pp, respectively. We analyze in detail some of the tests with large differences in Section 6.4.

Because the observed failure rates are based on a sample of runs, we also perform statistical tests to check whether the differences are statistically significant. We conduct a paired Wilcoxon signed-rank test and a Kolmogorov-Smirnov (KS) test on the failure rate distributions obtained in our experiments. We chose the Wilcoxon signed-rank test because (1) it is based on a pair-wise comparison of the failure rate for individual tests, rather than an overall statistic on the failure rate distribution; and (2) it intuitively captures how often either TSO or ISO yields a higher failure rate, weighted by the rank of the difference magnitude. By mapping the actual magnitude of the difference to a rank, the test statistic is robust to outliers in the differences of failure rates. On our dataset, the test results in a p-value of 0.041. Therefore, the difference between the observed failure rates for TSO and ISO test executions is significant at the 0.05 level. However, the Wilcoxon test's mapping to ranks loses information on the magnitude of difference, which is captured by the CDFs on which the KS test is based. For our dataset, the KS test results in a p-value of 2.9e-12, which strongly rejects the null hypothesis that the samples come from the same distribution.

We also compare the maximal burst length for TSO and ISO runs. Figure 6.5 shows the CDFs of the maximal burst length per test for all tests detected by both TSO and ISO. From the plot, we observe that TSO reaches 82% for a maximal burst length of 2 test failures. However, for ISO the maximal burst length tends to be longer, e.g., ISO would reach 82% for a maximal burst length of 5 test failures instead. Due to the larger maximal burst length for ISO test executions, if one aims to find whether a test can pass, it could be beneficial to rerun failing tests in their test suites rather than in isolation, especially when the number of tests in the prefix of the test suite run before the test that failed is low. While rerunning in isolation has the advantage of reducing test runtime to an individual test, it entails the overhead for a potentially longer burst length before a passing run can be observed. In contrast, if one aims to get a failure, e.g., for debugging, it appears more beneficial to run the tests in ISO.

Guideline: We find that 53% of flaky tests detected in TSO runs are not detected in ISO runs. We also find that the maximal burst length tends to be longer for ISO than for TSO, which suggests that developers debugging flaky tests should run the tests in ISO. Dually, to detect flaky tests, running tests in TSO tends to be better because the smaller burst length is more likely to lead to a passing and failing run in fewer TSRs.

6.3.4 RQ4: Effect of Flaky Tests on TSRs

Whether developers can merge their recent changes to a project typically depends on whether the entire test suite passes or not with their recent changes. Even one flaky-test failure from a test-suite run (TSR) would prevent the developers from merging their recent code changes. Whether a TSR would fail due to flaky tests depends on the number of flaky tests in the test suite, the failure rate of each flaky test, and how related the flaky tests are with one another. For example, a test suite with 4000 runs that has two flaky tests that each fail 20 times, will have a minimum of 20 TSRs that fail (with exactly two failures per failed TSR) and a maximum of 40 TSRs that fail (with exactly one failure per failed TSR).

We study the TSR failure rates and how they relate to individual test failure rates to understand how often (1) developers would encounter TSR failures and (2) failures of different flaky tests are related to one another. Table 6.1 shows the actual TSR failure rate we obtained across all 4000 TSRs. Overall, we find that developers encounter TSR failures between <0.1% (only one TSR failed for M4 and M17) to 50% (2000 TSRs failed from M20). On average, developers would encounter a TSR failure in 5.4% of TSRs.

From Table 6.1 we can also see that 12 modules have only one flaky test detected, so their minimum, maximum, and sum failure rate is the same as the TSR failure rate. For the

other 14 modules, we find that the TSR failure rate is the same as the maximum failure rate for 21% (3 / 14) of modules (M12, M16, M20). Thus, the flaky tests in these modules are related to one another (i.e., when one fails, more also fail). When we manually investigated M20's three flaky tests, we do indeed find that two tests always failed together, and the third test fails with them, but the third test fails in one more run and is responsible for M20's maximum failure rate. Specific details of these tests are in Section 6.4.2. We further find that the TSR failure rate is the same as the sum failure rate for 43% (6 / 14) of modules. Thus, the flaky tests in these six modules are likely independent of one another (i.e., they always fail separately). For the remaining 36% (5 / 14) of modules, the TSR failure rate is in between the potential minimum and maximum TSR failure rates, suggesting that their flaky tests are a mix of related and independent flaky tests.

Guideline: At least 21% and up to 57% of test suites with multiple flaky tests have flaky tests related to one another, so flaky-test management systems [107, 128] could present the related flaky-test failures to help developers prioritize which tests to fix, e.g., first fix flaky tests that are related.

6.4 MANUALLY INSPECTED FLAKY TESTS

To better understand flaky tests, we manually inspected the root causes of flakiness for a number of tests. We selected a variety of tests from different modules, including NOD and OD tests, tests with high TSO or ISO failure rates, and tests with high and low χ^2 p-values for failure rates across TCOs. Inspecting the first flaky test of a new test class takes about a day on average. In sum, we inspected 28 tests and found 7 OD, 14 NDOD, 4 NDOI, and 3 more NOD that are difficult to confirm as NDOD or NDOI. We found that low p-values properly mark NDOD tests, but high p-values may not be NDOI tests, especially for tests that have a small overall number of test failures across all TCOs.

6.4.1 OD Tests

While this chapter aims to study NOD tests, our experiments did encounter 7 tests categorized as OD, i.e., the failure rate for each TCO was either 0% or 100%. Our inspection confirmed that all these tests are indeed OD. 5 tests were already fixed [161, 162, 164] as part of our iFixFlakies work [186]; note that the iDFlakies dataset used in our experiments has older commits, not the latest commit. Note also that the iDFlakies study [108] detected many more OD tests in the modules used in our experiments, because the iDFlakies tool perturbs the order of *all test methods* in a test suite, whereas our use of `mvn test` only perturbs

the order of *some test classes*. The remaining 2 tests had not been fixed, so as a contribution of this work we provided a pull request [163] that the developers already accepted.

6.4.2 NDOD Tests

High TSO Failure Rate. Table 6.2 lists three tests that have a much higher failure rate for TSO than for ISO. All three tests are from the module M20 and class `ZKTest`. Our inspection shows that all three tests fail for the same reason. These tests check certain network operations, which require obtaining a port number. All three tests share the same port numbers, and when they use a port, they mark that by creating a file in the `/tmp` directory, which is never deleted. The test code allows these tests to use 152 different ports. As a result, after all three tests are run 50 times each, they mark 150 ports, and so in the 51st run two tests will pass and then one will fail, while from the 52nd run all three will start failing. In fact, these failures are deterministic. Strictly speaking, the tests are still NDOD because they both pass and fail for the exact same test order in TSO. In contrast, none of the tests fails in ISO. The reason is that we run each test 100 times on one virtual machine (VM) and then allocate a fresh VM for the next 100 runs. (Depending on the CI system, developers may also not encounter these failures, e.g., Travis [208] uses a fresh VM for every TSR.) Thus, each test marks only 100 ports in its 100 runs and does not reach the 152; had we run the test in ISO for 153 or more times on the same VM, we would have also encountered failures. These examples illustrate that running multiple tests can, in some circumstances, have a higher failure rate than running only each test in isolation.

High ISO Failure Rate. Table 6.2 also lists three tests that have a much higher failure rate for ISO than for TSO. Two of the tests are from the module M11, classes `TestPartitionMatcher` and `TestParallelMatcher`. It turns out that both are subclasses of an abstract class that defines the method `testParallelSlowLog`. Both tests fail in some isolation runs, and the exception message indicates that some transaction was too slow. These tests run three transactions each, and fail when one of them takes too much time. The transactions take much longer in ISO runs than in TSO runs for the following reason. Each transaction is executed in a thread. In TSO runs, a previous test creates threads and caches them, so a later test can run quicker by reusing the created cache. In fact, `testParallelSlowLog` uses an API call that checks the cache prior to creating threads. However, in ISO runs, this test always creates a thread and then runs the transaction. As a result, the test fails much more often in the ISO runs.

Another test is from the module M13, class `TestMemoryLimit`. This test fails when some

resource bound is exceeded. Specifically, the project provides a sandbox for executing JavaScript in Java, and this test checks that some execution of a JavaScript program does not exceed a certain amount of memory. The memory check does not consider the entire heap but only the amount of memory allocated by the thread that the test executes. When the test runs in isolation, it allocates all the memory and fails often, but not always, as the memory check is done every 50ms. Therefore, the test may pass the memory check at one point and then finish in less than 50ms, despite going over the memory limit after the check. When the test runs in the test suite, another test runs before it and allocates many shared objects. Thus, the second test can use these shared objects from the heap and allocates less, so it fails much less often.

These examples illustrate how running tests in isolation can fail more often than running in the test suite because the test depends on some resource (shared memory, runtime) that can benefit from the tests that run before this test. However, running a test after others in a test suite could also negatively impact the test. In general, we cannot tell a priori whether running a test after others would be beneficial or hurtful. For example, consider just the runtime. A test may run faster after others because others can prepare shared state such as (1) load classes needed for test execution so the test in question need not reload those classes; (2) execute shared code and trigger JIT compilation so that the execution of the test in question executes optimized code; or (3) bring files from disk into memory so it becomes faster to access for the test in question. On the other hand, a test may run slower after others because others can put some pressure on the shared resources, e.g., (1) allocate memory so that garbage collection takes more time; (2) spawn threads that are not shut down so that the test in question has to compete with the other threads; (3) create I/O requests (e.g., write to disk or send network packets) so that the requests from the test in question take more time, etc.

High χ^2 p-Value. We finally discuss two example tests that have high p-values but our inspection still finds the failures to depend on the test order. One test is from the module M15, class `CronTest`. The test creates a pattern matcher for time which itself calls `DateUtil.date().second()` to initialize the matcher. The test also explicitly creates another time object calling `DateUtil.current(false)` to be matched with the matcher. Both calls get milliseconds and translate them into seconds, minutes, hours, and dates. The test fails if the two calls have a different value for seconds. The two calls are executed nearby, so the chance is small that the first call is executed right at the end of one second interval and the second call right at the start of the next second interval. The probability for the test to fail depends on how much time it takes between the two method calls. In our experiments,

this time is ∼15ms for ISO runs, i.e., the test fails if the first call gets milliseconds that modulo 1000 give values 985–999, so the test fails in ∼15/1000=1.5% of runs. In contrast, the code runs faster in TSO (due to the already discussed effects of class loading and JIT compilation), so the test fails less frequently, only ∼1/1000=0.1% of runs. Moreover, the failure rates differ across the two test orders: in one order this class runs second, and the test never fails; in the other order the class runs first, so the test can fail but still less frequently than in ISO because other test methods run before this test.

Another test is from the module M13, class `TestIssue34`. It is similar to the previously discussed M13 test and fails if a memory limit is exceeded. The test takes more memory in ISO than in TSO, as expected. Our additional experiments, after 4000 TSRs, show the test takes 840–900K in ISO runs, and 510–630K in TSO runs (when run late in a TCO). The limit is 1000K, so one may expect the test to more likely fail in ISO than in TSO. However, in 4000 runs, the test exceeded the memory limit in one TSO run but never in ISO. Because of the small number of failures, the p-value is high, yet the test manifestly depends on the order and is an NDOD.

Others. One test is already explained in Section 1.2.3. We omit detailed descriptions of 5 tests due to their similarity to the tests already explained: `TestWriter` from M6 is flaky because of timeout, whereas `CompletableThrowingSafeSubscriberTest`, `CompletableThrowingTest`, and `SingleThrowingTest` from M21 and `Issue621Test` from M23 are flaky because of concurrency.

6.4.3 NDOI Tests

Tests that have similar failure rates across TCOs in TSO (and also similar in ISO) are likely NDOI. We inspect several tests with high p-value. All four tests from the module M16, class `TimeSeriesGroupingIteratorTest`, are NDOI. These tests have (1) for each test, similar failure rates in the TSO and ISO runs; and (2) across all tests, similar TSO failure rates and ISO failure rates. In fact, many of these tests often fail together in the test suite (thus the TSR failure rate for their module is the same as the maximum TSO failure rate for individual tests). Furthermore, in our experiments, we find that each test fails in bursts, whether in TSO or ISO, i.e., a test fails 3–4 times in a row (if it fails in 100 runs at all).

The error message does not hint at the root cause but says that some averages of numeric values differ in two data structures. Our inspection shows that all of the tests populate these data structures with random numbers, and the random seed is based on the current time. The time is taken in milliseconds and translated into seconds, minutes, hours, and the date.

A careful analysis of `testTimeSeriesDropOff` and `testMultipleTimeSeriesMovingAverage` shows that they fail when the time seed translates into the range of approximately 58min:20sec to 59min:55sec (for any hour or date); if a test is run earlier or later, it passes. (The reason is that each test initializes the two data structures based on the time using offsets of 5sec and 100sec.) Most precisely, in each hour there are 95000 millisecond values for which each test fails, so assuming that each test can be run uniformly for any millisecond, each test is expected to fail in 95000/(60*60*1000)=2.64% of runs. In our experiments, the tests indeed have similar failure rates: 2.95% in ISO and 2.72% in TSO for `testTimeSeriesDropOff`, and 2.80% in ISO and 2.72% in TSO for `testMultipleTimeSeriesMovingAverage`. The other two tests, `testManySparseTimeSeries` and `testAdditionalTimeSeries`, behave similarly.

Abstracting from the details, these tests show some example NDOI tests that do not depend on the test order but depend only on the time when they are run. Such tests that definitely do not depend on the order appear to be rather rare.

6.4.4 NOD Tests Difficult to Classify

We inspected three NOD tests that are difficult to classify as NDOD or NDOI. We selected these tests based on high p-values (e.g., one test has p-value of 1), but some have a low number of failures (e.g., one test fails twice in 2000 TSRs of one TCO but does not fail in 100 TSRs for any of the other 20 TCOs). The root cause for all three tests is concurrency [123].

Two tests are from the module M5, class `RepublisherImplTest`. Both tests have a concurrent order violation. Effectively, each test has two threads with a shared map object that has one element before one thread calls `toBePublished.remove(event)`, while another checks `assertEquals(0, toBePublished.size())`. If the execution switches from one thread to the other at a particular point, the test fails with `expected:<0> but was:<1>`. We can get each test to reproducibly fail if we add some delay at that point. Developers likely encountered these problems before as both tests have commented `sleep(2000)`. In fact, the message for one commit that commented out that sleep is "Speed up tests ..."; while the tests may run faster, they became (more) flaky. Unfortunately, reasoning about the probability that a test run with two threads makes a context switch at exactly some point is rather challenging, so we cannot precisely determine if these tests are NDOD or NDOI.

Another test is from module M14, class `WaiterTest`. This test creates one thread that executes `lock.wait(millis)`. The main thread has `Thread.sleep(20)` and then effectively calls `lock.notify()`. However, if `notify` is called before `wait`, the signal is missed, and `wait` would block forever if it were not for the timeout of `millis=1000`. We can make the test to fail deterministically by adding a delay in the right place in the code under test. We can

also delete the existing `sleep(20)` in the test to make it fail determinstically. Unfortunately, reasoning precisely and analytically whether the probability that `notify` is missed due to TCO is again rather challenging because it requires determining the execution times of various events controlled by the JVM. As discussed in Section 6.3.2, an empirical approach would be to just run the tests many more times to observe more failures and use a statistical analysis to check failure rates across TCOs.

6.5 THREATS TO VALIDITY

A threat to validity is that our study uses only 26 modules from 23 Maven-based, Java projects. These modules may not be representative, causing our results to not generalize well. We attempt to mitigate this threat by using modules from iDFlakies [85], selecting them as described in Section 6.2.1.

As our study is on flaky tests, particularly NOD tests, it is likely that some specific numbers (e.g., number of NOD tests or failure rates) would change if the tests were run more times or on different machines. We attempt to mitigate this threat by running every test suite 4000 times in 40 batches, and every TCO at least 100 times. For every flaky test found by TSO, we again run it 4000 times in isolation. At the time that our experiments were done, they consisted of the largest number of runs in any published study of flaky tests. We also manually inspect 28 tests to check the root cause and categorization in Section 6.4.

The findings from our RQs in Section 6.3 may be influenced by the types of statistical tests that we used to interpret the data. We attempt to mitigate this threat by considering two statistical tests for RQ3 (Section 6.3.3). We also make all data and scripts that were used to generate the plots and figures in our paper publicly available on our website [55] so that others may interpret the data however they see fit.

6.6 SUMMARY

Flaky tests are caused by various sources of non-determinism, and the research community can benefit from multiple studies to understand flaky tests and develop new solutions for them. Several studies of flaky test have keyed on one group of flaky tests, order-dependent tests. We show that the other group, called "non-deterministic" tests, also has many tests that actually do depend on the test order, sometimes in complex ways. These tests have significantly different failure rates in different test orders and in isolated runs. To capture the complexity of these tests, we propose the term *non-deterministic, order-dependent (NDOD)*

tests. We manually inspect a number of flaky tests to show concrete, real-world examples. We hope that our study motivates more researchers to tackle this practically important problem.

CHAPTER 7: [TAMING] ACCOMMODATING ORDER-DEPENDENT FLAKY TESTS

This chapter presents our work on enhancing regression testing techniques to accommodate OD tests so that these tests encounter fewer spurious failures when regression testing techniques are used. Section 7.1 presents our motivational study to understand the impact that OD tests have on regression testing techniques. Section 7.2 presents our work on enhancing regression testing techniques, and Section 7.3 presents our evaluation of our enhanced techniques. Section 7.4 presents our discussions of the work in this chapter. Finally, Section 7.5 presents our threats to validity, and Section 7.6 concludes this chapter.

7.1 IMPACT OF ORDER-DEPENDENT TESTS

To understand how often traditional regression testing techniques lead to flaky-test failures due to OD tests, which we call *OD-test failures*, we evaluate a total of 12 algorithms from three well-known regression testing techniques on 11 Java modules from 8 real-world projects with test suites that contain OD tests.

7.1.1 Traditional Regression Testing Techniques

Test prioritization, selection, and parallelization are traditional regression testing techniques that aim to detect faults faster than simply running all of the tests in the given test suite. We refer to the order in which developers typically run all of these tests as the *original order*. These traditional regression testing techniques produce orders (permutations of a subset of tests from the original order) that may not satisfy test dependencies.

Test Prioritization. Test prioritization aims to produce an order for running tests that would fail and indicate a fault sooner than later [45, 89, 101, 117, 171, 173, 192, 227]. Prior work [45] proposed test prioritization algorithms that reorder tests based on their (1) total coverage of code components (e.g., statements, methods) and (2) additional coverage of code components *not* previously covered. These algorithms typically take as input coverage information from a prior version of the code and test suite, and they use that information to reorder the tests on future versions[1]. We evaluate 4 test prioritization algorithms proposed in prior work [45]. Table 7.1 gives a concise description of each algorithm. Namely, the

[1]There is typically no point collecting coverage information on a future version to reorder and run tests on that version, because collecting coverage information requires one to run the tests already.

Table 7.1: Four evaluated test prioritization algorithms.

Label	Ordered by
T1	Total statement coverage of each test
T2	Additional statement coverage of each test
T3	Total method coverage of each test
T4	Additional method coverage of each test

Table 7.2: Six evaluated test selection algorithms.

Label	Selection granularity	Ordered by
S1	Statement	Test ID (no reordering)
S2	Statement	Total statement coverage of each test
S3	Statement	Additional statement coverage of each test
S4	Method	Test ID (no reordering)
S5	Method	Total method coverage of each test
S6	Method	Additional method coverage of each test

Table 7.3: Two evaluated test parallelization algorithms.

Label	Algorithm description
P1	Parallelize on test ID
P2	Parallelize on test execution time

algorithms reorder tests such that the ones with more total coverage of code components (statements or methods) are run earlier, or reorder tests with more additional coverage of code components not previously covered to run earlier.

