iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests

Wing Lam, Reed Oei, August Shi, Darko Marinov, Tao Xie
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{winglam2,reedoei2,awshi2,marinov,taoxie}@illinois.edu

***Abstract*—Regression testing is increasingly important with the wide use of continuous integration. A desirable requirement for regression testing is that a test failure reliably indicates a problem in the code under test and not a false alarm from the test code or the testing infrastructure. However, some test failures are unreliable, stemming from *flaky tests* that can non-deterministically pass or fail for the same code under test. There are many types of flaky tests, with order-dependent tests being a prominent type.**

To help advance research on flaky tests, we present (1) a framework, *iDFlakies*, to detect and partially classify flaky tests; (2) a dataset of flaky tests in open-source projects; and (3) a study with our dataset. iDFlakies automates experimentation with our tool for Maven-based Java projects. Using iDFlakies, we build a dataset of 422 flaky tests, with 50.5% order-dependent and 49.5% not. Our study of these flaky tests finds the prevalence of two types of flaky tests, probability of a test-suite run to have at least one failure due to flaky tests, and how different test reorderings affect the number of detected flaky tests. We envision that our work can spur research to alleviate the problem of flaky tests.

I. Introduction

Regression testing is becoming increasingly important and popular as both industry and the open-source community widely adopt continuous integration (CI) [30], [49]. When developers make changes to code, CI runs tests on the code version with the changes to check whether the changes introduce regressions. In an ideal world, failures from regression tests would reliably signal faults in the developer's latest changes, be they in the code under test or the test code, and every test failure would warrant investigation.

Unfortunately, some test failures may not be due to the latest changes but due to so-called *flaky tests*. Previous work [20], [38] defines flaky tests as tests that may non-deterministically pass or fail even on the *same* version of the code under test. Both practitioners and researchers are increasingly reporting problems with flaky tests [5], [11], [12], [14], [20], [22], [34], [36], [38], [40]–[43], [46], [53], [56], [57], including from large organizations such as Facebook [25], Google [18], [19], Huawei [32], Microsoft [26]–[28], and Mozilla [16]. A key problem is that flaky tests lead to unreliable signals from CI and can erode the trust of developers in their regression testing.

Flaky-test failures stem from a variety of causes, including faults in the code under test, faults in the test code [38], or unreliable testing infrastructure [32], [34]. Despite the false alarms that flaky tests can raise, these tests can sometimes detect real faults in the code under test and are therefore kept in regression test suites. Frequent causes of non-determinism that lead to flaky tests include concurrency, test-order dependency, resource leaks, real time, network/IO issues, etc. [38].

Previous work [38], [46], [56] on flaky tests points out that one substantial cause of flaky tests is the presence of test-order dependencies. Such dependencies can make the same tests (say, t_1 and t_2) pass when run in one order (say, t_1 before t_2) but fail when run in another order (say, t_2 before t_1) because of some resource shared between the tests (e.g., the state in the main memory or on the file system). Following prior work [56], we refer to flaky tests whose *only* source of non-determinism is order dependencies as *order-dependent (OD) tests*. Unlike other types of flaky tests whose causes of non-determinism may be hard to control, OD tests can deterministically pass or fail depending on the order in which the tests are run [35]. We refer to all other types of flaky tests, which are not OD tests, as *non-order-dependent (NOD) tests*.

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 [17] and Defects4J [33] that enabled research studies on regression testing, automated debugging, and program repair. Some prior work studies only flaky tests from older code versions [14] or focuses on only flaky tests that had been fixed [38]. Other work does not classify flaky tests into OD or NOD tests, or performs studies on only a relatively small number of projects [46], [56].

To offer a dataset of current flaky tests in real-world projects, we develop a framework, called *iDFlakies*, for collecting flaky tests from a large number of open-source projects, and create a dataset of such flaky tests. The core of our framework is our tool 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. Our current tool does not further classify the causes of the flaky tests. 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 ones. 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. [56], our tool reorders tests using random orderings or reversing the original order of the tests. However, our tool differs from

Zhang et al.'s tool in that our tool'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. Our study (Section VI) shows that first randomizing the test classes and then the test methods within each test class can detect the most flaky tests overall.

In addition to our core tool, our framework offers automated experimentation using our tool. In particular, our framework takes a list of URLs of open-source Java projects and a commit for each project, builds a Docker image for running each project, and then runs our tool with a given set of configurations to detect flaky tests for each project. We wrap the experimental runs into Docker images to increase the reproducibility of our results. Furthermore, we make our framework publicly available [7], allowing researchers to easier use our tool in their research experiments on flaky tests.

We apply our framework on 683 projects, and limit the cost of exploring different reordering configurations on them by setting a time limit of up to 56 hours per project. Our experiments find 82 projects with at least one flaky test and a total of 422 flaky tests, of which 50.5% are classified as OD and 49.5% as NOD, based on the observed runs. Moreover, a developer running regression tests often cares not just whether individual tests pass or fail but whether the entire test suite has any test failure. We compute the probability that one run of the test suite leads to flaky-test failures as the percentage of our tool's runs with at least one (flaky) test failure; we find the probability to be as high as 50.0%. We also find that running tests under random orders detects the most overall flaky tests, and it has about the same probability to detect NOD tests as running the tests in the same original order many times.

