My research interests are in software engineering, with a focus on improving the dependability of software. Software permeates many systems, and a lack of software dependability can have monumental societal impact. To improve software dependability, my Ph.D. dissertation research has focused on flaky tests [6–8, 10, 12, 13, 18, 22], with additional results on test-input generation [21, 23], code-clone detection [20], automatic program repair [17], record and replay [14], and parameterized unit tests [11]. My work on characterizing software dependability problems has helped open the new research topic of flaky tests. My work on detecting and taming software dependability problems has helped detect, fix, and alleviate hundreds of such problems in open-source and proprietary code using program analysis and machine learning techniques. These problems are in projects from organizations such as Alibaba, Apache, Eclipse, Google, Microsoft, the National Security Agency, Spotify, Tencent, and VMWare. My research has created practical, state-of-the-art techniques and enabled new ones for solving real-world software dependability problems.

1 Motivation and Overview

Developers are constantly making changes to improve software, and the most common way to check if software is reliable after changes is through regression testing. Namely, developers run tests after every change to check that they did not break existing functionality. If a test passed before but fails after a change, developers typically search for the cause of failure within the change. In the presence of flaky tests, which are tests that can non-deterministically pass or fail for the same code, such searching can waste substantial time and effort because the bugs are in the old (test) code, not in the new changes. Essentially, flaky tests can (mis)lead developers to debug the failures in the wrong places. Many software organizations, including Apple [4], Facebook [2, 3], Google [16], and Microsoft [6, 7], report that flaky tests are one of their biggest problems. For example, in my ICSE 2020 work [7], I reported finding 218,000+ flaky-test failures in a span of three months for just six proprietary projects at Microsoft.

My research tackles the flaky-test problem in the following main directions.

Characterizing Flaky Tests: I am among the first to study flaky tests, particularly flaky tests that depend on the order in which the tests are run (OD tests). My work on characterizing flaky tests [6, 8, 12, 13, 22] has helped open the new research topic of flaky tests, particularly on OD tests, which has inspired much work on this topic (e.g., one of my papers has been cited 100+ times [22], and another that was inspired by my characterizations [8] has been cited 30+ times in just a year).

Detecting and Taming Flaky Tests: I proposed techniques to preemptively detect flaky tests before they manifest as flaky-test failures [8, 19, 22] and techniques to tame flaky-test failures [7, 10, 18]. My work, my supervision of students, and the contributions of others have detected and publicized 2000+ flaky tests in open-source projects and fixed 500+ flaky tests (300+ detected and 100+ fixed are by others) [5]. Besides open-source projects, two of my techniques were evaluated on Microsoft proprietary code [6, 7] and have influenced how Microsoft developers deal with flaky tests.

Other work: Beyond my work on flaky tests, I also developed WCTester, an automatic test-input generation tool for Android apps [21, 23]. WCTester originated from my understanding of existing techniques’ limitations and my frequent interactions with the developers of WeChat, a popular messenger app, with 1+ billion monthly active users. WeChat developers adopted WCTester and found that it can achieve substantially higher code coverage than other techniques.

Overall, my research aims to identify real-world software problems and invent practical solutions that can be applied to such problems. I identify problems by contributing to open-source and proprietary code as well as interacting with developers to understand the greatest pitfalls of software engineering. For example, I have closely collaborated with developers at Microsoft [6, 7] and WeChat [21, 23], and I have detected and fixed hundreds of flaky tests in open-source code.
2 Characterizing Flaky Tests  A test suite is most often specified as a set of tests that developers run during regression testing. Having tests specified as a set provides benefits for regression testing techniques such as selection, prioritization, and parallelization [10]. Namely, the test-execution platform is free to run these tests in various test orders. However, as my research [8, 10, 12, 18, 19, 22] shows, when tests are run in different orders, some tests may deterministically pass or fail based on the order the tests are run. I refer to tests exhibiting such behavior as order-dependent (OD) flaky tests. My ISSTA 2014 work [22] is among the first papers on flaky tests, and it helped open the new research topic by characterizing OD flaky tests. Namely, an OD test must have at least one order in which the test deterministically passes and another (different) order in which the test deterministically fails. A flaky-test dataset that I collected [9] contains hundreds of OD flaky tests from open-source projects, with ∼50% of the dataset being OD tests.

Tests that are not OD are known as non-order-dependent (NOD). Such NOD tests nondeterministically pass and fail in at least one order, and are commonly assumed to not depend on the order in which the tests are run [1, 8, 15]. Contrary to the common belief, my ISSRE 2020 work [12] showed that 72% of NOD tests likely do depend on the order. Namely, across different test orders, these tests have significantly different failure rates—the ratio of the number of failed runs over the number of total runs for a particular test order. By studying the failure rates of flaky tests, I am the first to study the relationship of test orders on not just OD tests but also NOD tests.