Test Selection. Test selection aims to select and run a subsuite of a program's tests after every change, but to detect the same faults as if the full test suite is run [20, 77, 83, 146, 157, 169, 227, 234]. We evaluate 6 test selection algorithms that select tests based on their coverage of modified code components [20, 77]; Table 7.2 gives a concise description of each algorithm. The algorithms use program analysis to select every test that may be affected by recent code modifications [77]. Each algorithm first builds a control-flow graph (CFG) for the then-current version of the program P_{old}, runs P_{old}'s test suite, and maps each test to the set of CFG edges covered by the test. When the program is modified to P_{new}, the algorithm builds P_{new}'s CFG and then selects the tests that cover "dangerous" edges: program points where P_{old} and P_{new}'s CFGs differ. We choose to select based on two levels of code-component granularity traditionally evaluated before, namely statements and methods [227]. We then order the selected tests. Ordering by test ID (an integer representing the position of the test in the original order) essentially does no reordering, while the other orderings make the algorithm a combination of test selection followed by test prioritization.

Test Parallelization. Test parallelization schedules the input tests for execution across multiple machines to reduce test latency — the time to run all tests [93, 102, 140, 193]. Test parallelization techniques are widely adopted in industry. For example, Visual Studio 2010 (and later) supports a model of executing tests in parallel on a multi-CPU/core machine [51]. Two popular automated approaches for test parallelization are to parallelize a test suite based on (1) test ID and (2) execution time from prior runs [174]. A test ID is an integer representing the position of the test in the original order. We evaluate one test parallelization algorithm based on each approach (as described in Table 7.3). The algorithm that parallelizes based on test ID schedules the i^{th} test on machine $i \bmod k$, where k is the number of available machines and i is the test ID. The algorithm that parallelizes based on the tests' execution time (obtained from a prior execution) iteratively schedules each test on the machine that is expected to complete the earliest based on the tests already scheduled so far on that machine. We evaluate each test parallelization algorithm with $k = 2$, 4, 8, and 16 machines.

7.1.2 Evaluation Projects

Our evaluation projects consist of 11 modules from 8 Maven-based Java projects. These 11 modules are a subset of modules from the comprehensive version of a published dataset of flaky tests [85]. We include all of the modules that contain OD tests, except for eight modules that we exclude because they are either incompatible with the tool that we use to compute coverage information or they contain OD tests where the developers have already specified test orderings in which the OD tests should run. A list of modules that we exclude from the dataset and the reasons for why we exclude them are available on our website [5].

In addition to the existing human-written tests, we also evaluate automatically generated tests. Automated test-generation tools [25, 28, 29, 58, 158, 238] are attractive because they reduce developers' testing efforts. These tools typically generate tests by creating sequences of method calls into the code under test. Although the tests are meant to be generated independently from one another, these tools often do not enforce test independence because doing so can substantially increase the runtime of the tests (e.g., restarting the VM between each generated test). This optimization results in automatically generated test suites occasionally containing OD tests. Given the increasing importance of automatically generated tests in both research and industrial use, we also investigate them for OD tests. Specifically, we use Randoop [158] version 3.1.5, a state-of-the-art random-based test-generation tool. We configure Randoop to generate at most 5000 tests for each evaluation project, and to drop tests that are subsumed by other tests (tests whose sequence of method calls is a subsequence of those in other tests).

Table 7.4: Statistics of the projects used in our evaluation.

ID	Project	LOC		# Tests		# OD tests				# Evaluation versions	Days between vers.	
		Main	Tests	Human	Auto	Human		Auto			Average	Median
M1	apache/incubator-dubbo - m1	2394	2994	101	353	2	(2%)	0	(0%)	10	4	5
M2	- m2	167	1496	40	2857	3	(8%)	0	(0%)	10	25	18
M3	- m3	2716	1932	65	945	8	(12%)	0	(0%)	10	10	8
M4	- m4	198	1817	72	2210	14	(19%)	0	(0%)	6	32	43
M5	apache/struts	3015	1721	61	4190	4	(7%)	1	(<1%)	6	61	22
M6	dropwizard/dropwizard	1718	1489	70	639	1	(1%)	2	(<1%)	10	11	6
M7	elasticjob/elastic-job-lite	5323	7235	500	566	9	(2%)	4	(1%)	2	99	99
M8	jfree/jfreechart	93915	39944	2176	1233	1	(<1%)	0	(0%)	5	13	2
M9	kevinsawicki/http-request	1358	2647	160	4537	21	(13%)	0	(0%)	10	42	4
M10	undertow-io/undertow	4977	3325	49	967	6	(12%)	1	(<1%)	10	22	15
M11	wildfly/wildfly	7022	1931	78	140	42	(54%)	20	(14%)	10	37	19
Total / Average / Median		**122803**	**66531**	**3372**	**18637**	**111**	**(3%)**	**28**	**(<1%)**	**89**	**25**	**8**

7.1.3 Methodology

Regression testing algorithms analyze one version of code to obtain metadata such as coverage or time information for every test so that they can compute specific orders for future versions. For each of our evaluation projects, we treat the version of the project used in the published dataset [85] as the latest version in a sequence of versions. In our evaluation, we define a "version" for a project as a particular commit that has a change for the module containing the OD test, and the change consists of code changes to a Java file. Furthermore, the code must compile and all tests must pass through Maven. We go back at most 10 versions from this latest version to obtain the First Version (denoted as *firstVer*) of each project. We may not obtain 10 versions for a project if it does not have enough commits that satisfy our requirements, e.g., commits that are old may not be compilable anymore due to missing dependencies. We refer to each subsequent version after *firstVer* as a *subseqVer*. For our evaluation, we use *firstVer* to obtain the metadata for the regression testing algorithms, and we evaluate the use of such information on the *subseqVer*s. For automatically generated test suites, we generate the tests on *firstVer* and copy the tests to *subseqVer*s. Any copied test that does not compile on a *subseqVer* is dropped from the test suite on that version.

Table 7.4 summarizes the information of each evaluation project. Column "LOC" is the number of non-comment, non-blank lines in the project's main code and human-written tests as reported by `sloc` [189] for *firstVer* of each evaluation project. Column "# Tests" shows the number of human-written tests and those generated by Randoop [158] for *firstVer* of each evaluation project. Column "# Evaluation versions" shows the number of versions from *firstVer* to latest version that we use for our evaluation, and column "Days between versions" shows the average and median number of days between the versions that we use for our evaluation.

To evaluate how often OD tests fail when using test prioritization and test parallelization algorithms, we execute these algorithms on the version immediately following *firstVer*, called

the Second Version (denoted as *secondVer*). For test selection, we execute the algorithms on all versions after *firstVer* up to the latest version (the version from the dataset [85]). The versions that we use for each of our evaluation projects are available online [5]. For all of the algorithms, they may rank multiple tests the same (e.g., two tests cover the same statements or two tests take the same amount of time to run). To break ties, the algorithms deterministically sort tests based on their ordering in the original order. Therefore, with the same metadata for the tests, our algorithms would always produce the same order.

For the evaluation of test prioritization algorithms, we count the number of OD tests that fail in the prioritized order on *secondVer*. For test selection, given the change between *firstVer* and the future versions, we count the number of unique OD tests that fail from the possibly reordered selected tests on all future versions. For test parallelization, we count the number of OD tests that fail in the parallelized order on any of the machines where tests are run on *secondVer*. All test orders are run three times, and a test is counted as OD only if it consistently fails for all three runs. Note that we count all failed tests as OD tests because we ensure that all tests pass in the original order of each version that we use.

Note that the general form of the OD test detection problem is NP-complete [237]. To get an approximation for the maximum number of OD-test failures with which the orders produced by regression testing algorithms can cause, we apply DTDetector [237] to randomize the test ordering for 100 times on *firstVer* and all *subseqVer*s. We choose randomization because prior work [85, 237] found it to be the most effective strategy in terms of time cost when finding OD tests. The DTDetector tool is sound but incomplete, i.e., every OD test that DTDetector finds is a real OD test, but DTDetector is not guaranteed to find every OD test in the test suite. Thus, the reported number is a lower bound of the total number of OD tests. Column "# OD tests" in Table 7.4 reports the number of OD tests that DTDetector finds when run on all versions of our evaluation projects.

There are flaky tests that can pass or fail on the same version of code but are not OD tests (e.g., flaky tests due to concurrency). For all of the test suites, we run each test suite 100 times in its original order and record the tests that fail as flaky but not as OD tests. We use this set of NOD flaky tests to ensure that the test failures observed on versions after *firstVer* are likely due to OD tests and not other categories of flaky tests.

7.1.4 Results

Table 7.5 and Table 7.6 summarize our results (parallelization is averaged across $k = 2$, 4, 8, and 16). The exact number of OD-test failures for each regression testing algorithm is available on our website [5]. In Table 7.5, each cell shows the percentage of unique OD

tests that fail in all of the orders produced by the algorithms of a technique over the number of known OD tests for that specific evaluation project. Cells with a "n/a" represent test suites (of evaluation projects) that do not contain any OD tests according to DTDetector and the regression testing algorithms. The "Total" row shows the percentage of OD tests that fail across all evaluation projects per technique over all OD tests found by DTDetector. In Table 7.6, each cell shows the percentage of OD tests across all evaluation projects that fail per algorithm. The dependent tests that fail in any two cells of Table 7.6 may not be distinct from one another.

On average, 3% of human-written tests and <1% of automatically generated tests are OD tests. Although the percentage of OD tests may be low, the effect that these tests have on regression testing algorithms is substantial. More specifically, almost every project's human-written test suite has at least one OD-test failure in an order produced by one or more regression testing algorithms (the only exceptions are `jfree/jfreechart` (M8) and `wildfly/wildfly` (M11)). These OD-test failures waste developers' time or delay the discovery of a real fault.

According to Table 7.6, it may seem that algorithms that order tests by Total coverage (T1, T3, S2, S5) always have fewer OD-test failures than their respective algorithms that order tests by Additional coverage (T2, T4, S3, S6), particularly for test prioritization algorithms. However, when we investigate the algorithm and module that best exhibit this difference, namely kevinsawicki/http-request's (M9) test prioritization results, we find that this one case is largely responsible for the discrepancies that we see for the test prioritization algorithms. Specifically, M9 contains 0 OD-test failures for T1 and T3, but 14 and 24 OD-test failures for T2 and T4, respectively. All OD tests that fail for M9's T2 and T4 would fail when one particular test is run before them; we refer to this test that runs before as the *dependee test*. The dependent tests all have similar coverage, so in the Additional orders, these tests are not consecutive and many of them come later in the orders. The one dependee test then comes inbetween some dependent tests, causing the ones that come later than the dependee test to fail. In the Total orders, the dependee test has lower coverage than the dependent tests and is always later in the orders. If we omit M9's test prioritization results, we no longer observe any substantial difference in the number of OD-test failures of Total coverage algorithms compared to that of Additional coverage algorithms.

Impact on Test Prioritization. Test prioritization algorithms produce orders that cause OD-test failures for the human-written test suites in eight evaluation projects. For automatically generated test suites, test prioritization algorithms produce orders that cause OD-test failures in three projects. Our findings suggest that orders produced by test prioritization

Table 7.5: Percentage of OD tests that fail in orders produced by different regression testing techniques, per evaluation project.

	OD tests that fail (per evaluation project)					
	Prioritization		Selection		Parallelization	
ID	Human	Auto	Human	Auto	Human	Auto
M1	50%	n/a	100%	n/a	50%	n/a
M2	67%	n/a	67%	n/a	0%	n/a
M3	12%	n/a	50%	n/a	0%	n/a
M4	7%	n/a	14%	n/a	14%	n/a
M5	0%	100%	25%	100%	75%	100%
M6	100%	0%	0%	0%	0%	100%
M7	44%	50%	0%	0%	0%	25%
M8	0%	n/a	0%	n/a	0%	n/a
M9	71%	n/a	71%	n/a	0%	n/a
M10	17%	0%	17%	0%	0%	100%
M11	0%	60%	0%	0%	0%	30%
Total	23%	54%	24%	4%	5%	36%

Table 7.6: Percentage of OD tests that fail in orders produced by individual regression testing algorithms.

	OD tests that fail (per algorithm)											
	Prioritization				Selection						Parallelization	
Type	T1	T2	T3	T4	S1	S2	S3	S4	S5	S6	P1	P2
Human	5%	25%	5%	20%	1%	28%	31%	1%	5%	9%	2%	5%
Auto	36%	43%	71%	25%	4%	4%	4%	4%	4%	4%	21%	64%

algorithms are more likely to cause at least one OD-test failures in human-written test suites than automatically generated test suites. Overall, we find that test prioritization algorithms produce orders that cause 23% of the human-written OD tests to fail and 54% of the automatically generated OD tests to fail.

Impact on Test Selection. Test selection algorithms produce orders that cause OD-test failures for the human-written test suites in seven evaluation projects and for the automatically generated test suites in one project. These test selection algorithms produce orders that cause 24% of the human-written OD tests and 4% of the automatically generated OD tests to fail. The algorithms that do not reorder a test suite (S1 and S4) produce orders that cause fewer OD-test failures than the algorithms that do reorder. This finding suggests that while selecting tests itself is a factor, reordering tests is generally a more important factor that leads to OD-test failures.

Impact on Test Parallelization. Test parallelization algorithms produce orders that cause OD-test failures because the algorithms may schedule an OD test on a different machine than the test(s) that it depends on. We again find that the parallelization algorithm that

reorders tests, P2, produces orders that cause substantially more OD-test failures than P1. This result reaffirms our finding from Section 7.1.4 that reordering tests has a greater impact on OD-test failures than selecting tests. The percentages reported in Table 7.5 and Table 7.6 are calculated from the combined set of OD tests that fail due to parallelization algorithms for $k = 2$, 4, 8, and 16 machines. The orders of these algorithms cause 5% of the human-written OD tests and 36% of the automatically generated OD tests to fail on average.

7.1.5 Findings

Our study suggests the following two main findings.

(1) Regression testing algorithms that reorder tests in the given test suite are more likely to experience OD test failures than algorithms that do not reorder tests in the test suite. We see this effect both by comparing test selection algorithms that do not reorder tests (S1 and S4) to those that do, as well as comparing a test parallelization algorithm, P2, which does reorder tests, to P1, which does not. Developers using algorithms that reorder tests would especially benefit from our dependent-test-aware algorithms described in Section 7.2.

(2) Human-written and automatically generated test suites are likely to fail due to test dependencies. As shown in Table 7.5, we find that regression testing algorithms produce orders that cause OD-test failures in 82% (9 / 11) of human-written test suites with OD tests, compared to the 100% (5 / 5) of automatically generated test suites. Both percentages are substantial and showcase the likelihood of OD-test failures when using traditional regression testing algorithms that are unaware of OD tests.

7.2 DEPENDENT-TEST-AWARE REGRESSION TESTING TECHNIQUES

When a developer conducts regression testing, there are two options: running the tests in the original order or running the tests in the order produced by a regression testing algorithm. When the developer runs the tests in the original order, a test might fail because either (1) there is a fault somewhere in the program under test or test code, or (2) the test is flaky but it is not a OD test (e.g., flaky due to concurrency). However, when using a traditional regression testing algorithm, there is a third reason for why tests might fail: the produced orders may not be satisfying the test dependencies. Algorithms that are susceptible to this third reason do not adhere to a primary design goal of regression testing algorithms. Specifically, the orders produced by these algorithms should not cause OD-test failures: if the tests all pass in the original order, then the algorithms should produce only orders in

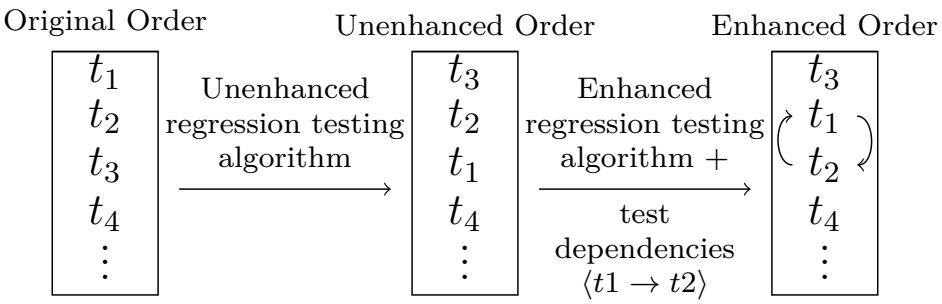


Figure 7.1: Example of an unenhanced and its enhanced, dependent-test-aware regression testing algorithms.

which all of the tests pass (and dually, if tests fail in the original order, then the algorithms should produce only orders in which these tests fail).[2] Since many real-world test suites contain OD tests, a regression testing algorithm that assumes its input contains no OD tests can produce orders that cause OD-test failures, violating this primary design goal (which we see from our results in Section 7.1).

We propose that regression testing techniques should be dependent-test-aware to remove OD-test failures. Our general approach completely removes all possible OD-test failures with respect to the input test dependencies. In practice, such test dependencies may or may not always be complete (all test dependencies that prevent OD-test failures are provided) or minimal (only test dependencies that are needed to prevent OD-test failures are provided). Nevertheless, our general approach requires as input an original order, a set of test dependencies, and an order outputted by a traditional regression testing algorithm to output an updated, enhanced order that satisfies the given test dependencies.

7.2.1 Example

Figure 7.1 shows an illustrated example of the orders produced by an unenhanced algorithm and its corresponding enhanced algorithm. The unenhanced algorithm does not take test dependencies into account, and therefore may produce orders that cause OD-test failures. Specifically, the unenhanced algorithm produces the Unenhanced Order. On the other hand, the enhanced algorithm produces the Enhanced Order, a test order that satisfies the provided test dependencies of the test suite. The enhanced algorithm does so by first using the unenhanced algorithm to produce the Unenhanced Order and then enforcing the test dependencies on the order by reordering or adding tests.

[2]Another design goal is to maximize fault-finding ability over time, i.e., efficiency. The efficiency goal is a trade-off against the correctness goal.

We define two different types of test dependencies, positive and negative dependencies. A *positive test dependency* $\langle p \rightarrow d \rangle$ denotes that for OD test d to pass, it should be run only after running test p, the test that d depends on. A *negative test dependency* $\langle n \nrightarrow d \rangle$ denotes that for OD test d to pass, it should *not* be run after test n. Prior work [186] refers to test p as a state-setter and test d in a positive dependency as a brittle. It also refers to test n as a polluter and d in a negative dependency as a victim. For simplicity, we refer to the tests that OD tests depend on (i.e., p and n) as *dependee tests*. For both types of dependencies, the dependee and OD test do not need to be run consecutively, but merely in an order that adheres to the specified dependencies.

In Figure 7.1, there is a single, positive test dependency $\langle t_1 \rightarrow t_2 \rangle$ in the input test dependencies. $\langle t_1 \rightarrow t_2 \rangle$ denotes that the OD test t_2 should be run only after running test t_1. In the Unenhanced Order, the positive test dependency $\langle t_1 \rightarrow t_2 \rangle$ is not satisfied and t_2 will fail. Our enhanced algorithm prevents the OD-test failure of t_2 by modifying the outputted order of the unenhanced algorithm so that the test dependency (t_2 should be run only after running t_1) is satisfied in the Enhanced Order.