This paper makes the following main contributions:

Tool. We develop and make publicly available a tool to detect flaky tests; our tool can be easily integrated into Maven projects that use JUnit.

Framework. We present an end-to-end framework, iDFlakies, that researchers can use to easily extend and apply our tool to detect flaky tests and classify them into two types.

Dataset. We describe a collection of artifacts, including Docker images and test-run logs, that we use to create the dataset and detect flaky tests. iDFlakies and our artifacts are publicly available [7].

Study. We present a study of flaky tests in open-source Java projects. Our findings include the prevalence of OD and NOD types of flaky tests and how to automatically detect these tests.

II. Examples of Flaky Tests

We show two example flaky tests—one order-dependent (OD) and one non-order-dependent (NOD)—that can non-deterministically pass or fail when run on the same code.

A. 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 [56], i.e., OD tests can be made to deterministically pass or fail by fixing the order of tests [21]. 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 shorten time of regression testing—including test-suite reduction [48], [50], [51], [54], test selection [15], [23], [37], [44], [45], [47], [52], [55], and test parallelization [4]—select only a subset of tests to run and could additionally expose failures of OD tests.

```
1  public void assertIsShutdownAlready() {
2   shutdownListenerManager.new
        InstanceShutdownStatusJobListener().dataChanged("/
        test_job/instances/127.0.0.1@-@0", Type.
        NODE_REMOVED, "");
3   verify(schedulerFacade, times(0)).shutdownInstance();
4  }
```

Fig. 1. An OD test from the Elastic-Job [3] project.

```
1  @Test(timeout=2000)
2  public void testIssue() throws Exception {
3   final int port = SocketUtil.getAvailablePort();
4   WebSocketServer server = new WebSocketServer( new
        InetSocketAddress( port ) ) { ...}
5    ...
6  }
```

Fig. 2. An NOD test from the Java WebSockets [8] project.

Figure 1 shows an example OD test that our tool found in a project from our study. Elastic-Job [3] is a popular Java project with over 4500 stars on GitHub as of January 2019. The project contains an OD test in its `ShutdownListenerManagerTest` class. The test, `assertIsShutdownAlready`, is OD because its passing depends on some tests *not* to run before it. This dependence exists because Line 3 of `assertIsShutdownAlready` checks whether an instance of a class variable is shut down, and the instance is started by some other tests. The test passes by itself or in orders where the tests that start the instance are run after the OD test. However, the test fails when the tests that start the instance, but do not shut it down, are run directly before `assertIsShutdownAlready`.

B. Non-order-dependent (NOD) test

NOD 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. Tests of this type are flaky due to concurrency, timeouts, network/IO, etc. [38].

Figure 2 shows an example NOD test that our tool found in a project from our study. Java WebSockets [8] is a popular Java project with over 4800 stars on GitHub as of January 2019. The project contains an NOD test in its `Issue713Test` class. The test, `testIssue`, is NOD because it is given a timeout on Line 1, and on some executions of the test, the setup of the WebSocket server and the broadcasting of some messages take so long that the test times out. However, in some executions, even on the same machine, the test is able to finish, due to differences in the load of the machine.

A test could be flaky depending on both the order of the tests and some other non-deterministic cause(s). For example, a flaky test may pass or fail only in a certain order, while always passing or always failing in all other orders. We aim to classify

such tests as NOD, because unlike pure OD tests, the execution results for these tests cannot be deterministically reproduced in some order. However, our tool classifies tests based on the observed runs, and if it does not encounter a relevant run, it may mis-classify a test as OD when it depends on both the test order and some other non-deterministic cause(s).

III. OUR TOOL

We develop a tool that detects flaky tests and classifies each as either OD or NOD; our tool does not further classify the NOD tests into more precise causes of flaky tests [38]. As inputs, our tool 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 III-A. As output, our tool 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, our tool 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 our tool as a Maven plugin that can be integrated into any project that builds using Maven and runs tests using JUnit. A Maven project is organized into one or more *modules*, and each module contains its own code and tests. Like most Maven plugins, our tool runs separately on each module. Our tool uses our own custom test runner to control the order of running JUnit test methods, hence, our tool can work on only Maven projects whose tests are written using JUnit. There are three main steps in our tool. The *setup* step checks whether all tests of a module pass; if not, our tool 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), our tool 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, our tool checks whether all tests pass in the *original order*. To determine this order, our tool runs Maven's unit-test plugin, Surefire [10], and collects the test logs (that Surefire stores in `.xml` files). From these logs, our tool extracts the original order in which Surefire ran test classes and test methods within the classes. 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. A test could fail in the original order due to being flaky but also due to 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 our tool 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 3 shows an example run of our tool, 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 `ATest` (`ATest#t1`) and an NOD test `t3` from `BTest` (`BTest#t3`).