Beyond studying the effect of test orders on flaky tests, I have also proposed RootFinder [6], the first automatic technique to root-cause flaky tests. RootFinder automatically identifies the root causes of a flaky test by analyzing runtime properties collected during passing and failing runs of a flaky test. To find root causes, RootFinder identifies predicates (e.g., an integer variable is even) that are true in all passing runs but false in all failing runs. RootFinder was evaluated on Microsoft proprietary software and is shown to help developers debug many kinds of flaky tests.

Contributions: My work on characterizing flaky tests helped open this research topic, particularly into the phenomenon of test orders affecting test results. My work with Microsoft developers also proposed the first technique to automatically root-cause flaky tests.

3 Detecting Flaky Tests  As OD tests are among the most prominent kinds of flaky tests [1, 15], I proposed iDFlakies, which runs tests in various orders selected randomly or systematically. iDFlakies employs various simple yet effective detection algorithms (e.g., random order, reverse of a passing order) to detect and automatically categorize OD and NOD tests. Using iDFlakies, I publicized a dataset of 400+ flaky tests from popular, open-source projects [9]. My dataset contains 50.5% OD and 49.5% NOD tests. iDFlakies and my dataset have already been cited 30+ times in just a year and since I started crowdsourcing my dataset [5], it has grown to 2000+ flaky tests. To more rigorously understand iDFlakies’ effectiveness, I presented an analysis of the probability of detecting OD tests using random orders in my TACAS 2021 work [19]. I also propose other detection algorithms including cost-effective ones that guarantee the detection of some OD tests.

Beyond inventing tools to detect flaky tests, I also empirically studied when such tools should be used. Although existing tools to detect flaky tests, known as detectors, are shown to be effective at detecting flaky tests, the tools are often impractical to be used all the time due to their high runtime cost. In my OOPSLA 2020 work [13], I conduct a study to determine when flaky tests become flaky and what changes cause them to become flaky. I find that 85% of flaky tests are flaky when they are added or directly modified, indicating that there is substantial potential value for developers to run detectors only on such tests. My study is the first to empirically evaluate when tests become flaky and to recommend actionable guidelines for applying detectors.
Contributions: My work proposed one of the first effective tools to detect flaky tests, and using the tool, I publicized a trending dataset of flaky tests in open-source projects. My work also empirically studied when such tools should be used and provides actionable guidelines to developers.

4 Taming Flaky Tests

4.1 Fixing Flaky Tests One important challenge for automatically fixing flaky tests is identifying how to get the logic to fix the tests. By definition, OD tests must have at least one order in which the test deterministically passes and another order in which the test deterministically fails. In the failing order, a polluter test may run before the OD test and pollute the state shared among tests in a way that causes the OD test to fail. While manually inspecting and fixing OD tests, I noticed that logic for cleaning the shared, polluted state for OD tests is often already in the test suite in the form of other tests! This insight enabled me to develop iFixFlakies [18], a framework for automatically fixing OD tests. iFixFlakies works by first finding cleaner tests that contain logic to help OD tests pass by cleaning the shared state. iFixFlakies then minimizes the code of cleaner tests to find the minimal change so that OD tests can pass even when they are run after polluter tests. On a dataset of 110 OD tests, iFixFlakies automatically fixed 101 tests. I created patches using the fixes, and developers have already accepted my patches for 76 of these tests. The others are pending review, and none has been rejected.

4.2 Accommodating Flaky Tests Flaky tests can be beneficial for many reasons, such as speeding up test runs by allowing OD tests to share resources. To reap the benefits of flaky tests while minimizing the occurrence that flaky tests mislead developers, I am the first to explore various techniques to accommodate flaky tests. Two important challenges for accommodating flaky tests are identifying what information should be used and how to use it.

4.2.1 OD tests Although OD tests can be beneficial, these tests are problematic for regression testing algorithms that may reorder a test suite (e.g., test prioritization) or run only a subset of tests (e.g., test selection or parallelization). To help developers use regression testing algorithms even when a test suite has OD tests, my ISSTA 2020 work [10] proposes a new, general technique to enhance traditional regression testing algorithms to make them dependent-test-aware, so that OD tests would not manifest as flaky-test failures. To enhance a traditional algorithm, my general technique first uses the traditional algorithm to produce an order and then reorders or adds tests such that all provided test dependencies are satisfied. My evaluation finds that the use of automated tools [8, 22] to generate test dependencies and the careful placement of tests to satisfy the dependencies can result in test orders that cause substantially fewer OD-test failures (80%) and run only marginally slower (<1%) than test orders from unenhanced algorithms. By accommodating test dependencies, my work avoids the flaky-test failures that would otherwise manifest from these OD tests, while still enabling these tests to provide useful information about the quality of the software.