7.2.2 General Approach for Enhancing Regression Testing Algorithms

Figure 7.2 shows our general algorithm, enhanceOrder, for enhancing an order produced by a traditional regression testing algorithm to become dependent-test-aware. enhanceOrder takes as input T_u, which is the order produced by the traditional unenhanced algorithm that enhanceOrder is enhancing (this order can be a different permutation or subset of T_{orig}), the set of test dependencies D, and the original test suite T_{orig}, which is an ordered list of tests in the original order. While in theory one could provide test dependencies that are not linearizable (e.g., both $\langle t_1 \rightarrow t_2 \rangle$ and $\langle t_2 \rightarrow t_1 \rangle$ in D), we assume that the provided test dependencies are linearizable, and that T_{orig} is one possible total order of the tests that satisfies all of the test dependencies in D. For this reason, enhanceOrder does not check for cycles within D. Based on T_u, enhanceOrder uses the described inputs to output a new order (T_e) that satisfies the provided test dependencies D.

enhanceOrder starts with an empty enhanced order T_e and then adds each test in the unenhanced order T_u into T_e using the addTest function (Line 7). To do so, enhanceOrder first computes T_a, the set of tests that the tests in T_u transitively depend on from the positive test dependencies (Line 4). T_a represents the tests that the traditional test selection or parallelization algorithms did not include in T_u, but they are needed for OD tests in T_u to pass. enhanceOrder then iterates through T_u in order (Line 5) to minimize the perturbations that it makes to the optimal order found by the traditional unenhanced algorithm. The addTest

enhanceOrder$(T_u, D, T_{\text{orig}})$:

1: $T_e \leftarrow []$
2: $P \leftarrow \{\langle p \rightarrow d \rangle \in D\}$ // Positive test dependencies
3: $N \leftarrow \{\langle n \nrightarrow d \rangle \in D\}$ // Negative test dependencies
4: $T_a \leftarrow T_u \circ (P^{-1})^*$ // Get all transitive positive dependee tests
5: **for** $t : T_u$ **do** // Iterate T_u sequence in order
6: **if** $t \in T_e$ **then** continue **end if**
7: $T_e \leftarrow \text{addTest}(t, T_e, T_u, T_{\text{orig}}, T_a, P, N)$
8: **end for**
9: **return** T_e

addTest$(t, T_e, T_u, T_{\text{orig}}, T_a, P, N)$:

10: $B \leftarrow \{t' \in T_u \cup T_a | \langle t' \rightarrow t \rangle \in P \ \vee \langle t \nrightarrow t' \rangle \in N\}$
11: $L \leftarrow \text{sort}(B \cap T_u, orderBy(T_u)) \oplus \text{sort}(B \setminus T_u, orderBy(T_{\text{orig}}))$
12: **for** $b : L$ **do** // Iterate the before tests in order of L
13: **if** $b \in T_e$ **then** continue **end if**
14: $T_e \leftarrow \text{addTest}(b, T_e, T_u, T_{\text{orig}}, T_a, P, N)$
15: **end for**
16: **return** $T_e \oplus [t]$

Figure 7.2: General approach to enhance an order from traditional regression testing algorithms.

function adds a test t into the current T_e while ensuring that all of the provided (transitive) test dependencies are satisfied. Once all of T_u's tests are added into T_e, enhanceOrder returns T_e.

On a high-level, the function addTest has the precondition that all of the tests in the current enhanced order T_e have their test dependencies satisfied in T_e, and addTest has the postcondition that test t is added to the end of T_e (Line 16) and all tests in T_e still have their test dependencies satisfied. To satisfy these conditions, addTest first obtains all of the tests that need to run before the input test t (Line 10), represented as the set of tests B.

The tests in B are all of the dependee tests within the positive test dependencies P for t, i.e., all tests p where $\langle p \rightarrow t \rangle$ are in P. Note that these additional tests must come from either T_u or T_a (the additional tests that the traditional algorithm does not add to T_u). Line 10 just includes into B the *direct* dependee tests of t and not those that it indirectly depends on; these indirect dependee tests are added to T_e through the recursive call to addTest (Line 14). The tests in B also include the dependent tests within the negative test dependencies N whose dependee test is t, i.e., all tests d where $\langle t \nrightarrow d \rangle$ are in N. addTest does not include test d that depends on t from the negative test dependencies if d is not in T_u or T_a. Specifically, these are tests that the unenhanced algorithm originally did not find necessary to include (i.e., for test selection the test is not affected by the change, or for test

parallelization the test is scheduled on another machine), and they are also not needed to prevent any OD tests already included in T_u from failing.

Once addTest obtains all of the tests that need to run before t, it then adds all of these tests into the enhanced order T_e, which addTest accomplishes by recursively calling addTest on each of these tests (Line 14). Line 11 first sorts these tests based on their order in the unenhanced order T_u. This sorting is to minimize the perturbations that it makes to the optimal order found by the unenhanced algorithm. For any additional tests not in T_u (tests added through T_a), they are sorted to appear at the end and based on their order in the original order, providing a deterministic ordering for our evaluation (in principle, one can use any topological order). Once all of the tests that must run before t are included into T_e, addTest will finally add t (Line 16).

OD-test failures may still arise even when using an enhanced algorithm because the provided test dependencies (D) may not be complete. For example, a developer may forget to manually specify some test dependencies, and even if the developer uses an automatic tool for computing test dependencies, such tool may not find all dependencies as prior work [237] has shown that computing *all* test dependencies is an NP-complete problem. Also, a developer may have made changes that invalidate some of the existing test dependencies or introduce new test dependencies, but the developer does not properly update the input test dependencies.

7.3 EVALUATION OF GENERAL APPROACH

Section 7.1 shows how both human-written and automatically generated test suites with OD tests have OD-test failures when developers apply traditional, unenhanced regression testing algorithms on these test suites. To address this issue, we apply our general approach described in Section 7.2 to enhance 12 regression testing algorithms and evaluate them with the following metrics.

- **Effectiveness of reducing OD-test failures:** the reduction in the number of OD-test failures after using the enhanced algorithms. Ideally, every test should pass, because we confirm that all tests pass in the original order on the versions that we evaluate on (the same projects and versions from Section 7.1.2). This metric is the most important desideratum.

- **Efficiency of orders:** how much longer-running are orders produced by the enhanced regression testing algorithms than those produced by the unenhanced algorithms.

7.3.1 Methodology

To evaluate 12 enhanced regression testing algorithms, we start with *firstVer* for each of the evaluation projects described in Section 7.1.2. Compared to the unenhanced algorithms, the only additional input that the enhanced algorithms require is test dependencies D. These test dependencies D and the other metadata needed by the regression testing algorithms are computed on *firstVer* of each evaluation project. The details on how we compute test dependencies for our evaluation are in Section 7.3.2. With D and the other inputs required by the unenhanced and enhanced algorithms, we then evaluate these algorithms on *subseqVer*s of the projects.

In between *firstVer* and *subseqVer*s there may be tests that exist in one but not the other. We refer to tests that exist in *firstVer* as *old tests* and for tests that are introduced by developers in a future version as *new tests*. When running the *old tests* on future versions, we use the enhanced algorithms, which use coverage or time information from *firstVer* (also needed by the unenhanced algorithms) and D. Specifically, for all of the algorithms, we use the following procedure to handle changes in tests between *firstVer* and a *subseqVer*.

1. The test in *firstVer* is skipped if *subseqVer* no longer contains the corresponding test.

2. Similar to most traditional regression testing algorithms, we treat tests with the same fully qualified name in *firstVer* and *subseqVer* as the same test.

3. We ignore *new tests* (tests in *subseqVer* but not in *firstVer*), because both unenhanced and enhanced regression testing algorithms would treat these tests the same (i.e., run all of these tests before or after *old tests*).

7.3.2 Computing Test Dependencies

Developers can obtain test dependencies for a test suite by (1) manually specifying the test dependencies, or (2) using automatic tools to compute the test dependencies [59, 73, 85, 216, 237]. For our evaluation, we obtain test dependencies through the latter approach, using automatic tools, because we want to evaluate the scenario of how any developer can benefit from our enhanced algorithms without having to manually specify test dependencies. Among the automatic tools to compute test dependencies, both DTDetector [237] and iDFlakies [85] suggest that randomizing a test suite many times is the most cost effective way to compute test dependencies. For our evaluation, we choose to use DTDetector since it is the more widely cited work on computing test dependencies. Before we compute test dependencies, we first filter out tests that are flaky but are not OD tests (e.g., tests that are flaky due to

concurrency [85, 126]) for each of our evaluation projects. We filter these tests by running each test suite 100 times in its original order and removing all tests that had test failures. We remove these tests because they can fail for other reasons and would have the same chance of affecting unenhanced and enhanced algorithms. To simulate how developers would compute test dependencies on a current version to use on future versions, we compute test dependencies on a prior version (*firstVer*) of a project's test suite and use them with our enhanced algorithms on future versions (*subseqVer*s).

DTDetector. DTDetector [237] is a tool that detects test dependencies by running a test suite in a variety of different orders and observing the changes in the test outcomes. DTDetector outputs a test dependency if it observes a test t to pass in one order (denoted as po) and fail in a different order (denoted as fo). Typically, t depends on either some tests that run before t in po to be dependee tests in a positive test dependency (some tests must always run before t), or some tests that run before t in fo to be dependee tests in a negative test dependency (some tests must always run after t). t can also have both a positive dependee test and a negative dependee test; this case is rather rare, and we do not observe such a case in our evaluation projects. DTDetector outputs the minimal set of test dependencies by delta-debugging [71, 230] the list of tests coming before t in po and fo to remove as many tests as possible, while still causing t to output the same test outcome.

When we use DTDetector directly as its authors intended, we find that DTDetector's reordering strategies require many hours to run and are designed to search for test dependencies by randomizing test orders; however, we are interested in test dependencies only in the orders that arise from regression testing algorithms. To address this problem, we extend DTDetector to compute test dependencies using the output of the regression testing algorithms. This strategy is in contrast to DTDetector's default strategies, which compute test dependencies for a variety of orders that may not resemble the outputs of regression testing algorithms. Specifically, for test prioritization or test parallelization, we use the orders produced by their unenhanced algorithms. DTDetector will find test dependencies for any test that fails in these orders, since these tests now have a passing order (all tests must have passed in the original order) and a failing order. Using these two orders, DTDetector will minimize the list of tests before an OD test, and we would then use the minimized list as test dependencies for the enhanced algorithms. If the new test dependencies with the enhanced algorithms cause new failing OD tests, then we repeat this process again until the orders for the enhanced algorithms no longer cause any OD-test failure.

For test selection, we simply combine and use all of the test dependencies that we find for the test prioritization and test parallelization algorithms. We use the other algorithms'

Table 7.7: Average time in seconds to run the test suite and average time to compute test dependencies for an OD test. "Prioritization" and "Parallelization" show the average time per algorithm, while "All 6" shows the average time across *all* 6 algorithms. "-" denotes cases that have no OD test for all algorithms of a particular technique.

	Suite run time		**Time to precompute test dependencies**					
			Prioritization		**Parallelization**		**All 6**	
ID	**Human**	**Auto**	**Human**	**Auto**	**Human**	**Auto**	**Human**	**Auto**
M1	7.3	0.3	64	-	24	-	44	-
M2	0.4	0.2	95	-	-	-	95	-
M3	184.2	0.2	1769	-	-	-	1769	-
M4	0.1	0.2	19	-	13	-	17	-
M5	2.4	0.4	-	396	31	215	31	275
M6	4.1	0.3	75	-	-	29	75	29
M7	20.4	45.6	241	242	-	176	241	216
M8	1.2	1.3	-	-	-	-	-	-
M9	1.2	0.1	28	-	-	-	28	-
M10	19.9	0.8	157	-	-	39	157	39
M11	2.3	0.4	-	484	-	210	-	438

orders because it is difficult to predict test selection orders on future versions, i.e., the set of tests selected in one version will likely be different than the set of tests selected in another version. This methodology simulates what real developers would do: they know what regression testing algorithm to use but do not know what code changes they will make in the future.

Time to Precompute Dependencies. Developers should not compute test dependencies as they are performing regression testing. Instead, as we show in our evaluation, test dependencies can be collected on the current version and be reused later.

Developers can compute test dependencies infrequently and offline. Recomputing test dependencies can be beneficial if new test dependencies are needed or if existing test dependencies are no longer needed because of the developers' recent changes. While the developers are working between versions v_i and v_{i+1}, they can use that time to compute test dependencies. Table 7.7 shows the time in seconds to compute the test dependencies that we use in our evaluation. The table shows the average time to compute test dependencies per OD test across all test prioritization or parallelization algorithms (for the columns under "Prioritization" and "Parallelization", respectively), and the time under "All 6" is the average time to compute dependencies per OD test across all six test prioritization and parallelization algorithms. The reported time includes the time for checks such as rerunning failing tests to ensure that it is actually an OD test (i.e., it is not flaky for other reasons) as to avoid the computation of non-existent test dependencies.

Table 7.8: Percentage of how many fewer OD-test failures occur in the test suites produced by the enhanced algorithms compared to those produced by the unenhanced algorithms. Higher percentages indicate that the test suites produced by the enhanced algorithms have fewer OD-test failures.

	% Reduction in OD-test failures					
	Prioritization		Selection		Parallelization	
ID	Human	Auto	Human	Auto	Human	Auto
M1	100%	n/a	50%	n/a	100%	n/a
M2	100%	n/a	100%	n/a	-	n/a
M3	100%	n/a	-25%	n/a	-	n/a
M4	100%	n/a	60%	n/a	100%	n/a
M5	-	60%	0%	100%	100%	82%
M6	100%	-	-	-	-	100%
M7	100%	57%	-	-	-	12%
M8	-	n/a	-	n/a	-	n/a
M9	100%	n/a	92%	n/a	-	n/a
M10	100%	-	100%	-	-	100%
M11	-	56%	-	-	-	29%
Total	100%	57%	79%	100%	100%	66%

Although the time to compute test dependencies is substantially more than the time to run the test suite, we can see from tables 7.4 and 7.7 that the time between versions is still much more than the time to compute test dependencies. For example, while it takes about half an hour, on average, to compute test dependencies per OD test in M3's human-written test suite, the average number of days between the versions of M3 is about 10 days, thus still giving developers substantial time in between versions to compute test dependencies. Although such a case does not occur in our evaluation, even if the time between v_i and v_{i+1} is less than the time to compute test dependencies, the computation can start running on v_i while traditional regression testing algorithms (that may waste developers' time due to OD-test failures) can still run on v_{i+1}. Once computation finishes, the enhanced regression testing algorithms can start using the computed test dependencies starting at the current version of the code (e.g., version v_{i+n} when the computation starts on v_i). As we show in Section 7.3.3, these test dependencies are still beneficial many versions after they are computed.

7.3.3 Reducing Failures

Table 7.8 shows the reduction in the number of OD-test failures from the orders produced by the enhanced algorithms compared to those produced by the unenhanced algorithms. We denote cases that have no OD tests (as we find in Section 7.1.4) as "n/a", and cases where the unenhanced algorithms do not produce an order that causes any OD-test failures

as "-". Higher percentages indicate that the enhanced algorithms are more effective than the unenhanced algorithms at reducing OD-test failures.

Concerning human-written tests, we see that enhanced prioritization and parallelization algorithms are very effective at reducing the number of OD-test failures. In fact, the enhanced prioritization and parallelization algorithms reduce the number of OD-test failures by 100% across all of the evaluation projects. For test selection, the algorithms reduce the number of OD-test failures by 79%. This percentage is largely influenced by M3, where the enhanced selection orders surprisingly lead to *more* failures than the unenhanced orders as indicated by the negative number (-25%) in the table.

There are two main reasons for why an enhanced order can still have OD-test failures: (1) changes from later versions introduce new OD tests that could not have been detected in *firstVer*, and (2) the computed test dependencies from *firstVer* are incomplete; as such, the regression testing algorithms would have OD-test failures due to the missing test dependencies. In the case of (1), if this case were to happen, then both enhanced and unenhanced orders would be equally affected, and for our evaluation, we simply ignored all newly added tests. In the case of (2), it is possible that the test dependencies computed on *firstVer* are incomplete for the same OD tests on a new version, either because the missing test dependencies are not captured on *firstVer* or because the test initially is not an OD test in *firstVer* but becomes OD due to newly introduced test dependencies in the new version. In fact, for M3's human-written test selection results, we find that the enhanced orders have more OD-test failures because the enhanced orders in later versions expose a test dependency that is missing from what is computed on *firstVer*. As we describe in Section 7.3.2, to efficiently compute test dependencies on *firstVer*, we use only the orders of test prioritization and test parallelization instead of many random orders as done in prior work [85, 237]. Note that such a case occurs in our evaluation only for M3, but nonetheless, this case does demonstrate the challenge of computing test dependencies effectively and efficiently.

For automatically generated tests, the enhanced algorithms are also quite effective at reducing OD-test failures, though they are not as effective as the enhanced algorithms for human-written tests. For test selection, only M5's unenhanced orders have OD-test failures, and the enhanced orders completely remove all of the OD-test failures across all versions. Specifically, the OD test that fails is missing a positive dependee test, and the enhanced test selection algorithms would add in that missing positive dependee test into the selected tests, which prevents the OD-test failure. Similarly, the enhanced test parallelization algorithms also add in some missing positive dependee tests, leading to a reduction in OD-test failures. Once again though, there are still some OD-test failures because some test dependencies were not captured on *firstVer*.

Table 7.9: Slowdown of orders produced by the enhanced algorithms run compared to those produced by the unenhanced algorithms. Higher percentages indicate that orders produced by the enhanced algorithms are slower.

	% Time Slowdown			
	Selection		Parallelization	
ID	**Human**	**Auto**	**Human**	**Auto**
M1	9%	=	16%	7%
M2	-1%	=	9%	-8%
M3	3%	=	0%	0%
M4	5%	=	-2%	-2%
M5	=	1%	-1%	-1%
M6	-1%	=	2%	-3%
M7	6%	=	1%	0%
M8	=	=	5%	4%
M9	11%	=	2%	3%
M10	-5%	=	2%	-14%
M11	=	=	-2%	-7%
Total	1%	1%	1%	0%

In summary, we find that the orders produced by all of the enhanced regression testing algorithms collectively reduce the number of OD-test failures by 81% and 71% for human-written and automatically generated tests, respectively. When considering both human-written and automatically generated tests together, the enhanced regression testing algorithms produce orders that reduce the number of OD-test failures by 80% compared to the orders produced by the unenhanced algorithms.

7.3.4 Efficiency

To accommodate test dependencies, our enhanced regression testing algorithms may add extra tests to the orders produced by the unenhanced test selection or parallelization algorithms (T_a on Line 4 in Figure 7.2). The added tests can make the orders produced by the enhanced algorithms run slower than those produced by the unenhanced algorithms. Table 7.9 shows the slowdown of running the orders from the enhanced test selection and parallelization algorithms. We do not compare the time for orders where the enhanced and unenhanced algorithms produce the exact same orders, because both orders should have the same running time modulo noise. We mark projects that have the same orders for enhanced and unenhanced with "=" in Table 7.9. For parallelization, we compare the runtime of the longest running subsuite from the enhanced algorithms to the runtime of the longest running subsuite from the unenhanced algorithms. For each of the projects in Table 7.9, we compute the percentage by summing up the runtimes for all of the enhanced orders, subtracting the summed up runtimes for all of the unenhanced orders, and then dividing the summed up

runtimes for all of the unenhanced orders. The percentage in the final row is computed the same way except we sum up the runtimes for the orders across all of the projects.

Overall, we see that the slowdown is rather small, and the small speedups (indicated by negative numbers) are mainly due to noise in the tests' runtime. For test parallelization of automatically generated tests, we do observe a few cases that do not appear to be due to noise in the tests' runtime though. Specifically, we see that there are often speedups even when tests are added to satisfy test dependencies (e.g., M10). We find that in these cases the enhanced orders are faster because the OD-test failures encountered by the unenhanced orders actually slow down the test suite runtime more than the tests added to the enhanced orders. This observation further demonstrates that avoiding OD-test failures is desirable, because doing so not only helps developers avoid having to debug non-existent faults in their changes, but can also potentially speed up test suite runtime.

The overall test suite runtime slowdown of the enhanced algorithms compared to the unenhanced ones is 1% across all orders produced and across all types of tests (human-written and automatically generated) for all evaluation projects.

7.3.5 Findings

Our evaluation suggests the following two main findings.

Reducing Failures. The enhanced regression testing algorithms can reduce OD-test failures by 81% for human-written test suites. The enhanced algorithms are somewhat less effective for automatically generated test suites, reducing OD-test failures by 71%, but the unenhanced algorithms for these test suites generally cause fewer OD-test failures. Across all regression testing algorithms and test suites, the enhanced algorithms produce orders that cause 80% fewer OD-test failures than the orders produced by the unenhanced algorithms.

Efficiency. Our enhanced algorithms produce orders that run only marginally slower than those produced by the unenhanced algorithms. Specifically, for test selection and test parallelization, the orders produced by the enhanced algorithms run only 1% slower than the orders produced by the unenhanced algorithms.

7.4 DISCUSSION

7.4.1 General Approach vs. Customized Algorithms

In our work, we focus on a general approach for enhancing existing regression testing algorithms. Our approach works on any output of these existing regression testing algorithms

to create an enhanced order that satisfies the provided test dependencies. While our approach works for many different algorithms that produce an order, it may not generate the most optimal order for the specific purpose of the regression testing algorithm being enhanced.

For example, we enhance the orders produced by test parallelization algorithms by adding in the missing tests for an OD test to pass on the machine where it is scheduled to run. A more customized test parallelization algorithm could consider the test dependencies as it decides which tests get scheduled to which machines. If the test dependencies are considered at this point during the test parallelization algorithms, then it could create faster, more optimized scheduling of tests across the machines. However, such an approach would be specific to test parallelization (and may even need to be specialized to each particular test parallelization algorithm) and may therefore not generalize to other regression testing algorithms. Nevertheless, it can be worthwhile for future work to explore how customized algorithms can enhance traditional regression testing algorithms.

7.4.2 Cost to Provide Test Dependencies

Developers may create OD tests purposefully to optimize test execution time by doing some expensive setup in one test and have that setup be shared with other, later-running tests. If developers are aware that they are creating such tests, then the human cost for providing test dependencies to our enhanced algorithms is low.

If developers are not purposefully creating OD tests, and they are unaware that they are creating OD tests, then it would be beneficial to rely on automated tools to discover such OD tests for them and use the outputs of these tools for our enhanced algorithms, as we demonstrate in our evaluation. The cost to automatically compute test dependencies (machine cost) is cheaper than the cost for developers to investigate test failures (human cost). Herzig et al. [80] quantified human and machine cost. They reported that the cost to inspect one test failure for whether it is a flaky-test failure is $9.60 on average, and the total cost of these inspections can be about $7 million per year for products such as Microsoft Dynamics. They also reported that machines cost $0.03 per minute. For our experiments, the longest time to compute test dependencies for an OD test is for M3, needing about half an hour, which equates to about $0.90.

7.4.3 Removing OD Tests

OD-test failures that do not indicate faults in changes are detrimental to developers in the long run, to the point that, if these failures are not handled properly, one might wonder

why a developer does not just remove these OD tests entirely. However, it is important to note that these tests function exactly as they are intended (i.e., finding faults in the code under test) when they are run in the original order. Therefore, simply removing them would mean compromising the quality of the test suite to reduce OD-test failures, being often an unacceptable tradeoff. Removing OD tests is especially undesirable for developers who are purposefully writing them for the sake of faster testing [175], evident by the over 270k times JUnit annotations or TestNG attributes to control the ordering of tests have been used on GitHub as of June 2021. As such, we hope to provide support for accommodating OD tests not just for regression testing algorithms but for a variety of different testing tasks. We believe that our work on making regression testing algorithms dependent-test-aware by taking test dependencies into account is an important step in this direction.

7.4.4 Evaluation Metrics

In our evaluation, we focus on the reduction in the number of OD-test failures in enhanced orders over unenhanced orders as well as the potential increase in testing time due to the additional tests that we may need for test selection and test parallelization (Section 7.3.4). Concerning test prioritization algorithms, prior work commonly evaluates them using Average Percentage of Faults Detected (APFD) [171, 227]. Traditionally, researchers evaluate the quality of different test prioritization algorithms by seeding faults/mutants into the code under test, running the tests on the faulty code, and then mapping what tests detect which seeded fault/mutant. To compare the different orders produced by the different test prioritization algorithms, researchers would measure APFD for each order, which represents how early an order has a test that detects each fault/mutant.

In our work, we use real-world software that does not have failing tests due to faults in the code, ensured by choosing versions where the tests pass in the original order (Section 7.1.3). We do not seed faults/mutants as we want to capture the effects of real software evolution. As such, we do not and cannot measure APFD because there would be no test failures due to faults in the code under test; APFD would simply be undefined in such cases. Any test failures that we observe would be either OD-test failures or test failures due to other sources of flakiness.

7.5 THREATS TO VALIDITY

A threat to validity is that our evaluation considers only 11 modules from 8 Java projects. These modules may not be representative causing our results not to generalize. Our approach

might behave differently on different programs, such as ones from different application domains or those not written in Java.

Another threat to validity is our choice of 12 traditional regression testing algorithms. Future work could evaluate other algorithms based on static code analysis, system models, history of known faults, test execution results, and so forth. Future work could also enhance other techniques that execute tests out of order, such as mutation testing [181, 233, 234], test factoring [44, 94, 95, 156, 177, 224], and experimental debugging techniques [193, 230, 235].

Another threat is the presence of NOD flaky tests when we compute test dependencies. NOD tests may also affect the metrics that we use for the regression testing algorithms (e.g., coverage and timing of tests can be flaky), thereby affecting the produced orders. We mitigate this threat by filtering out NOD flaky tests through the rerunning of tests in the original order. We also suggest that developers use tools [85, 184] to identify these tests and remove or fix them; a developer does not gain much from a test whose failures they would ignore. In future work, we plan to evaluate the impact of these issues and to improve our algorithms to directly handle these NOD tests.

7.6 SUMMARY

Test suites often contain OD tests, but traditional regression testing techniques ignore test dependencies. In this work, we have empirically investigated the impact of OD tests on regression testing algorithms. Our evaluation results show that 12 traditional, dependent-test-unaware regression testing algorithms produce orders that cause OD-test failures in 82% of the human-written test suites and 100% of the automatically generated test suites that contain OD tests. We have proposed a general approach that we then use to enhance the 12 regression testing algorithms so that they are dependent-test-aware. We have made these 12 algorithms and our general approach publicly available [5]. Developers can use the enhanced algorithms with test dependencies manually provided or automatically computed using various tools, and the enhanced algorithms are highly effective in reducing the number of OD-test failures. Our proposed enhanced algorithms produce orders that result in 80% fewer OD-test failures, while being 1% slower to run than the unenhanced algorithms.

CHAPTER 8: [TAMING] ACCOMMODATING ASYNC-WAIT FLAKY TESTS

This chapter presents our work on FaTB, a technique developed with Microsoft collaborators to speed up async-wait flaky tests while also reducing their spurious failures. To better understand the problem of flaky tests at Microsoft, we begin this chapter by studying the flaky tests in six large-scale, diverse proprietary projects at Microsoft. Specifically, we study the prevalence, reproducibility, characteristics (e.g., reoccurrence, runtimes), categories, and resolution (e.g., time-before-fix) of flaky tests. Our study of prevalence and reproducibility reveals the substantial negative impact that flaky tests have on developers at Microsoft, while our study on the characteristics, categories, and resolution of flaky tests confirms that some of the findings from a study on open-source projects also hold for proprietary projects. For example, similar to a prior study [126] on flaky tests in open-source projects, we also find that the most common category of flaky tests in proprietary projects is the AW category. Example of an AW test is presented in Section 1.2.2.

Realizing the substantial impact that flaky tests have on Microsoft developers and how common AW flaky tests are, we propose an automated solution called FaTB, to accommodate the negative impact of these tests. The remainder of this chapter is organized as follows. Section 8.1 presents some background on Microsoft's flaky-test management system, and Section 8.2 presents the setup of our study. Section 8.3 presents our analysis of the results and our work on FaTB to accommodate async-wait flaky tests. Finally, Section 8.4 presents our threats to validity, and Section 8.5 concludes this chapter.

8.1 BACKGROUND ON MICROSOFT'S FLAKY-TEST MANAGEMENT SYSTEM

After the work presented in Chapter 5, Microsoft developers created a comprehensive flaky-test management system called *Flakes* and integrated Flakes as part of CloudBuild. Section 5.1 presents more details about Microsoft's CloudBuild system. Flakes includes four major features: Detection, Reporting, Suppression, and Resolution. The *Detection* feature aims at labeling flaky tests among all tests executed by CloudBuild. More specifically, whenever there is a test failure, CloudBuild automatically retries the test once by default, and if the retry passes, then the test is considered flaky and the build continues. Once a test is considered flaky, Flakes proceeds to *Reporting* where it reports the flaky test to developers by automatically creating a bug report. These bug reports help notify developers of the flaky tests and encourage the developers to fix the flaky tests. Note that Flakes will link multiple flaky tests to the same bug report by looking for similarities in the flaky tests' error messages.

Table 8.1: Statistics of the projects with flaky tests using Flakes during July 2019 (over a 30-day period).

Project	# Tests	# Failed Builds	# Test executions	Median build time (min)	# Flaky-test failures	# Builds with flaky-test failures	Project purpose
ProjA	7,281	2,127	2,045,513	29	157	68 (3.2%)	Ads
ProjB	29,589	13,025	45,063,356	21	17,064	1,133 (8.7%)	Cloud computing
ProjC	2,866	2,047	1,429,295	3	24	22 (1.1%)	Engr. infrastructure
ProjD	3,182	9,371	28,557,302	10	39	35 (0.4%)	Database
ProjE	7,939	847	4,702,325	11	734	302 (35.7%)	Engr. monitoring
ProjF	4,197	491	332,557	27	1,775	133 (27.1%)	Search

Doing so prevents tests with the same root cause from creating many different bug reports.

For *Suppression*, Flakes updates a suppression file that lists all known flaky tests within the project. Specifically, Flakes adds information (e.g., error message, bug report URL, code version, fully qualified test name) about a test to the suppression file when a test is found to be flaky. This suppression file is primarily used to suppress future failures of flaky tests, since they are known to be flaky already. Flakes does not prevent the running of suppressed tests but will label these failures to help reduce developers' effort in diagnosing test failures due to flaky tests. To discourage developers from fully relying on suppressions to deal with flaky tests, Flakes simply suppresses the failures for 30 days by default. Finally, for *Resolution*, when developers close a bug report related to a flaky test, Flakes automatically removes the test from the suppression file. If the test is found to be flaky later, Flakes will reopen a new bug report and repeat all of the steps above. As of August 2019, Flakes was used by 11 projects in total at Microsoft. Of these 11 projects, Flakes had already detected at least one flaky test in six of the projects. Across all of these projects, Flakes had created over 4000 bug reports and between May to August 2019, Flakes suppressed over 218000 flaky-test failures for these six projects.

8.2 STUDY SETUP

This section describes the projects and the three datasets of flaky tests we use in our study, the research questions of our study, and how we use the datasets for each question.

8.2.1 Evaluation Projects

Table 8.1 provides some statistics collected during July 2019, over a 30-day period, for six projects that use Flakes. Each of these projects has at least one flaky test in it currently or had one sometime in its past. Due to company confidentiality reasons, the names of the projects are anonymized. None of the authors of the paper [107] worked on any of the

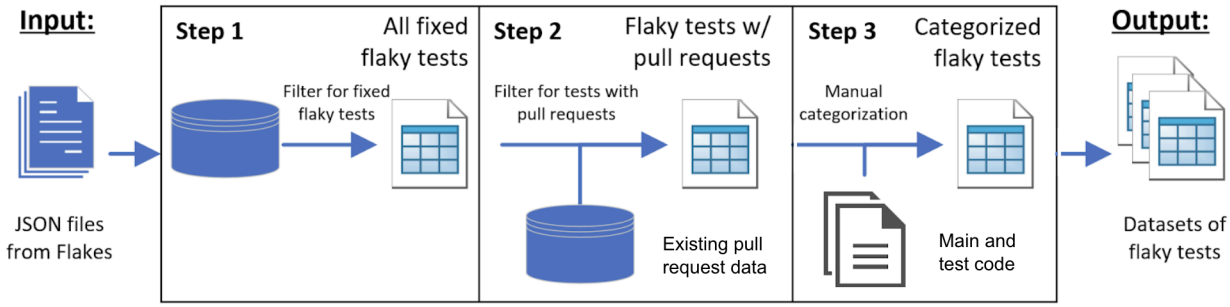


Figure 8.1: Overview of how we obtain the three datasets used in our study.

projects that uses Flakes. In the table, Column 2 shows the number of distinct tests in each project. Column 3 shows the number of failed builds for each project. Column 4 shows the number of tests executed in all builds. Note that CloudBuild does not execute all tests in each build; rather, it executes only those tests that are within the modules impacted by the change. Column 5 presents the median time of each build in minutes, and Column 6 shows the number of total flaky-test failures suppressed by Flakes. Column 7 shows the number and percentage of failed builds that contained at least one flaky-test failure suppressed by Flakes. Note that each of these builds can have more than one flaky-test failure. Finally, Column 8 shows the purpose of the project. As this table shows, the projects that use Flakes and have at least one flaky test are quite diverse. Specifically, the median build times for these projects vary from 3 to 29 minutes, and they all have distinct purposes.

8.2.2 Datasets

We conduct our study of flaky tests at Microsoft using three datasets. Figure 8.1 shows an overview for how we obtain these three datasets from the six projects that use Flakes. As Figure 8.1 shows, we obtain the datasets using three main steps, with each subsequent step using some or all of the data in the previous step. To obtain our datasets, we start with all versions of the suppression files that Flakes maintains for each of the six projects. These suppression files are version-controlled, and Flakes uses them to keep track of known flaky tests. Having previous versions of these suppression files consequently allows us to find flaky tests that Flakes found in the past regardless of whether these tests are fixed or not. In total, Flakes identified 2089 flaky tests from the entire history of the six projects shown in Table 8.1.

We use the suppression files maintained by Flakes for each project to create three datasets labeled as All-Fixed, Pull-Requests, and Categorized. Dataset *All-Fixed* includes all flaky tests that Flakes has observed to be fixed and contains 1040 flaky tests. Dataset *Pull-*

Requests includes all flaky tests that are fixed and the bug report associated with the flaky test includes a pull request that the developer manually linked to the bug report. This dataset contains 134 flaky tests. Lastly, dataset *Categorized* includes all flaky tests that have pull requests and, upon our manual investigation of the pull requests, bug reports, and main and test code, we categorize these flaky tests with the categories defined in a prior study [126]. This dataset also contains 134 flaky tests.

To obtain the All-Fixed dataset (the result of Step 1), we parse the suppression files of Flakes into a SQL-like database known as Azure Data Explorer [13]. We parse the suppression files from the oldest to the newest version, and when we see a flaky test get added to the file, we consider that test to be flaky. On the other hand, when we see a flaky test get removed from the file, we consider that test to be fixed.[1] The number of times a flaky test is detected and fixed depends on the number of times the test is added and removed (respectively) from the suppression file.

To obtain the Pull-Requests dataset (the result of Step 2), we join the All-Fixed dataset with an existing Azure Data Explorer table that keeps track of which pull requests, if any, are linked to a bug report. Not all closed bug reports are linked to a pull request, because developers have to manually link the pull requests themselves. We join the All-Fixed dataset with an existing Azure Data Explorer table because Flakes keeps track only of the bug report it creates for a particular flaky test, and would not otherwise know if a particular bug report has pull request(s) linked to it. Note that by design ProjB's bug reports are not accessible through Azure Data Explorer. Therefore, all bug report and pull request information for the flaky tests of ProjB is omitted from our study.

To obtain the Categorized dataset (the result of Step 3), we study the pull request, main code, and test code of each flaky test in the Pull-Requests dataset. Each flaky test that we categorize is verified independently by two or more of the authors of our paper [107]. Our categorization considers four kinds of locations and 10 root-cause categories that we obtain from a prior study [126] on flaky tests.

Table 8.2 summarizes for each project the flaky tests we find in it. Overall, Flakes identified 2089 flaky tests from six projects. On average, Flakes has been tracking flaky test information in these six projects for 181 days. Of the 2089 flaky tests, 1040 tests are fixed. Of the 1040 flaky tests that are fixed, we find that the developers attached a pull request to the bug report for 134 tests.

[1] A flaky test can also be removed from the suppression files because the test is removed from the project. Our All-Fixed dataset does include such tests.

Table 8.2: Flaky-test statistics of the projects in our study. *ProjB's pull requests (PRs) are inaccessible for our study.

Project	# Flaky tests	# Fixed flaky tests	# Flaky tests with PRs
ProjA	10	3	2
ProjB	352	31	*0
ProjC	73	63	8
ProjD	1453	878	96
ProjE	176	64	27
ProjF	25	1	1
Total	**2089**	**1040**	**134**

8.2.3 Research Questions

To better understand the lifecycle of flaky tests at Microsoft, we study the prevalence, reproducibility, characteristics, categories, and resolution of flaky tests. More specifically, we address the following research questions:

RQ1 [**Prevalence**]: How prevalent are flaky tests and to what extent do they impact developers' workflow?

RQ2 [**Reproducibility**]: How many runs are needed to reproduce flaky-test failures?

RQ3 [**Characteristics**]: Does test flakiness reoccur after fixes? If so, what are the reasons for it to reoccur?

RQ4 [**Characteristics**]: How does the runtime of a flaky test differ between passing and failing runs?

RQ5 [**Categories**]: What are the categories (e.g., root cause, location) of the flaky-test fixes?

RQ6 [**Resolution**]: How much time do developers take to fix flaky tests?

RQ7 [**Resolution**]: How effective are developers at identifying and fixing the timing-related async-wait issues in flaky tests?

We first address RQ1 to understand how problematic flaky tests are at Microsoft. We then address RQ2 to understand the difficulty developers may have in debugging and fixing flaky tests. Knowing the difficulty of reproducing flaky-test failures, we then address RQ3 and RQ4 to understand the characteristics of these flaky tests. We then address RQ5 to extend our characteristics study by categorizing the location and root cause of the flaky-test fixes in our dataset. Lastly, we address RQ6 and RQ7 to understand how effective developers are at fixing flaky tests.

8.2.4 Methodology

All of our research questions use either the All-Fixed, Pull-Requests, or Categorized datasets of flaky tests described in Section 8.2.2.

RQ1: Prevalence and Impact of Flaky Tests. For RQ1, we use the All-Fixed dataset, our entire dataset of fixed flaky tests. For this RQ, we rely on the information collected by Microsoft's Flakes during July 2019. Specifically, we look at the number of failed builds that would have occurred due to flaky-test failures if Flakes did not suppress such failures from these builds.

RQ2 and RQ4: Reproducibility and Runtime of Flaky Tests. For RQ2 and RQ4, we again start by using the All-Fixed dataset. Specifically, for each flaky test, we run the test 500 times in the actual build and testing environment. We use only three projects (ProjC, ProjD, ProjE), because we perform these experiments on real proprietary projects, and we cannot interrupt or slow the actual testing environments of the other projects.

RQ3: Reoccurrence of Flaky Tests. For RQ3, we use the All-Fixed dataset, but we filter for tests that have been fixed more than once. We identify the flaky tests that are fixed more than once by looking for tests that are removed from a project's suppression file more than once. For the tests that are fixed more than once, we look at the commits, bug reports, and main and test code to understand why they reoccur. We use the commits instead of pull requests because not all tests in the All-Fixed dataset have pull requests linked to the tests' bug reports. Indeed, we find that for the flaky tests that we study for this RQ, all of their bug reports do not have pull requests. We obtain the commits for these tests, by using the dates of their bug reports and the version-control history of the test code to find the likely commits for these tests. We then confirm these commits with the developers of the flaky tests.

RQ5: Categories of Flaky-Test Fixes. For RQ5, we use the Pull-Requests dataset, which contains 134 flaky tests that are fixed and have a pull request associated with the test. Specifically, we study (1) where the changes are located in (i.e., main or test code), and (2) what root causes of flaky-test fixes do these pull requests belong to? A prior study [126] on flaky tests found four kinds of locations in which fixes were located in and also identified 10 root causes of flaky-test fixes. For our study, we manually label each pull request along with its corresponding bug report and main/test code with the same four kinds of locations and 10 root causes as the prior study. We decide to use pull requests, which consist of one or more

commits, because pull requests represent a more complete set of changes. These changes from pull requests generally build without errors and have been tested on the developers' machines to ensure that they do not fail any tests.

RQ6: Time-before-fix of Flaky Tests. For RQ6, we use the All-Fixed dataset. Specifically, for each fixed test, we study the bug report linked to the test. Recall that Flakes' Reporting feature, as described in Section 8.1, will automatically create a bug report for each test it finds to be flaky. To obtain the time-before-fix of flaky tests, we study the time the bug reports of these tests took from being created to them being closed.

RQ7: Developers' Effectiveness in Identifying and Fixing Async Wait Tests. For RQ7, we use the Categorized dataset, which contains 134 flaky tests that are fixed, have a pull request associated with the test, and its pull request, bug report, and code are categorized. Since in RQ5 we find that Async Wait is the most common category of flaky tests, we focus specifically on this category for RQ7. To understand how effective developers are at identifying and fixing async-wait tests, we sample five async-wait tests whose fix by developers is to increase the wait/timeout. We then calculate the flaky-test-failure rate with the developer-suggested fix and measure how the rate changes when the time value increases or decreases.