4.2.2 Async Wait tests Besides OD tests, prior studies [1, 15] on flaky tests find that another prominent kind of flaky tests is Async Wait. Such tests are flaky because they make asynchronous calls and do not properly wait for the calls to return. To help with the problem of Async Wait tests, I developed the Flakiness and Time Balancer (FaTB) technique. For each flaky test, FaTB tries various time values for the timeouts and thread waits of the test while keeping track of how these values affect the test’s failure rate. FaTB outputs the minimum time values that developers should use depending on their tolerance for flaky-test failures. My evaluation of FaTB on Microsoft proprietary projects shows that FaTB can effectively reduce the runtime of tests while controlling the failure rate, e.g., some tests can run up to 78% faster while maintaining a low failure rate.
Contributions: My work proposed the first tool to fix flaky tests automatically, and the culmination of my work, my supervision of students, and the contributions of others have led to 500+ patches for flaky tests. My work is also the first to explore the notion of accommodating flaky tests for two kinds so that developers can benefit from flaky tests while minimizing the problems of such tests.

5 Other work Beyond my work on flaky tests, I have also made substantial advancements in test-input generation [21, 23], code-clone detection [20], automatic program repair [17], record and replay [14], and parameterized unit tests [11]. My work on test-input generation overcomes several limitations of existing work and incorporates the needs of developers, especially those of WeChat. By identifying and solving problems this way, I developed techniques that developers would actually integrate as part of their development process. My work on code-clone detection developed a new technique to detect code clones using a tree-based convolutional neural network (CNN). By leveraging a tree-based CNN, my work was able to leverage structural information from code that prior work had neglected. To address the lack of generalization from using machine learning (ML) to detect code clones, my technique also included ideas inspired by natural language processing (NLP) (e.g., one-hot embeddings). My results show that by drawing inspirations from ML and NLP, my technique substantially outperformed existing state-of-the-art techniques. Similarly, my other work on automatic program repair, record and replay, and parameterized unit tests all aim to identify problems in real-world code and help provide solutions to such problems.

6 Future work As long as people write software, there will be bugs, and software testing will be a practical way to detect such bugs. In the future, I plan to focus my research on software engineering in general, with a focus on software testing in particular. My work on flaky tests scratches only the surface of the problems that nondeterminism brings to software development. In the short term, I will work on a variety of challenges that still plague flaky tests, such as more tailored solutions to detect and tame different kinds of flaky tests. In the longer term, I plan to explore the testing of nondeterministic programs and how software development can actually benefit from nondeterminism. Beyond work on nondeterminism, I also plan to explore how humans and tools can better benefit one another.

6.1 Testing Nondeterministic Programs Programs in domains such as machine learning, probabilistic programming, and quantum computing are inherently nondeterministic. It is more difficult to test these programs because test oracles are generally difficult to write for nondeterministic programs. Yet, testing them is important as companies such as Apple, Facebook, Google, and Microsoft are increasingly relying on these programs for many tasks. To develop dependable testing techniques for these programs, I plan to leverage my expertise in nondeterministic flaky tests, which largely focused on nondeterminism in test code or test infrastructure, to bring unique and proven insights to test these programs. For example, my ISSRE 2020 work [12] found that many tests, even though they are nondeterministic, have significantly different failure rates depending on the order in which the tests are run. By exploiting such dependencies, I can use the failure rates as oracles to determine whether a patch is successful or not. Using a similar idea, I plan to develop new fuzzing techniques to generate values that can be used to automatically derive test oracles for different kinds of programs, such as machine-learning based programs. Using the derived test oracles, I can then generate tests for these inherently nondeterministic programs such that these tests can provide dependable testing outcomes without being flaky.

6.2 Benefiting From Nondeterminism Prior work on flaky tests has largely painted them as a problem that developers should get rid of because the tests can nondeterministically pass or fail
for the same code. However, nondeterminism 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 nondeterministic order. These variations can cause flaky-test failures. My techniques on accommodating flaky tests took the first step to accommodate the problems of nondeterminism 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. I plan to explore how nondeterminism would affect many of these approaches and how they can be accommodated to benefit from nondeterminism. For example, how will nondeterminism affect test generation? How can test generation approaches be accommodated in the presence of nondeterminism such that the tests generated can find more bugs, run faster, and be easier for developers to maintain? To help address these problems, I will first conduct empirical studies to understand the effect of nondeterminism on various approaches and use my results to develop accommodation techniques for such approaches.

6.3 Humans and Tools Interaction

To help software development, a substantial amount of work in recent years is focused on improving tool automation. Indeed, such a focus can be beneficial to reduce developers’ workload. However, it is unlikely that many software development tasks can be fully automated in the near future. I propose to rethink how humans can benefit from tools and how tools can benefit from humans. For humans benefiting from tools, my work on test-input generation [21, 23] has demonstrated how WeChat developers can benefit from better tools. For tools benefiting from humans, I have begun crowd-sourcing my dataset of flaky tests [5], so that my dataset can enable the use of techniques that require much data (e.g., machine learning-based ones) for the flaky test problem soon. Beyond having humans interact with tools or vice-versa, I will also explore how tools can enhance the effectiveness of other tools. For example, my work on fixing flaky tests [18] is made possible by my work on detecting flaky tests [8]. I plan to continue exploring the various ways that humans and tools can benefit from one another on many other software development tasks.
References