8.3 ANALYSIS OF THE RESULTS

This section presents the results of our study for the research questions in Section 8.2.3 using the methodology we describe in Section 8.2.4. An anonymized version of the data we use for our study is available online [34].

8.3.1 RQ1: Prevalence and Impact of Flaky Tests

We begin our study by first investigating how prevalent flaky tests are at Microsoft. From tables 8.1 and 8.2, we see that for all projects except for ProjD, the number of flaky tests ever found is only a small fraction of the total number of tests these projects' have during the month of July 2019. However, just because a project contains many or few flaky tests, it does not necessarily mean that the developers' workflow is often or rarely impacted by these tests. To understand whether these flaky tests do impact developers' workflow or not, we also show in Table 8.1 the percentage of developers' builds in which the build would have failed due to flaky-test failures if Flakes did not suppress such failures. One interesting point

to note here is that even though some projects have lots of flaky tests, these projects' chance for builds to fail due to flaky-test failures are not particularly high. For example, ProjD has 1453 flaky tests with over half still not fixed, but flaky-test failures only affect 0.4% of its builds over a 30-day period, while ProjE has only 176 flaky tests but 35.7% of its builds are affected by flaky-test failures during the same 30-day period. Our results demonstrate that, although flaky tests may not always be very prevalent, the percentage of builds that are impacted by flaky-test failures can still be quite substantial.

8.3.2 RQ2: Reproducibility of Flaky-Test Failures

One of the biggest challenges developers have when debugging or fixing flaky tests is to reproduce the flaky-test failure. To understand how much of an imposition the reproducibility of flaky-test failures may have on developers, we study the number of flaky tests in which we can reproduce the flaky-test failures, and for the tests where we can reproduce the flaky-test failure, we also study these tests' flaky-test-failure rates. For each of the flaky tests that we use for this RQ, we run the test 500 times using the same configuration of the machines and the version of code on which Flakes detected the test to be flaky.

Table 8.3 summarizes our results. We use only a subset of our dataset for this RQ, because these experiments are performed on real proprietary projects, and we could not slow down the actual testing environment of the other projects. Furthermore, not all flaky tests can be run again due to problems compiling the version of code on which the test was detected. The actual number of flaky tests that we are able to run 500 times is shown in Column 2. Column 3 shows the number of tests that pass and fail at least once, and Columns 4 and 5 show the average and median (respectively) percentage rate of flaky-test failures for the flaky tests that pass and fail at least once.

Column 2 of Table 8.3 shows that flaky-test failures are reproducible between 17% to 43%, depending on the project. This finding suggests that there are many flaky tests (up to 83%, depending on the project) where even with 500 runs, we cannot reproduce the flaky-test failures of these tests. We also see from Column 4 that the median percentage rate of flaky-test failures can be quite low, particularly for ProjC and ProjE. This finding suggests that even when flaky-test failures can be reproduced in 500 runs, only a small number of runs results in a failure.

Figure 8.2 shows a box plot for the percentage rate of flaky-test failures for the flaky tests that pass and fail at least once in each project. We can see in this figure that the averages of both ProjD and ProjE (Column 3 in Table 8.3) are quite high due to the percentage rates of 23 outlier tests. To understand these outliers better, we analyze them and find

Table 8.3: Statistics on reproducibility of flaky-test failures.

Proj.	# Flaky tests	# Flaky tests 1+ pass & fail	Average % fail	Median % fail
ProjC	7	3 (43%)	0.2	0.2
ProjD	545	95 (17%)	36.8	29.4
ProjE	85	21 (25%)	9.7	0.6

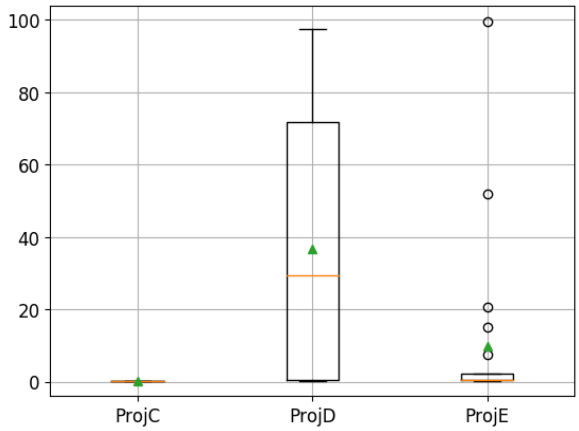


Figure 8.2: Percentage rate of flaky-test failures.

that 9 of these tests are likely Async Wait—their names contain "Async". Of these 9 tests, we have the pull request, bug report, and code for 3 of the tests, and while studying the categorization of flaky tests in RQ5 (Section 8.3.5), we do indeed categorize these 3 tests as async-wait tests.

8.3.3 RQ3: Reoccurrence of Test Flakiness

As Section 8.3.2 demonstrates, flaky-test failures can be quite difficult for developers to reproduce. This difficulty makes it so that when developers are fixing flaky tests, they may just assume that their changes fixed the flaky-test failure and they may not actually confirm their assumption. For our study on the reoccurrence of flaky tests, we use the All-Fixed dataset, which contains 1040 fixed flaky tests. Of these 1040 flaky tests, we find that four flaky tests are fixed more than once.

If a flaky test is found to be flaky more than once, then it is either because (1) the developers' initial fix for the flakiness was inadequate, or (2) the cause of flakiness was reintroduced. In either case, sufficient time must be given for the developers to notice the reoccurrence and for the test to run many times for it to fail builds again. On average, the fixed flaky tests in our study have been fixed for more than 86 days.

To understand why four flaky tests had to be fixed more than once, we manually investigate

the commits, bug reports, and the main and test code for each flaky test. We use commits instead of pull requests because not all tests in the All-Fixed dataset have pull requests, and all four tests that are fixed more than once do not have pull requests. We obtain the commits for these tests by using the time their bug reports closed and the version-control history of the test code to find the likely commits for these tests. We then confirm the likely commits with the developers of the four tests.

Our investigation into the four flaky tests that reoccur more than once reveals that all four reoccurrences are due to case (1), i.e., the developers' initial fix was inadequate. For these flaky tests, we can confirm this case because the developers described their initial fix as being inadequate in their latest fix. For example, one developer described the latest fix as "Increase the wait time for idle timeout test case to ensure that the receive loop exits first. Previously, the wait time was 1 second more than the idle timeout, which was cutting it too fine". Overall, our investigation into the reoccurrence of these four tests finds that developers have the following important sentiments about fixing flaky tests.

1. Developers are rarely able to reproduce the flaky-test failures locally on their own machines or on servers.

2. As a consequent of (1), developers often resort to making multiple changes to fix test flakiness. These changes are often either

   (a) made by trial-and-error guessing, and the developers rely on the frequent runs of the test on the servers to determine whether the fix was adequate, or

   (b) made simply to log additional information so that the developers can know more about the flaky-test failure before attempting a real fix.

8.3.4 RQ4: Runtime of Flaky Tests

To begin understanding why a test may be flaky, we study the runtime of flaky tests. We study the runtime because when flaky tests fail, they may run faster than when they pass, because the test may have encountered a fault and stopped early. However, flaky tests may also take longer in their failing runs if they are flaky because they time out. In such cases, the flaky test may wait for a callback that simply never happens, indicating that these tests likely make asynchronous calls. Similar to RQ2 (Section 8.3.2), we can use only a subset of our dataset for this RQ since we could not slow down the testing environment of the other projects, and the version of code in which some flaky tests were detected no longer compiles.

Table 8.4: Statistics on runtime (in seconds) of flaky tests.

Proj.	Test result	# Runs	Average runtime	Median runtime
ProjC	Pass	1,497	2.39	1.34
ProjC	Fail	3	2.47	1.48
ProjD	Pass	30,023	9.41	4.31
ProjD	Fail	17,477	22.25	25.00
ProjE	Pass	9,477	2.14	1.61
ProjE	Fail	1,023	1.72	0.80

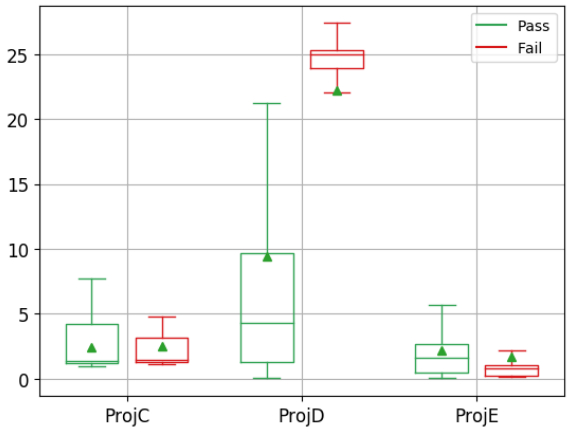


Figure 8.3: Runtime (in seconds) of flaky tests.

Table 8.4 and Figure 8.3 show the runtime in seconds of the flaky tests that pass and fail at least once in 500 runs. Overall, we see that for ProjE, the average and median runtime of passing runs is more than failing runs. As for ProjC, we see that the average and median runtime of passing runs is about the same as the failing runs. This result suggests that for these two projects, their flaky tests are likely unrelated to asynchronous method calls. On the other hand, we can see that for ProjD, the average and median runtime of failing runs is substantially more than the runtime of passing runs. This result suggests that ProjD's flaky tests are likely related to asynchronous method calls. When we categorize flaky tests in RQ5, we do indeed find that the majority of ProjD's flaky tests are async-wait tests.

8.3.5 RQ5: Categories of Flaky-Test Fixes

To understand the categories of flaky-test fixes, we use the Pull-Requests dataset, which contains 134 fixed flaky tests that all have associated pull requests. We categorize these tests by studying their pull requests, bug reports, and main and test code. Our study focuses on two main questions; (1) where were the majority of the changes located in (i.e., main or test code), and (2) what root causes of flaky-test fixes do these pull requests belong to? A prior

Table 8.5: Comparison of flaky-test-fix categories with a prior study [126].

Categories	**Our study**	**[126]**
Location of fixes		
Main only	1%	12%
Test only	71%	73%
Main and Test	11%	11%
Other	17%	4%
Root cause of fixes		
Async Wait	78%	45%
Network	14%	6%
Concurrency	8%	20%
Resource Leak	5%	7%
Randomness	5%	2%
IO	5%	2%
Time	4%	3%
Floating Point Operations	2%	2%
Test Order Dependency	0%	12%
Unordered Collections	0%	1%
Difficult to categorize	26%	20%

study [126] on flaky tests found four kinds of location in which fixes were located and also identified 10 root causes of flaky-test fixes. Table 8.5 summarizes our findings and those of a prior study.

Location of Flaky-Test Fixes. A prior study [126] on flaky tests identified four kinds of locations for flaky-test fixes; (1) Main code only, (2) Test code only, (3) Main and Test code, and (4) Configuration. We adopt similar kinds of locations for our study. The only change we make is that "Configuration" is changed to "Other" instead, and a fix is considered to be "Other" if it includes changing anything besides main or test code (e.g., test input data, configuration).

Overall, our findings confirm what a prior study [126] found: the majority of fixes (71%) for flaky tests are in the test code. Interestingly, we find that, in 5% of the fixes, developers simply removed the test. Our investigation shows that developers sometimes temporarily removed the failing test or claimed that the test is for functionality that is no longer supported. We also find that about 12% (1% + 11%) of fixes involve changes to main code. Overall, our results show that ignoring flaky tests can be dangerous since they do indicate faults in both main and test code.

Root Causes of Flaky-Test Fixes. When performing our study on the categories of flaky-test fixes, we use the same categories used by a prior study. We find that the most common category of fixes is Async Wait, with 78% of the fixes belonging to that category. Async Wait flaky tests make an asynchronous call, and they do not properly wait for the call to return. The second most common category is Network with 14% of the fixes. Note that unlike the prior study, one fix of ours may belong to multiple categories. Also, similar to a prior study [126], we find a number of fixes (26%) that we could not categorize due to the large number of changes. Specifically, in our study, these fixes modify an average of 785 files. When we examine such fixes in detail, we see that they were a part of a version upgrade or major refactoring.

Overall, our study differs from a prior study [126] in two main ways. (1) The prior study used flaky tests from open-source projects, while we used flaky tests from proprietary projects at Microsoft, and (2) we study the pull requests, bug reports, and main and test code of flaky tests, while the prior study [126] studied only commits. We believe these differences between our studies are responsible for the minor differences in our findings. One example of how our results differ from the prior study is the percentage of fixes that are categorized as Test Order Dependency. This difference is likely because the way we run tests at Microsoft heavily reduces the chance of Test Order Dependency causing test flakiness. As explained in Section 5.1, CloudBuild always runs tests in the same order, but this requirement is not true for the open-source projects in the prior study[2]. Indeed, as Table 8.5 shows, none of the flaky tests within the six projects we study are flaky due to Test Order Dependency, even though this category is the third most common category of flaky tests in open-source projects.

Even though the composition of our study differs from a prior study, our findings on the location and common root causes of fixes remain largely the same. Specifically, we all find that the majority of flaky-test fixes are located in test code, but a nontrivial amount of them do also involve main code (12%). Also, the most common category of flaky-test fixes is Async Wait. Our findings here suggest that solutions, like the one we propose in Section 8.3.7, that can help reduce flaky-test failures of async-wait tests would highly help accommodate the negative impact of flaky tests.

[2]Note that it is still possible for a flaky test to fail due to Test Order Dependency at Microsoft, because the flaky test could fail when a new test is added and the new test runs before the flaky test. Dually, a flaky test could start failing as well when a test that was needed to run before the flaky test is removed from the test suite.

Table 8.6: Time given in days for developers to close flaky-test or non-flaky-test related bug reports (BRs). ProjB is omitted because its BRs are inaccessible for our study.

	Flaky-test			Non-flaky-test		
Proj.	**# BRs**	**Median**	**Avg.**	**# BRs**	**Median**	**Avg.**
ProjA	2	5	5	238	3	17
ProjC	55	90	95	96	11	30
ProjD	759	6	11	1575	8	12
ProjE	43	14	39	55	10	13
ProjF	1	8	8	3	8	12
Overall	**861**	**7**	**18**	**1967**	**7**	**13**

8.3.6 RQ6: Time-Before-Fix of flaky tests

Prior work [42, 76, 80, 106, 137, 165] highlighted how flaky tests negatively impact the software development process and how important it is for developers to fix flaky tests.

To understand how developers at Microsoft view the importance of fixing flaky tests, we study how long developers take on average to fix flaky tests. More specifically, we study the time developers take on average to close the bug report linked to a flaky test. At Microsoft closing a bug report typically means that the bug has been fixed. For our study, we start with the flaky tests in our All-Fixed dataset, which consists of 1040 fixed flaky tests. We find that these 1040 flaky tests are linked to 861 bug reports. As we explain in Section 8.1, multiple flaky tests may be linked to the same bug report if these tests share similar error messages (e.g., two tests are flaky due to the same setup method).

Table 8.6 shows the average and median number of days each project takes to close flaky-test and non-flaky-test related bug reports. Note that ProjB's results are omitted from Table 8.6, because as we explain in Section 8.2.2, all bug report and pull request information for the flaky tests of ProjB are inaccessible for our study. As we show in Table 8.6, developers take, on average, 18 days, with a median of 7 days, to close flaky-test related bug reports, while they take on average 13 days, with a median of 7 days, to close all non-flaky-test related bug reports. The median times developers take to close flaky-test and non-flaky-test related bug reports are the same, suggesting that developers consider these bug reports to be of equal importance. However, when we compare the average time developers take to close flaky-test related bug reports to the non-flaky-test ones, we see that flaky-test related ones take substantially longer than non-flaky-test ones (18 days for flaky-test related ones compared to 13 days for non-flaky-test related ones). As Table 8.6 shows, part of the reason why the average for flaky-test related ones are higher is because the flaky-test related bug reports in ProjC take much longer to close than the non-flaky-test ones. When we compare the flaky-test and non-flaky-test related bug reports per project, we see that ProjA's and

Table 8.7: Categorization of Async Wait flaky-test fixes.

Categories	Tests
Timing-related fix	
Increase wait/timeout	21 (31%)
Add/improve callback	19 (28%)
Add/improve polling	7 (10%)
Timing-unrelated fix	
Removing code	17 (25%)
Mocking async calls	10 (15%)
Difficult to categorize	20 (23%)

ProjF's average time to close non-flaky-test related bug reports are actually more than the time to close flaky-test related ones. On the contrary, we also see that ProjC's and ProjE's average time to close flaky-test related bug reports are substantially more than the time to close non-flaky-test related ones. Our findings suggest that although there has been a substantial amount of work from both industry and academia on flaky tests, it can still be important to communicate to some developers the importance of fixing flaky tests. Following our study, we personally approached a number of teams (i.e., those from ProjC and ProjE) to better communicate to them this importance.

8.3.7 RQ7: Developers' Effectiveness in Identifying and Fixing Async Wait Tests

Based on the prevalence of async-wait tests as described in RQ5 (Section 8.3.5) and in a prior study on flaky tests [126], we proceed to study developers' effectiveness in identifying and fixing async-wait tests at Microsoft. To study this RQ we first categorize the Async Wait related flaky-test fixes in our dataset. In total from our work in Section 8.3.5, we find 87 flaky tests that have fixes related to asynchronous method calls.

Prior work [57] on asynchronous tests proposed three main ways one should test asynchronous code. (1) Create a "synchronous" interface between tests and asynchronous code, (2) implement callbacks on all asynchronous code, and (3) check, or poll, frequently whether an asynchronous service is complete. When we study the Async Wait flaky-test fixes (or *Async Wait fixes* for short) in our dataset, we find no cases in which (1) was done by developers. Our results are likely because (1) requires substantial effort from developers to create and maintain such interfaces. On the other hand, we do see developers using both callbacks (28% of Async Wait fixes) and polling (10% of Async Wait fixes) to fix their async-wait tests. Beyond the three categories laid out in this prior study, we also find three new categories for these Async Wait fixes. Specifically, we find that the most common category of fix (31%

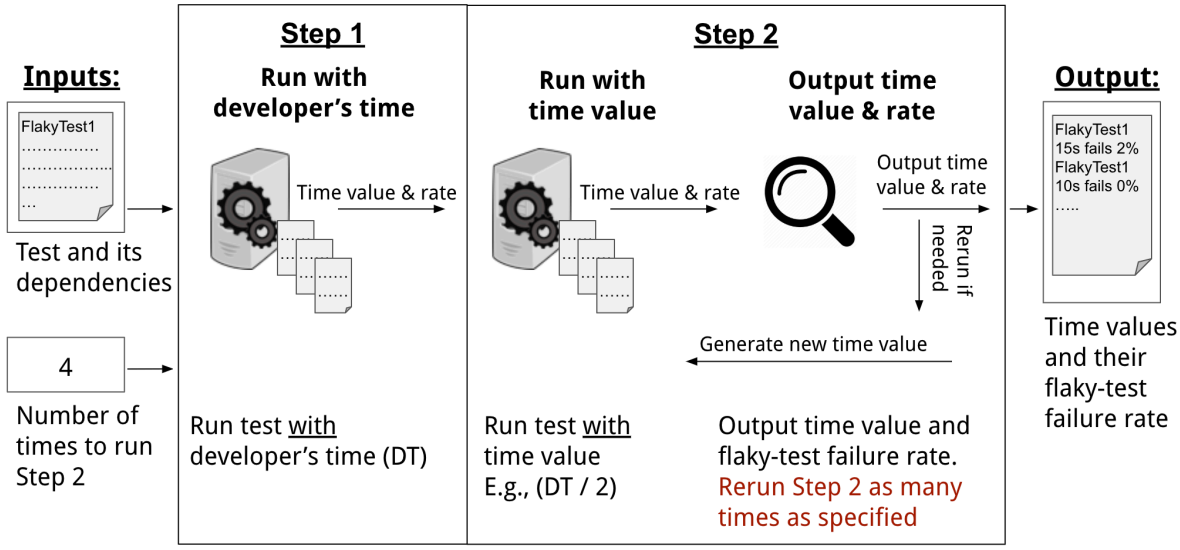


Figure 8.4: Overview of how the Flakiness and Time Balancer (FaTB) works.

of Async Wait fixes) involves simply increasing the wait time or timeout of asynchronous method calls. The other two categories are to simply remove code related to the flaky-test failure (25% of Async Wait fixes) or to mock the asynchronous calls (15% of Async Wait fixes). Lastly, we find that 23% of the Async Wait fixes are difficult to categorize, since they involve changes to asynchronous method calls, but we cannot identify any particular categories for these fixes. Table 8.7 summarizes the findings from our categorization. Note that the fix for each test may be categorized into one or more categories.

Evaluating and Improving Developers' Async Wait Fixes. Because the majority of the async-wait fixes involve simply increasing the wait time or timeout (*time value* for short) of an asynchronous call, we proceed to study how well these time values set by developers are at reducing flaky-test failures and how these time values affect the runtime of these flaky tests. To evaluate and improve developers' Async Wait fixes, we propose the *Flakiness and Time Balancer* (*FaTB*). FaTB first finds the flaky-test-failure rate of the test using the developers' fix. Once FaTB obtains the rate associated with the developers' fix, it then increases or decreases the time value and again measures the flaky-test-failure rate associated with the new time value. Figure 8.4 shows an overview for what FaTB does.

Depending on whether the test is still flaky with the new time value, FaTB will use that information to either increase or decrease the next time value to try. At a high-level, to lower the time value, FaTB will first use the time value between the developer's fix time and the time set before the fix. For an example, if the developer's fix time was 1000 milliseconds (ms) and the time value before their fix was 500 ms, then FaTB would first set the time value to be 750 ms. If lowering the time value does not cause flaky-test failures in some

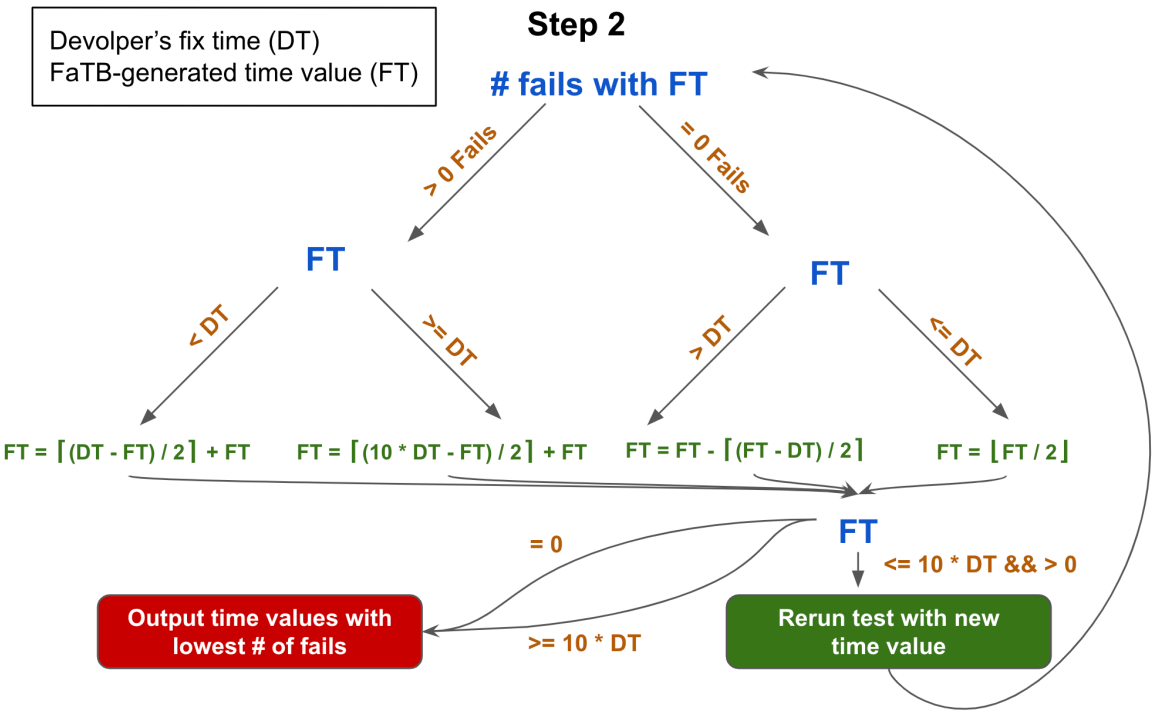


Figure 8.5: How FaTB chooses the next time value an Async Wait test should try.

number of runs (100 in our experiments), then FaTB will halve the current time value (e.g., 375 ms). If lowering the time value does cause flaky-test failures, then the next time value will be between the current value and the developer's fix time (e.g., 875 ms).

FaTB outputs the observed flaky-test-failure rate and average test runtime for each time value (e.g., [time value: 375ms, fails: 1%, runtime: 875ms], [time value: 1000ms, fails 0%, runtime: 1500ms]). As a post-processing step, FaTB will also remove time values that have the same flaky-test-failure rate, choosing to output only the minimum time value and runtime for all observed flaky-test-failure rates. This output enables developers to finely balance the trade-off of their tests' runtime and flaky-test-failure rate. The logic FaTB uses to generate different time values is shown in Figure 8.5. FaTB generates time values specific to the machine on which FaTB is run on. To ensure that these generated time values perform well on different machines, developers can run small benchmarks on these different machines (e.g., their own development machine) and the machines on which they run FaTB (e.g., a development server). The difference in the machines' performance on the small benchmarks can then be used to scale the generated time values as needed.

Results. To evaluate FaTB, we randomly sample five tests from the 21 tests whose fix was to increase the wait/timeout. The results from applying FaTB on these five flaky tests are shown in Table 8.8. Specifically, we use FaTB to generate four time values for each test,

Table 8.8: Statistics of the results produced by FaTB on five versions of five async-wait tests. The unit for the Time value depends on the test. Runtime is in seconds. Pre-fix value is the value before the developer's fix. Version 0's time value is the value after the developer's fix.

Flaky test	**Version**	**Time value**	**% Fails**	**Average runtime**	**Median runtime**
Test1	0	600	0	1.39	1.37
(Pre-	1	450	0	1.22	1.22
fix	2	225	73	1.07	1.08
value:	3	413	0	1.19	1.19
300)	4	206	82	1.10	1.07
Test2	0	1,000	0	1.75	1.75
(Pre-	1	800	0	1.61	1.54
fix	2	400	0	1.14	1.13
value:	3	200	0	0.96	0.94
600)	4	100	0	0.84	0.84
Test3	0	600	0	1.67	1.66
(Pre-	1	300	0	1.36	1.36
fix	2	150	0	1.21	1.21
value:	3	75	0	1.14	1.14
0)	4	37	0	1.09	1.08
Test4	0	100	0	8.08	8.03
(Pre-	1	50	0	7.57	7.49
fix	2	25	0	7.43	7.38
value:	3	12	0	7.39	7.30
0)	4	6	0	7.20	7.10
Test5	0	150	0	0.18	0.18
(Pre-	1	83	0	0.11	0.11
fix	2	41	0	0.07	0.07
value:	3	20	0	0.05	0.05
15)	4	10	0	0.04	0.04

and Version 0 represents the value the developers proposed to fix the flaky tests. We run the test 100 times for each version to measure that version's flaky-test-failure rate. Due to confidentiality reasons, the names of the tests are anonymized.

We apply Step 2 of FaTB four times on five flaky tests and find that for four of the flaky tests, even when the time value is set to be substantially lower than the value set by the developers to fix the test, the tests' flaky-test-failure rates appear to be unaffected. More specifically, for Test2, Test3, Test4, and Test5, we see 0% flaky-test-failure rates even when we substantially decrease the time value set by the developers. Our finding here further echoes the sentiments that we find about fixing flaky tests in Section 8.3.3. More specifically, we see that the fixes employed by the developers for these flaky tests were likely educated

guesses that turn out to be unrelated to the flaky-test failures of the test. This finding here is largely related to how truly understanding the root cause of a flaky test is often challenging. For these four flaky tests, neither we nor the developers were able to actually determine the root cause for the flaky test, and, consequently, neither we nor they are able to fix the flaky test. Future studies on the root causes of flaky tests should be more cautious when basing their results on the changes by developers.

Besides the four flaky tests that do not encounter any flaky-test failures, we see one example (Test1) that does exhibit flaky-test failures once we lower the wait/timeout value of the test. For Test1, we can see that once the value goes lower than the value before the developer fixed the flaky test (300), we start observing a high flaky-test-failure rate (e.g., when set to 225, we see a 73% flaky-test-failure rate). However, once FaTB sets the value to be 413, we see a 0% flaky-test-failure rate. When we compare the flaky-test-failure rate with the developer's fix (time value for Version 0) to the flaky-test-failure rate with the time value for Version 3, we see that this flaky test can obtain the same flaky-test-failure rate when the time value is set to 600 or 413. However, the lower time value (413) enables this test's average runtime to be about 14% faster than the higher value (600). Similarly, Test2, Test3, Test4, and Test5 do not encounter any flaky-test failures on all versions, and their average runtime can also be faster by about 52%, 35%, 11%, and 78%, respectively.

It is surprising that the developers of Test2, Test3, Test4, and Test5 would increase the time values of these tests when the values do not appear to empirically affect the tests' flaky-test-failure rate. Our results suggest that there are perhaps some other changes in these pull requests that actually fix the flaky tests, and that the changes in time values were not intended as the fix. To understand why the developers may have changed these time values, we study the pull request messages of the fixes. We find that for all four of the flaky tests besides Test4, the messages all say that they are increasing the time values as a fix to the flakiness. For example, Test3's message says "Fix is to wait 2 * X time". On the other hand, Test4's message says "Fix flaky test" and then explains how some refactoring was done to some asynchronous code. From these pull request messages, we see that at least for Test2, Test3, and Test5, these developers are purposefully trying to fix their async-wait tests by increasing time values. Note that Microsoft does not encourage developers to fix their async-wait tests by increasing the time value. However, since regulating how developers fix their code would be very costly, we do plan to use FaTB to help developers at Microsoft. With the prevalence of async-wait tests and how developers prefer to fix these tests by increasing the time value, we suspect that there are many other tests whose collective reduction in test runtime can substantially lessen the time developers spend waiting for test results, machine resources needed to run these tests, and amount of flaky-test failures developers debug.

8.4 THREATS TO VALIDITY

Our work contains many of the common threats typically found in empirical studies. In this section we focus on the issues that are more specific to our study.

Subjects of our study. Our study consists of just six projects at Microsoft, and our findings from studying these six projects may not generalize to other projects or companies. To avoid any bias in the selection of our projects, we include all projects using Flakes in our study. As we describe in Section 8.1, there are a total of 11 projects using Flakes, and of these 11 projects, the six projects we study are the ones where Flakes found at least one flaky test. The projects we study also greatly vary in activity (e.g., number of builds per month) and in purpose (e.g., database, search).

Aside from the projects used in our study, our decision to study pull requests, bug reports, and main and test code to understand the fixes of flaky tests in Section 8.3.5 may also be a threat. A prior study [126] on flaky tests used the commits to understand the characteristics of flaky tests. For our study, we use pull requests, which consists of one or more commits, because we believe that pull requests represent a more complete set of changes made by the developers. Unlike commits, changes from pull requests generally build without errors and have been tested locally to ensure they do not fail any tests. Furthermore, the prior study [126] did not run the tests and observed them to be flaky like we do.

Metrics used in our study. The metrics we use in our study pose a potential threat to our findings and results. For example, we use the time a bug report is opened till when it is closed to understand developers' sense of importance in fixing flaky tests. In reality, a developer taking a short or long amount of time to fix flaky tests could be an indicator of how easy or difficult the fix was and would be irrelevant to the developer's sense of importance. The timing we report may also be inaccurate, because different teams work differently and some teams may close bugs as soon as they are fixed, while other teams may only close them at their next team meeting.

Flaky tests used in our study. Flaky tests, by definition, may pass or fail on the same code. To identify the flaky tests used in our study, we rely on Flakes, which simply reruns each failing test once to see whether it would pass on the rerun. Because there are no guarantees that the rerun would pass if the test is indeed flaky, Flakes may potentially contain many false negatives, in which a test that is flaky is undetected. Nevertheless, Flakes contains no false positives, meaning that all flaky tests detected by Flakes must indeed be flaky. Due to this threat, the number of flaky tests we report in our study is simply the minimal number of flaky tests in our projects.

Our findings in regards to the runtime and reproducibility of flaky tests identify that

there are some patterns. However, more runs of the flaky tests may change our findings. To mitigate this threat, we choose to run each test a high number of runs, specifically 500 runs.

Findings from manual inspection. Certain research questions in our study require us to manually inspect the information of flaky tests. Specifically, for categorizing flaky tests and categorizing async-wait tests, we minimize the occurrence of miscategorization by having more than one author of our paper [107] inspect every pull request, every bug report, and all main and test code, and we discussed our categorizations until everyone agrees.

8.5 SUMMARY

Flaky tests have received much attention from both the industry and the research community in recent years. Although flaky tests are the focus of several existing studies, none of them study (1) the reoccurrence, runtimes, and time-before-fix of flaky tests, and (2) flaky tests in-depth on proprietary projects. To fill this knowledge gap, we study the lifecycle of flaky tests on six large-scale, diverse proprietary projects at Microsoft. Our study of prevalence and reproducibility reveals the substantial negative impact that flaky tests have on developers at Microsoft, while our study on the characteristics, categories, and resolution of flaky tests confirms that some of the findings from a study on open-source projects also hold for proprietary projects. For example, similar to the prior study on flaky tests in open-source projects, we also find that the most common category of flaky tests in proprietary projects is the Async Wait category. To help alleviate the problem of Async Wait flaky tests, we propose the *Flakiness and Time Balancer* (*FaTB*). FaTB identifies the method calls in the test code that are related to timeouts or thread waits, and then it calculates the flaky-test-failure rate of the flaky test. Based on the current flaky-test-failure rate, FaTB then tries various time values and outputs the minimum time values that developers should use depending on their tolerance for flaky-test failures. Our evaluation of FaTB on five versions each of five flaky tests shows that tests can run up to 78% faster and still achieve the same flaky-test-failure rate as before. We also find that the developers thought they "fixed" the flaky tests by increasing some time values, but our empirical experiments show that these time values actually have no effect on the flaky-test-failure rates. Our finding suggests that what developers claim as "fixes" for flaky tests in bug reports, commit messages, etc. can be unreliable, and future work should be more cautious when basing their results on changes that developers claim to be "fixes".

CHAPTER 9: RELATED WORK

This chapter presents work related to this dissertation. Namely, Section 9.1 presents related work on detecting flaky tests, Section 9.2 presents related work on characterizing flaky tests, and Section 9.3 presents related work on taming flaky tests.

9.1 DETECTING FLAKY TESTS

Several projects have proposed techniques for detecting order-dependent (OD) flaky tests [19, 59, 216, 237]. Specifically, Zhang et al. [237] proposed randomizing test orders to detect OD tests. Biagiola et al. [19] studied test dependencies within web applications, noting the challenges in tracking the test dependencies in shared state between server-side and client-side parts of web applications. They proposed a technique for detecting these test dependencies based on string analysis and natural language processing. Waterloo et al. [216] built a static analysis tool to detect inter-test and external dependencies. Gambi et al. [59] proposed using data-flow analysis to detect OD tests.

Our contributions to detecting flaky tests include the development of iDFlakies, a tool that randomizes test orders to detect OD tests. Our work follows Zhang et al. [237] who detected OD tests through random ordering of all the test methods in a test suite. However, unlike Zhang et al. [237], the test orders that iDFlakies run do not interleave the test methods across different test classes, and consequently, the test orders would respect how JUnit actually runs tests. Furthermore, we publicize a dataset of flaky tests across a much larger number of projects. We also presented a probabilistic analysis for OD tests to fail that all prior work on OD tests lacked and a systematic approach to cover all test-method pairs. As presented in Section 3.3.3, our probabilistic analysis finds that whether test methods are allowed to interleave across different test classes or not, the probability of detecting OD tests is similar. However, by not interleaving test methods across different test classes, the test orders from iDFlakies avoid potential false alarms. Finally, we presented a study to understand when and how flaky-test detection tools should be used. Namely, flaky-test detection tools would not be effective if they are applied immediately on a test introducing commit if the test is not yet flaky. This observation is important because running these tools constantly (i.e., on every commit) can be costly in machine time, even if the detection is done offline to not block the developers' critical path.

Beyond techniques specific for detecting OD tests, Bell et al. [18] proposed monitoring the changes between revisions to determine whether test failures are due to flaky tests or

not. Pinto et al. [160] proposed using machine learning and natural language processing techniques based on tokens obtained from test-method code to detect flaky tests. Alshammari et al. [7] also used similar techniques and included other features such as the presence of test smells, hard coded values, the age of tests, and more to detect flaky tests. Shi et al. [184] proposed NonDex, a tool for detecting incorrect assumptions by developers on specifications. Terragni et al. [199] proposed using various containers, each specialized to detect and root cause a specific category of flaky tests. Similarly, Mozilla's Test Verification [201] also aims to detect various categories of flaky tests by having verification steps that each use a different strategy to detect flaky tests. Recently, Mudduluru et al. [143] proposed verifying the absence of flaky tests using type systems. Overall, detecting flaky tests has garnered much attention in many domains over the years. In fact, in recent years, other domains such as probabilistic and machine learning applications [40] and Android [38, 188] have also received much attention on detecting this important problem.

Gyori et al. [73] proposed PolDet, which monitors the heap and filesystem to detect tests that may become polluters later. Complementary, Huo and Clause [84] proposed detecting "brittle assertions", which can detect tests that may become either victims or brittles later. These two techniques could indeed be effective at detecting *potential* flaky tests when the tests are first introduced. However, these specific techniques focus on the shared heap and filesystem, while flaky tests have many other sources of flakiness (e.g., random numbers, network, timing, or concurrency) that still remain to be addressed. Another similar work that aims to detect potential flaky tests when they are first introduced is FRITTER by Parry et al. [159]. FRITTER aims to generate tests using automatic program repair techniques to expose whether a test can be flaky or not.

9.2 CHARACTERIZING FLAKY TESTS

In this section, we present related work on studying the categories of flaky tests, on studying the reproducibility of flaky-test failures, and on debugging flaky tests.

Studies of Flaky-Test Categories. In recent years we have seen a number of studies on the categories of flaky tests [42, 61, 126, 167, 205, 237]. Zhang et al. [237] was the first academic study to formally define Test Order Dependency, which is when test outcomes can be affected by the order in which the tests are run. Luo et al. [126] then provided the first comprehensive academic study on the various categories of flaky tests. Their study is based on a *manual investigation* of commits that are likely related to fixing flaky tests where the commits are found by searching for certain flaky-test related keywords. They manually

investigated 201 commits from 51 open-source projects, finding that the primary causes for flakiness are (1) Async Wait (AW), (2) Concurrency, and (3) Test Order Dependency (OD). The study reported that 78% of flaky tests are flaky in the commit they are introduced. Another study on flaky tests interviewed Mozilla developers after flaky tests were fixed in Mozilla-related projects [42]. This work studied developers' perception of flaky tests and had developers categorize the patches that developers claim to have fixed flaky tests. This study extends the one from Luo et al. [126] by identifying additional categories of flaky tests based on developer surveys. A recent study by Gruber et al. [72] on flaky tests in Python programs found that Test Order Dependency accounted for 59% of the flaky tests studied and 28% were caused by test infrastructure problems. The remaining 13% are mainly due to the use of network and randomness APIs.

The studies presented in this dissertation provide information on flaky tests that prior studies lacked. Specifically, in Chapter 4, our work presented the first dataset of commits when flaky tests first become flaky obtained from using two state-of-the-art flaky-test detection tools, while prior work [42, 126] found flaky tests by reading change logs or surveying developers. The manual work from prior studies is highly costly to reproduce, so we do not further compare how our results may differ if we used a similar experimental methodology as prior studies. However, as we reported in Chapter 8, we find several cases where developers claim they "fixed" a flaky test when our experiments show that their changes do not fix or reduce these tests' frequency of flaky-test failures. As more studies are conducted on the fixes of flaky tests, future work should better explore how one can confirm whether changes by developers do indeed fix flaky tests.

Gao et al. [61] studied flaky GUI tests. They found that it is difficult to reliably reproduce results from tests that interact with the GUI and that there exists tests that change the state for later-run tests, resulting in GUI-OD tests. Thorve et al. [205] found additional root causes for flaky tests when studying flaky-test commits in Android. A more recent study of Android flaky tests by Romano et al. [167] found that the main causes of Android flaky tests are issues with Async Wait, differences in the underlying platform used to run tests, issues in how the tests interact with the applications (e.g., an intended short click is treated as a long click instead), and issues within the tests (e.g., a test randomly generating data and can fail for certain values).

Studies of Flaky-Test Failure Reproducibility. Labuschagne et al. [104] found that, because of flaky tests, 12.8% of builds on average would pass in some runs and fail in some other runs when the build is run three times. Another study [165] has shown that ignoring flaky-test failures can lead to more crashes in production code for the Firefox web browser.

Thorve et al. [205] examined 77 commits that had commit messages containing the keywords "flaky" or "intermittent". The commits came from 29 Android projects, and they found that 13% of the commits simply skipped or removed flaky tests. As shown in Chapter 8, our study on the reproducibility of flaky tests in proprietary projects finds that the likelihood to reproduce at least one failure from a flaky test with 500 runs can range between 17% to 43% depending on the project. We also find in our study on open-source projects, shown in Chapter 6, that the likelihood to reproduce NOD flaky-test failures for the 107 flaky tests is on average, 2.7%, with the minimum being 0.025% (1 in 4000 runs) and the maximum of 50%. Recently, Kowalczyk et al. [103] proposed using the historical flake rate of flaky tests to model and rank flaky tests at Apple.

Debugging Flaky Tests. Like RootFinder presented in Chapter 5, several other systems use differences in runtime invariants in passing and failing tests to identify likely causes of failures [48, 70, 74, 118, 198]. Fault localization techniques and tools, such as Barinel [3], DStar [222], Ochiai [4], Op2 [145], and Tarantula [92] analyze different passing and failing tests in order to localize likely faults in programs. In contrast, RootFinder focuses only on non-deterministic tests that sometimes pass and sometimes fail, and it depends on the collection of a large volume of runtime logs (to not miss rare flakiness and to check for unplanned invariants).

9.3 TAMING FLAKY TESTS

As discussed in Section 1.5, the main ways to help developers tame flaky test are to *reduce* the chance of flaky-test failures or completely *remove* the chance of flaky-test failures. In this section, we present related work on reducing and removing the chance of flaky-test failures.

Reducing Flaky-Test Failures. Only a few techniques and tools have been developed to reduce or accommodate the impact of flaky tests.

For OD tests, some testing frameworks provide mechanisms for developers to specify the order in which tests must run. For tests written in Java, JUnit since version 4.11 supports executing tests in lexicographic order by test method name [200], while TestNG [203] supports execution policies that respect programmer-written dependence annotations. Other Java testing frameworks such as DepUnit [35], Cucumber [31], and Spock [191] also provide similar mechanisms for developers to manually define test dependencies. Beyond Java testing frameworks, similar mechanisms exist in two of the three popular C# testing frameworks. Namely, both NUnit [153] and xUnit [225] provide mechanisms for using annotations to

customize the order that tests run [155]. While starting in version 2 of MSTest [209], tests can be run in only alphabetical order [211]. These test dependencies specified by developers could be used directly by our approach proposed in Chapter 7, or to improve the test dependencies computed using automatic tools (by adding missing or removing unnecessary test dependencies). Haidry and Miller [75] proposed test prioritization based on OD tests, hypothesizing that running tests with more test dependencies is more likely to expose faults. Arlt et al. [12] proposed that test dependencies and the results of already executed tests can be used to infer the test results of yet to be executed tests. In our work presented in Chapter 7, we enhance existing, traditional test prioritization algorithms (along with other regression testing algorithms) to satisfy test dependencies. Test dependencies have also been considered for test-suite minimization by Lin et al. [119].

VMVM [16] is a technique for accommodating test dependencies through a modified runtime environment. VMVM's runtime resets the reachable shared state (namely, the parts of the in-memory heap reachable from static variables in Java) between test runs. The restoration is all done within one JVM execution of all tests, providing benefits of isolation per test without needing to start/stop separate JVMs per test. Similar to the work by Kapfhammer and Soffa [99], VMVM considers a test as an order-dependent test if it accesses a memory location that has been written by another test, being neither necessary nor sufficient to affect the test outcome. Note that VMVM does not aim to detect flaky tests, and resets shared state for *all* tests, regardless of pollution or not. Similarly, Muşlu et al. [142] proposed a technique to run tests in separate processes regardless of whether tests pollute or use a shared state or not. Bell et al. [18] also evaluated how various forms of isolation can help in test reruns to detect which test failures are due to flaky tests. However, all forms of isolation add extra overhead on top of executing tests, even with the recent work that describes how to reduce the overhead of test isolation in Maven Surefire [152].

For AW tests, Jagannath et al. [87] proposed IMUnit, a new language that allows developers to specify the execution flow of tests that make asynchronous method calls. Similarly, Elmas et al. [46] proposed CONCURRIT, a scripting language that allows developers to control the scheduling of threads to find or reproduce concurrency bugs. Other work [127, 130, 214] proposed tools that help enforce policies specified by the developers. These policies dictate the scheduling in which threads run, and developers have to manually write these policies. Unlike these prior projects, FaTB presented in Chapter 8 does not require the developers to provide additional information (e.g., policies in which threads should execute), or write their code differently. Instead, FaTB assists the developers by systematically deriving the time a test should wait for asynchronous calls. Fowler [57] proposed three main ways in testing asynchronous code: (1) creating a synchronous interface between tests and asynchronous

code, (2) implementing callbacks on all asynchronous code, and (3) checking frequently on whether an asynchronous service is complete. The implementation of (1) and (2) requires substantial effort from developers to setup and maintain. The implementation of (2) and (3) also requires the developers to provide some timeout value for the asynchronous call, which may never complete. FaTB can help the developers systematically derive a timeout value that minimizes the runtime while keeping flaky-test-failure rate low.

For other categories of flaky tests, Bell et al. [18] proposed DeFlaker, a technique that monitors the code coverage of recent code changes and marks as flaky any newly failing test that does not execute any of the code changes. Developers then need not investigate test failures that are from tests marked as flaky for regression faults. Dutta et al. has also proposed TERA [39] and FLEX [41]. Both techniques aim to reduce flaky-test failures for machine learning applications.

Removing Flaky-Test Failures. Automatic patch generation is a well-studied topic [115, 121, 122, 133, 141, 150, 218, 219]. The goal is to automatically patch faults in the code, exposed by failing tests. These techniques generate patches using a variety of mechanisms such as systematically mutating code, learning from example patches, and symbolic execution. To validate the success of the patches, most of the techniques rely on the outcomes of tests. Shi et al. [186] proposed iFixFlakies, a tool to automatically fix OD tests. Unlike most work on automatic patch generation, iFixFlakies aims to patch *tests* as opposed to the code under test. iFixFlakies does so by creating test-code patches after searching for code among the existing tests that can be used to make OD tests pass. Zhang [236] proposed DexFix, an automated approach to repair wrong assumptions on underdetermined specifications in both the (flaky-) test code and the code under test. Daniel et al. [32, 33], Mirzaaghaei et al. [139], and Yang et al. [226] also fixed test code, while Gao et al. [60] and Stocco et al. [194] fixed test scripts for GUI. However, they all fixed tests that become broken due to code evolution, not flaky tests.

CHAPTER 10: CONCLUSIONS AND FUTURE WORK

Developers typically perform regression testing to ensure that their recent changes do not break existing functionality. Unfortunately, during regression testing, developers can waste time debugging their code changes because of spurious failures from flaky tests, which are tests that can both pass and fail when run multiple times on the same version of code. In recent years, many software organizations have reported that flaky tests are one of their biggest problems in software development.

This dissertation tackled three main aspects of flaky tests. First, this dissertation presented novel techniques to *detect* flaky tests so that developers can preemptively prevent the problem of flaky tests from affecting their regression testing results (Chapter 2, Chapter 3, and Chapter 4). Second, this dissertation presented novel techniques to *characterize* flaky tests to help developers better understand their flaky tests and to help researchers invent new solutions to the flaky-test problem (Chapter 5 and Chapter 6). Lastly, this dissertation presented novel techniques to *tame* the problem of flaky tests by accommodating the flakiness so that flaky tests do not mislead developers during regression testing (Chapter 7 and Chapter 8). Overall, the work in this dissertation has helped detect more than 2000 flaky tests in over 150 open-source projects and fix more than 500 flaky tests in over 80 open-source projects.

One prominent category of flaky tests is *order-dependent (OD)* flaky tests. Each OD test has at least one order in which the test passes and another order in which the test fails, and for every test order, the test either passes or fails in all runs of that test order. On the other hand, as described in Section 1.2, flaky tests that are not OD are referred to as *Non-deterministic (NOD)* tests. A prominent category of NOD tests is *async-wait (AW)* flaky tests. Each AW test makes at least one asynchronous call and passes if the asynchronous call finishes on time but fails if the call finishes too early or too late.

For detecting flaky tests, this dissertation first presented a tool called iDFlakies that we developed and made publicly available [85]. iDFlakies can (1) detect flaky tests and classify them into two categories, and (2) be easily integrated into Maven projects that use JUnit. Second, this dissertation presented a methodology to analytically obtain the flake rates of OD tests and proposed a simple change to the random sampling of test orders to increase the probability of detecting OD tests. Our analysis found that some OD tests have a rather low flake rate, as low as 1.2%. Third, this dissertation presented an algorithm that systematically explores all consecutive test pairs, guaranteeing the detection of all OD tests that depend on one other test, while running substantially fewer tests than a naive

exploration that runs every pair by itself. Lastly, this dissertation presented a collection of artifacts, including Docker images and test-run logs, that we used to create a dataset of flaky tests [105]. Using the dataset, this dissertation also presented a study of flaky tests in open-source Java projects. Our findings included how prevalent OD and NOD types of flaky tests are, how to automatically detect these tests, when flaky tests are introduced, what changes cause tests to be flaky, and how should developers use flaky-test detection tools.

For characterizing flaky tests, this dissertation first presented a tool called RootFinder that we developed and made publicly available [168]. RootFinder analyzes the logs of passing and failing executions of the same test to suggest method calls that could be responsible for the flakiness. Second, this dissertation presented an end-to-end framework, developed within Microsoft that uses RootFinder to root-cause flaky tests. Third, this dissertation presented a qualitative study of flaky tests in Microsoft proprietary projects. Our qualitative study provided insights on root causing flaky tests with in-depth examples that demonstrate the root causes of flaky tests. Lastly, this dissertation presented our empirical evaluation of flaky tests in open-source Java projects. Our evaluation provided actionable guidelines and practical suggestions for developers and researchers to rerun, detect, debug, and prioritize flaky tests.

For taming flaky tests, this dissertation first presented a study of how OD tests affect traditional regression-testing techniques such as test prioritization, test selection, and test parallelization. When we applied regression testing techniques to test suites containing OD tests, 82% of the human-written and 100% of the automatically generated test suites contain one or more OD tests that fail. Second, this dissertation presented a general approach to enhance traditional regression-testing techniques to be dependent-test-aware. We applied our general approach to 12 traditional regression-testing algorithms, and made them and our approach publicly available [5]. An evaluation of the 12 enhanced algorithms showed that the orders produced by the enhanced algorithms can have 80% fewer OD-test failures, while being only 1% slower than the orders produced by the unenhanced algorithms. Third, this dissertation presented a study on the lifecycle and categories of flaky tests in Microsoft proprietary projects. Our study results suggested the need for an approach to accommodate AW tests. Lastly, this dissertation presented an automated approach, called FaTB, to balance test flakiness and runtime. Our empirical experiments showed that FaTB can help AW tests run up to 78% faster, and the AW tests will still have the same flaky-test-failure rates as before.

10.1 FUTURE WORK

In this section, we describe potential future work that can further advance the work presented in this dissertation.

Detecting Other Causes of Flaky Tests. This dissertation focused on two prominent categories of flaky tests: OD and AW tests. However, as prior studies have found [42, 126], there are many other categories of flaky tests based on concurrency, network, I/O, time, random numbers, etc. Future work should explore how to better detect these other categories of flaky tests. For example, the Concurrency category of flaky tests can pass or fail due to different threads interacting in a non-deterministic manner (e.g., data races, deadlocks). The use of data-race and deadlock detection tools [10, 24, 65, 79, 86, 135, 212] may be effective in detecting this category of flaky tests. However, not every data race would actually result in a failing test result. Similarly, not every data race would indicate a fault in the program. In fact, prior papers [47, 100, 148, 213, 232] have found that 76%–90% of data races are harmless races, i.e., the races are added either fortuitously or by design and they do not harm the program's correctness. Therefore, future work should also explore how data races can be prioritized so that harmful races (i.e., those that harm program correctness or test results) come before harmless races (i.e., those that do not harm program correctness or test results). Overall, we foresee that the detection of data races can help developers better detect Concurrency flaky tests and help them understand why their tests may be flaky.

Detecting Flaky Tests in New Domains. Most existing papers from academic researchers on flaky tests use Java [18, 38, 59, 61, 73, 84, 108, 109, 111, 112, 126, 160, 167, 184, 186, 188, 205, 237] programs with a few recent papers using Python [39–41, 72, 159], C# [106, 107], and C [42] programs. Although the diversity of the programming languages has improved recently, few existing papers study UI-based programs. One recent paper [167] performed an empirical study of flaky tests in Android applications. The study found that AW tests from the loading of UIs are a prominent category of Android flaky tests. To help with this category of flaky tests, future work can help developers translate UI-based tests to non-UI-based tests (similar to prior work on factoring system tests to unit tests [44, 94, 95, 156, 177, 224]) and consequently avoid async-wait issues, or improve the replayability of UI events by dynamically deciding how long to wait based on the load of the system during test execution. Beyond flaky tests in the mobile domain, programs in domains such as machine learning, probabilistic programming [39–41], and quantum computing are inherently non-deterministic. These programs are more difficult to test, because test or-

acles are generally difficult to write for non-deterministic programs. Yet, testing them is important as many companies such as Apple, Facebook, Google, and Microsoft are increasingly relying on these programs for many tasks. Future work should also explore detecting, characterizing, and taming of flaky tests in these other domains.

Verifying (Lack Of) Flaky Tests. The most common approach to detect flaky tests is by rerunning tests many times, and observing both passing and failing runs. However, reruns can be quite costly and could still miss detecting flaky tests. Applying formal methods to detect flaky tests could provide better results and provide guarantees on whether tests can be flaky or not [143]. For example, applying model-checking techniques to detect flaky tests can be promising. Model-checking techniques use a model to check whether a system meets a given specification (e.g., job manager is shutdown before victim test is run for the example in Section 1.2.1). Specifically, for OD tests, victims must depend on a state that is polluted by a polluter. Therefore, for every victim and polluter pair, there must be at least two (pre-)states, one in which the victim passes and one in which the victim fails. Automatically identifying the minimal state for the victim to pass and the minimal state for the victim to fail would aid the debugging of the victim, and the searching of test orders where the victim passes and orders where the victim fails. Besides model checking, symbolic execution [67, 182, 206] is an increasingly used, formally based technique for software testing. In recent years, symbolic execution has been shown to be effective in various testing tasks. Given a program with some inputs labeled as symbolic, the execution can generate concrete values for the inputs, e.g., to achieve high code coverage or reach specific program points. Future work can encode the problem of detecting flaky tests into symbolic execution by making some values symbolic (e.g., system time) and attempting to generate values that make the test pass and values that make the test fail.

Reducing Testing Time With OD Tests. OD tests can be useful sometimes because they can help developers run their tests faster by allowing the tests to share resources [175]. For example, one test may create a database that another test uses, allowing the latter test to run faster as an OD test instead of recreating the database. Prior papers have already suggested changing the order that tests are run to reduce the runtime of tests [195, 196]. Additionally, every popular Java testing framework already permits developers to specify dependencies among tests, including JUnit, TestNG, DepUnit, Cucumber, and Spock. In fact, as of June 2021, the JUnit annotations `@FixMethodOrder` and `@TestMethodOrder`, and the TestNG attributes `dependsOnMethods` and `dependsOnGroups` have been used over 270k times in Java files on GitHub. With the popularity of OD tests to help improve the runtime of test

suites, there currently exists no automated solution to help developers (1) find tests that can be made order-dependent, (2) create helper functions to effectively share resources between tests, and (3) optimize the placement of tests so that helper functions can be effectively shared to minimize the runtime of test suites. Future work should explore the use of static and dynamic program analyses to help identify common resources [147] that can be shared between tests to create the helper functions. Using the helper functions, developers can then optimize the placement of tests so that the time spent setting up each test class is increased, but the time spent setting up each test method is substantially decreased.

Automatically Accommodating and Fixing More Categories of Flaky Tests. This dissertation described two pieces of work on accommodating flaky tests (Chapter 7 and Chapter 8) with one piece of the work focused on OD tests and the other on AW tests. However, as prior studies have found [42, 126], there are many other categories of flaky tests. Future work should explore how accommodating techniques can be developed for these other categories. Beyond accommodating flaky tests, developers may also prefer to fix their flaky tests. One proposal for automated fixing is iFixFlakies [186], an automatic technique to fix OD tests. Similar to accommodating techniques, fixing techniques should also explore how the techniques can be developed for other categories of flaky tests.

Benefiting From Non-determinism. Prior work [42, 126] on flaky tests has largely painted them as a problem that developers should get rid of because the tests can non-deterministically pass or fail for the same code. However, non-determinism can be used to improve performance (e.g., parallel computing) and security (e.g., address space layout randomization). For example, a test for a parallel problem can run more efficiently using asynchronous code, but the timing variations of asynchronous code can cause parts of the test to run in a non-deterministic order. These variations can cause flaky-test failures. Our techniques on accommodating flaky tests took the first step to accommodate the problems of non-determinism so that developers can still benefit from using asynchronous code. However, many software engineering approaches (e.g., test generation and automatic program repair) still assume determinism. Future work should explore how non-determinism would affect many of these approaches and how the approaches can be accommodated to benefit from non-determinism. For example, how will non-determinism affect test generation [11]? How can non-determinism be accommodated for test-generation approaches such that the tests generated can find more bugs, run faster, and be easier for developers to maintain? To help address these problems, one may conduct studies to understand the effect of non-determinism on various approaches and then use the results to develop accommodation techniques.

As long as people write software (or even generate software, e.g., using machine learning), there will be bugs, and software testing will be a practical way to detect such bugs. The work on flaky tests presented in this dissertation scratches only the surface of the problems that non-determinism brings to software development. In the future, we foresee many more advancements to detecting, characterizing, and taming flaky tests. We also foresee many more advancements to the problems of non-determinism beyond the domain of regression testing, such as test generation and automatic program repair. All of these advancements would help reduce the problems that non-determinism brings to software development and improve the dependability of software.

REFERENCES

[1] "A large-scale longitudinal study of flaky tests - Tools and dataset," Accessed 2021. [Online]. Available: https://sites.google.com/view/first-commit-flaky-test

[2] "A machine learning solution for detecting and mitigating flaky tests," Accessed 2021. [Online]. Available: https://eng.fitbit.com/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests

[3] R. Abreu, P. Zoeteweij, and A. Gemund, "Spectrum-based multiple fault localization," in *ASE*, 2009.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, 2009.

[5] "Accommodating test dependence project web," Accessed 2021. [Online]. Available: https://sites.google.com/view/test-dependence-impact

[6] "Activiti," Accessed 2021. [Online]. Available: https://github.com/activiti/activiti

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

[8] "Anjiang-Wei/TuscanSquare," Accessed 2021. [Online]. Available: https://github.com/Anjiang-Wei/TuscanSquare

[9] "Apache Hadoop," Accessed 2021. [Online]. Available: https://github.com/apache/hadoop

[10] "Archer," Accessed 2021. [Online]. Available: https://github.com/PRUNERS/archer

[11] A. Arcuri, G. Fraser, and J. P. Galeotti, "Automated unit test generation for classes with environment dependencies," in *ASE*, 2014.

[12] S. Arlt, T. Morciniec, A. Podelski, and S. Wagner, "If A fails, can B still succeed? Inferring dependencies between test results in automotive system testing," in *ICST*, 2015.

[13] "Azure Data Explorer," Accessed 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/data-explorer

[14] "Bazel," Accessed 2021. [Online]. Available: https://bazel.build

[15] J. Bell, "Detecting, isolating, and enforcing dependencies among and within test cases," in *FSE Doctoral Symposium*, 2014.

[16] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *ICSE*, 2014.

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

[18] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *ICSE*, 2018.

[19] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella, "Web test dependency detection," in *ESEC/FSE*, 2019.

[20] L. C. Briand, Y. Labiche, and S. He, "Automating regression test selection based on UML designs," *IST*, 2009.

[21] "Buck," Accessed 2021. [Online]. Available: https://buckbuild.com

[22] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *ASPLOS*, 2010.

[23] A. Chou, "Static analysis in industry," Accessed 2021. [Online]. Available: http://popl.mpi-sws.org/2014/andy.pdf

[24] M. Christiaens and K. De Bosschere, "TRaDe: Data race detection for Java," in *ICCS*, 2001.

[25] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *ICFP*, 2000.

[26] "Coefficient of variation," Accessed 2021. [Online]. Available: https://en.wikipedia.org/wiki/Coefficient_of_variation

[27] C. Croux and C. Dehon, "Influence functions of the Spearman and Kendall correlation measures," *Statistical methods & applications*, 2010.

[28] C. Csallner and Y. Smaragdakis, "JCrasher: An automatic robustness tester for Java," *SPE*, 2004.

[29] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A hybrid analysis tool for bug finding," *TOSEM*, 2008.

[30] "Cucumber," Accessed 2021. [Online]. Available: https://cucumber.io/docs/cucumber

[31] "Cucumber reference - Scenario hooks," Accessed 2021. [Online]. Available: https://cucumber.io/docs/cucumber/api/#hooks

[32] B. Daniel, T. Gvero, and D. Marinov, "On test repair using symbolic execution," in *ISSTA*, 2010.

[33] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *ASE*, 2009.

[34] "Data used by "A Study on the Lifecycle of Flaky Tests"," Accessed 2021. [Online]. Available: https://github.com/winglam/flaky-test-lifecycle-data

[35] "DepUnit," Accessed 2021. [Online]. Available: https://www.openhub.net/p/depunit

[36] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *ESE*, 2005.

[37] "Docker," Accessed 2021. [Online]. Available: https://www.docker.com

[38] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in Android via event order exploration," To appear.

[39] S. Dutta, J. Selvam, A. Jain, and S. Misailovic, "TERA: Optimizing stochastic regression tests in machine learning projects," in *ISSTA*, 2021.

[40] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, "Detecting flaky tests in probabilistic and machine learning applications," in *ISSTA*, 2020.

[41] S. Dutta, A. Shi, and S. Misailovic, "FLEX: Fixing flaky tests in machine learning projects by updating assertion bounds," in *ESEC/FSE*, To appear.

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

[43] "Elastic-Job," Accessed 2021. [Online]. Available: https://github.com/elasticjob/elastic-job-lite

[44] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *FSE*, 2006.

[45] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *ISSTA*, 2000.

[46] T. Elmas, J. Burnim, G. Necula, and K. Sen, "CONCURRIT: A domain specific language for reproducing concurrency bugs," in *PLDI*, 2013.

[47] D. Engler and K. Ashcraft, "RacerX: Effective, static detection of race conditions and deadlocks," in *SOSP*, 2003.

[48] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 2007.

[49] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula, "CloudBuild: Microsoft's distributed and caching build service," in *ICSE*, 2016.

[50] "Event tracing for Windows (ETW) simplified," Accessed 2021. [Online]. Available: https://bit.ly/2NgEBzl

[51] "Executing unit tests in parallel on a multi-CPU/core machine in Visual Studio," Accessed 2021. [Online]. Available: http://blogs.msdn.com/b/vstsqualitytools/archive/2009/12/01/executing-unit-tests-in-parallel-on-a-multi-cpu-core-machine.aspx

[52] "Facebook testing and verification request for proposals," Accessed 2021. [Online]. Available: https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019

[53] "fastjson - Git issue," Accessed 2021. [Online]. Available: https://github.com/alibaba/fastjson/issues/2584

[54] "Flakiness Dashboard HOWTO," Accessed 2021. [Online]. Available: http://www.chromium.org/developers/testing/flakiness-dashboard

[55] "Flaky test statistics," Accessed 2021. [Online]. Available: https://sites.google.com/view/flakyteststatistics

[56] "Flaky tests (and how to avoid them)," Accessed 2021. [Online]. Available: https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60

[57] M. Fowler, "Eradicating non-determinism in tests," Accessed 2021. [Online]. Available: https://martinfowler.com/articles/nondeterminism.html

[58] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *ISSTA*, 2011.

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

[60] Z. Gao, Z. Chen, Y. Zou, and A. M. Memon, "SITAR: GUI test script repair," *TSE*, 2016.

[61] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" in *ICSE*, 2015.

[62] "Git bisect," Accessed 2021. [Online]. Available: https://git-scm.com/docs/git-bisect

[63] "GitHub," Accessed 2021. [Online]. Available: https://github.com

[64] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *ISSTA*, 2015.

[65] "Go data race detector," Accessed 2021. [Online]. Available: https://golang.org/doc/articles/race_detector

[66] P. Gochenour and R. Andre, "How to deal with flaky Java tests," Accessed 2021. [Online]. Available: https://wiki.saucelabs.com/display/DOCS/How+to+Deal+with+Flaky+Java+Tests

[67] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI*, 2005.

[68] S. W. Golomb and H. Taylor, "Tuscan squares – A new family of combinatorial designs," *Ars Combinatoria*, 1985.

[69] Google, "TotT: Avoiding flakey tests," Accessed 2021. [Online]. Available: http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html

[70] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *SPIN*, 2003.

[71] A. Groce, A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction for quick testing," in *ICST*, 2014.

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

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

[74] J. Ha, J. Yi, P. Dinges, J. Manson, C. Sadowski, and N. Meng, "System to uncover root cause of non-deterministic (flaky) tests," in *Google Patent*, 2013. [Online]. Available: https://patents.google.com/patent/US9311220

[75] S. Z. Haidry and T. Miller, "Using dependency structures for prioritization of functional test suites," *TSE*, 2013.

[76] 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.

[77] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for Java software," in *OOPSLA*, 2001.

[78] B. Harry, "How we approach testing VSTS to enable continuous delivery," Accessed 2021. [Online]. Available: https://blogs.msdn.microsoft.com/bharry/2017/06/28/testing-in-a-cloud-delivery-cadence

[79] "Helgrind," Accessed 2021. [Online]. Available: https://www.valgrind.org/docs/manual/hg-manual.html

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

[81] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *ASE*, 2016.

[82] R. Houston, "Tackling the minimal superpermutation problem," arXiv.

[83] H.-Y. Hsu and A. Orso, "MINTS: A general framework and tool for supporting test-suite minimization," in *ICSE*, 2009.

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

[85] "iDFlakies: Flaky test dataset," Accessed 2021. [Online]. Available: https://sites.google.com/view/flakytestdataset

[86] "Infer static analyzer," Accessed 2021. [Online]. Available: https://fbinfer.com

[87] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Roşu, and D. Marinov, "Improved multithreaded unit testing," in *ESEC/FSE*, 2011.

[88] "Java platform module system," Accessed 2021. [Online]. Available: https://www.oracle.com/corporate/features/understanding-java-9-modules.html

[89] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *ASE*, 2009.

[90] 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.

[91] Y. Jiang, L. Sivalingam, S. Nath, and R. Govindan, "Webperf: Evaluating what-if scenarios for cloud-hosted web applications," in *SIGCOMM*, 2016.

[92] J. Jones and M. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *ASE*, 2005.

[93] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002.

[94] M. Jorde, S. G. Elbaum, and M. B. Dwyer, "Increasing test granularity by aggregating unit tests," in *ASE*, 2008.

[95] S. Joshi and A. Orso, "SCARPE: A technique and tool for selective capture and replay of program executions," in *ICSM*, 2007.

[96] "JUnit," Accessed 2021. [Online]. Available: https://junit.org

[97] "JUnit @Ignore annotation," Accessed 2021. [Online]. Available: http://junit.sourceforge.net/javadoc/org/junit/Ignore.html

[98] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *ISSTA*, 2014.

[99] G. M. Kapfhammer and M. L. Soffa, "A family of test adequacy criteria for database-driven applications," in *ESEC/FSE*, 2003.

[100] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. Data race bugs: Telling the difference with Portend," in *ASPLOS*, 2012.

[101] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, 2002.

[102] T. Kim, R. Chandra, and N. Zeldovich, "Optimizing unit test execution in large software programs using dependency analysis," in *APSys*, 2013.

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

[104] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of Java projects using continuous integration," in *ESEC/FSE*, 2017.

[105] W. Lam, "Illinois Dataset of Flaky Tests (IDoFT)," Accessed 2021. [Online]. Available: http://mir.cs.illinois.edu/flakytests

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

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

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

[109] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *ISSTA*, 2020.

[110] W. Lam, S. Srisakaokul, B. Bassett, P. Mahdian, T. Xie, P. Lakshman, and J. de Halleux, "A characteristic study of parameterized unit tests in .NET open source projects," in *ECOOP*, 2018.

[111] 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.

[112] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," in *OOPSLA*, 2020.

[113] W. Lam, Z. Wu, D. Li, W. Wang, H. Zheng, H. Luo, P. Yan, Y. Deng, and T. Xie, "Record and replay for Android: Are we there yet in industrial cases?" in *ESEC/FSE Industry Track*, 2017.

[114] J. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. Rajamani, and R. Venkatapathy, "Righting software," *IEEE Software*, 2004.

[115] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*, 2012.

[116] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *FSE*, 2016.

[117] J. Liang, S. Elbaum, and G. Rothermel, "Redefining prioritization: Continuous prioritization for continuous integration," in *ICSE*, 2018.

[118] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable statistical bug isolation," in *PLDI*, 2005.

[119] J.-W. Lin, R. Jabbarvand, J. Garcia, and S. Malek, "Nemo: Multi-criteria test-suite minimization with integer nonlinear programming," in *ICSE*, 2018.

[120] B. Liskov and J. Guttag, *Program development in Java: Abstraction, specification, and object-oriented design.* Addison-Wesley Longman Publishing Company, 2000.

[121] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017.

[122] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *POPL*, 2016.

[123] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[124] E. Lucas, "Récréations mathématiques," 1894.

[125] L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze, "Troubleshooting transiently-recurring problems in production systems with blame-proportional logging," in *USENIX*, 2018.

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

[127] Q. Luo and G. Roşu, "EnforceMOP: A runtime property enforcement system for multithreaded programs," in *ISSTA*, 2013.

[128] "Manage flaky tests," Accessed 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/pipelines/test/flaky-test-management

[129] "Managed vs. Unmanaged development," Accessed 2021. [Online]. Available: https://bit.ly/2Or4C3p

[130] E. R. B. Marques, F. Martins, and M. Simões, "Cooperari: A tool for cooperative testing of multithreaded Java programs," in *PPPJ*, 2014.

[131] "Maven," Accessed 2021. [Online]. Available: https://maven.apache.org

[132] "Maven Surefire plugin," Accessed 2021. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin

[133] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016.

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

[135] "MemorySanitizer," Accessed 2021. [Online]. Available: https://clang.llvm.org/docs/MemorySanitizer.html

[136] J. Micco, "Continuous integration at Google scale," Accessed 2021. [Online]. Available: https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale

[137] ——, "The state of continuous integration testing at Google," in *ICST*, 2017.

[138] "Microsoft Azure," Accessed 2021. [Online]. Available: https://azure.microsoft.com

[139] M. Mirzaaghaei, F. Pastore, and M. Pezzè, "Supporting test suite evolution through test case adaptation," in *ICST*, 2012.

[140] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel test generation and execution with Korat," in *ESEC/FSE*, 2007.

[141] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computer Survey*, 2018.

[142] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *ESEC/FSE*, 2011.

[143] R. Mudduluru, J. Waataja, S. Millstein, and M. D. Ernst, "Verifying determinism in sequential programs," in *ICSE*, 2021.

[144] K. Muşlu, Y. Brun, and A. Meliou, "Preventing data errors with continuous testing," in *ISSTA*, 2015.

[145] L. Naish, H. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *TOSEM*, 2011.

[146] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso, "Regression testing in the presence of non-code changes," in *ICST*, 2011.

[147] S. C. Narayanan, "Clustered test execution using Java PathFinder," in *University of Texas at Austin, Master's Thesis*, 2010.

[148] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *PLDI*, 2007.

[149] "Netflix automation talks - Test automation at scale," Accessed 2021. [Online]. Available: https://youtu.be/FrBN94gUn_I?t=764

[150] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *ICSE*, 2013.

[151] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys*, 2011.

[152] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric, "Debugging the performance of Maven's test isolation: Experience report," in *ISSTA*, 2020.

[153] "NUnit framework," Accessed 2021. [Online]. Available: https://nunit.org

[154] M. Ollis, "Sequenceable groups and related topics," *Electronic Journal of Combinatorics*, 2013.

[155] "Order unit tests," Accessed 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/testing/order-unit-tests

[156] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *WODA*, 2005.

[157] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *FSE*, 2004.

[158] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007.

[159] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "Flake it 'till you make it: Using automated repair to induce and fix latent test flakiness," in *APR*, 2020.

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

[161] "Pull request #2148: Fixing flaky tests in DateTest4_indian and DateTest5_iso8601," Accessed 2021. [Online]. Available: https://github.com/alibaba/fastjson/pull/2148

[162] "Pull request #2906: Fixing flaky tests in PortTelnetHandlerTest," Accessed 2021. [Online]. Available: https://github.com/apache/incubator-dubbo/pull/2906

[163] "Pull request #3291: Fixing flaky tests in DefaultExtJSONParser_parseArray," Accessed 2021. [Online]. Available: https://github.com/alibaba/fastjson/pull/3291

[164] "Pull request #592: Fixing flaky test ShutdownListenerManagerTest.assertIsShutdownAlready," Accessed 2021. [Online]. Available: https://github.com/elasticjob/elastic-job-lite/pull/592

[165] 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.

[166] "Riptide," Accessed 2021. [Online]. Available: https://github.com/zalando/riptide

[167] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang, "An empirical analysis of UI-based flaky tests," in *ICSE*, 2021.

[168] "Root causing flaky tests," Accessed 2021. [Online]. Available: https://sites.google.com/view/root-causing-flaky-tests

[169] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *TOSEM*, 2004.

[170] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An empirical study of the effects of minimization on the fault detection capabilities of test suites," in *ICSM*, 1998.

[171] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *TSE*, 2001.

[172] "Rspec-core issue 635," Accessed 2021. [Online]. Available: https://github.com/rspec/rspec-core/issues/635

[173] M. J. Rummel, G. M. Kapfhammer, and A. Thall, "Towards the prioritization of regression test suites with data flow information," in *SAC*, 2005.

[174] "Run tests in parallel using the Visual Studio Test task," Accessed 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/pipelines/test/parallel-testing-vstest

[175] "Running your tests in a specific order," Accessed 2021. [Online]. Available: https://www.ontestautomation.com/running-your-tests-in-a-specific-order

[176] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *ICSE*, 2015.

[177] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for Java," in *ASE*, 2005.

[178] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *ISSRE*, 2003.

[179] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world Java bugs," in *MSR*, 2018.

[180] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *ICSE Companion*, 2016.

[181] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ISSTA*, 2009.

[182] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE*, 2005.

[183] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *FSE*, 2014.

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

[185] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov, "Evaluating test-suite reduction in real software evolution," in *ISSTA*, 2018.

[186] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE*, 2019.

[187] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka, "Optimizing test placement for module-level regression testing," in *ICSE*, 2017.

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

[189] "SLOCCount," Accessed 2021. [Online]. Available: https://dwheeler.com/sloccount

[190] "Spock," Accessed 2021. [Online]. Available: http://docs.spockframework.org

[191] "Spock stepwise," Accessed 2021. [Online]. Available: https://www.canoo.com/blog/2011/04/12/spock-stepwise

[192] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *ISSTA*, 2002.

[193] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *ISSTA*, 2013.

[194] A. Stocco, R. Yandrapally, and A. Mesbah, "Visual web test repair," in *ESEC/FSE*, 2018.

[195] P. Stratis and A. Rajan, "Test case permutation to improve execution time," in *ASE*, 2016.

[196] ——, "Speeding up test execution with increased cache locality," *STVR*, 2018.

[197] P. Sudarshan, "No more flaky tests on the Go team," Accessed 2021. [Online]. Available: http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team

[198] W. Sumner, T. Bao, and X. Zhang, "Selecting peers for execution comparison," in *ISSTA*, 2011.

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

[200] "Test execution order," Accessed 2021. [Online]. Available: https://github.com/junit-team/junit4/wiki/Test-execution-order

[201] "Test verification," Accessed 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

[202] "TestingResearchIllinois/testrunner," Accessed 2021. [Online]. Available: https://github.com/TestingResearchIllinois/testrunner

[203] "TestNG," Accessed 2021. [Online]. Available: http://testng.org

[204] "TestNG Documentation," Accessed 2021. [Online]. Available: https://testng.org/doc/documentation-main.html

[205] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in Android apps," in *ICSME, NIER Track*, 2018.

[206] N. Tillmann and J. De Halleux, "Pex: White box test generation for .NET," in *TAP*, 2008.

[207] T. W. Tillson, "A Hamiltonian decomposition of K^*_{2m}, $2m \geq 8$," *Journal of Combinatorial Theory, Series B*, 1980.

[208] "Travis CI - Test and deploy with confidence," Accessed 2021. [Online]. Available: https://travis-ci.org

[209] "Unit testing C# with MSTest and .NET," Accessed 2021. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest

[210] "Valgrind," Accessed 2021. [Online]. Available: http://valgrind.org

[211] "Visual Studio Developer Community - How to run tests in a particular order instead of default alphabetical order," Accessed 2021. [Online]. Available: https://developercommunity.visualstudio.com/t/how-to-run-tests-in-a-particular-order-instead-of/824789

[212] "vmlens," Accessed 2021. [Online]. Available: https://vmlens.com

[213] J. W. Voung, R. Jhala, and S. Lerner, "RELAY: Static race detection on millions of lines of code," in *ESEC/FSE*, 2007.

[214] K. Wang, S. Khurshid, and M. Gligoric, "JPR: Replaying JPF traces using standard JVM," *SEN*, 2018.

[215] W. Wang, W. Lam, and T. Xie, "An infrastructure approach to improving effectiveness of Android UI testing tools," in *ISSTA*, 2021.

[216] M. Waterloo, S. Person, and S. Elbaum, "Test analysis: Searching for faults in tests," in *ASE*, 2015.

[217] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in *TACAS*, 2021.

[218] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*, 2010.

[219] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009.

[220] E. Wendelin, "Introducing flaky test mitigation tools," Accessed 2021. [Online]. Available: https://blog.gradle.org/gradle-flaky-test-retry-plugin

[221] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Publishing Company, 2012.

[222] E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *Transactions on Reliability*, 2014.

[223] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *ICSE*, 1995.

[224] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang, "Language-based replay via data flow cut," in *FSE*, 2010.

[225] "xUnit framework," Accessed 2021. [Online]. Available: https://xunit.net

[226] G. Yang, S. Khurshid, and M. Kim, "Specification-based test repair using a lightweight formal method," in *FM*, 2012.

[227] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *STVR*, 2012.

[228] H. Yu, W. Lam, L. Chen, G. Li, T. Xie, and Q. Wang, "Neural detection of semantic code clones via tree-based convolution," in *ICPC*, 2019.

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

[230] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *TSE*, 2002.

[231] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for Android: Are we really there yet in an industrial case?" in *FSE Industry Track*, 2016.

[232] J. Zhang, W. Xiong, Y. Liu, S. Park, Y. Zhou, and Z. Ma, "ATDetector: Improving the accuracy of a commercial data race detector by identifying address transfer," in *MICRO*, 2011.

[233] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ISSTA*, 2013.

[234] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *ISSTA*, 2012.

[235] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013.

[236] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *ICSE*, 2021.

[237] 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.

[238] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *ISSTA*, 2011.

[239] H. Zheng, D. Li, X. Zeng, B. Liang, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated test input generation for Android: Towards getting there in an industrial case," in *ICSE SEIP*, 2017.

[240] 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.