To classify each failed test, the classification step can rerun 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 could not immediately determine whether the test fails due to the change in the order 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 2 could time out twice in a row due to the machine load, independent of the test order. Our tool 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.

A. Configurations

Our tool has five configurations for ordering tests:
(1) **original-order** repeatedly runs the tests in the original order and classifies any failing test as NOD. The configuration cannot detect OD tests, because the order is always the same.

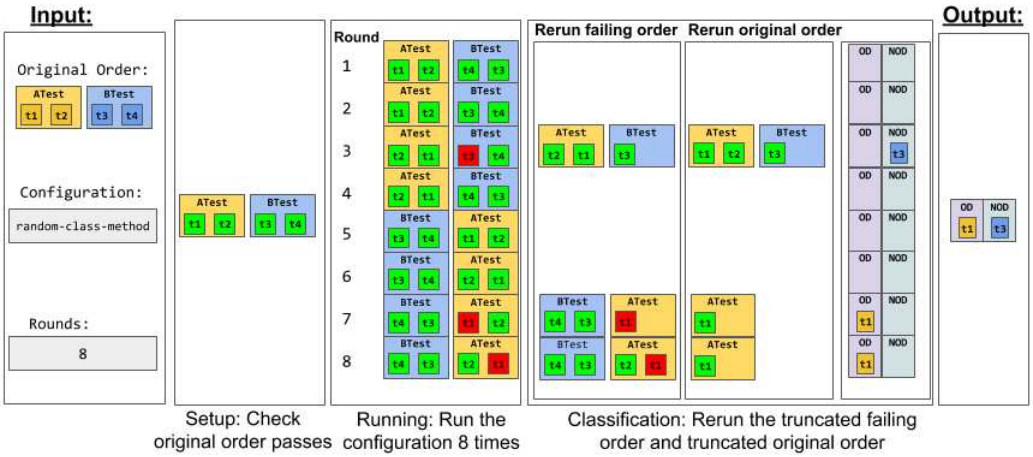

Fig. 3. A sample run of our tool using the random-class-method configuration with 8 rounds, detecting an OD test and an NOD test.

(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 `2` and `5` from Figure 3). 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. Figure 3 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 3); our tool 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 3); 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 III-B). The original-order configuration skips the classification step because all failing tests from this configuration are classified as NOD tests.

B. Classification

When our tool 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, our tool 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, our tool classifies the test as an OD test. Otherwise, our tool classifies it as an NOD test.

The test classification can happen at 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 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 our tool 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.

C. Rounds and timeouts

Our tool 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 VI-D 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

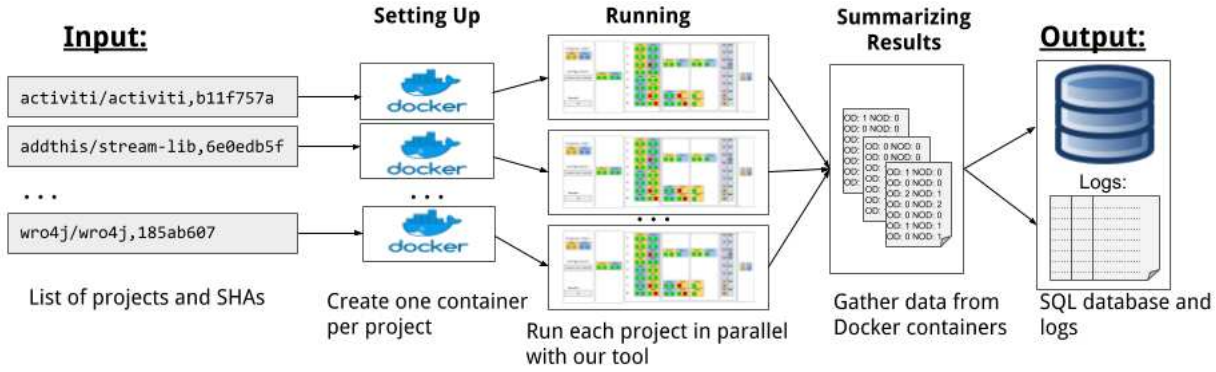

Fig. 4. Overview of the iDFlakies framework.

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, our tool first calculates the number of rounds as for Timeout, and then chooses the minimum of that calculated number and the given number of rounds.

IV. iDFlakies Framework

In addition to our core tool, we also develop a framework, iDFlakies, for using the tool 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 4 shows an overview of the framework. It has three main steps: (1) *setup* of the projects, (2) *running* our tool on the projects to detect flaky tests, and (3) *summarizing* the results for the user. The code for all three steps is publicly available [7].

A. 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 [2] 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 [7].

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.

B. Running step

The running step runs our tool 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 our Maven plugin. Next, the framework determines the number of rounds to run our tool; in the Timeout or Both modes (Section III-C), 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 our tool for each tool configuration that the user specified.

C. 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 our tool 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 [7].

V. Study Setup

All projects in our study are Java projects that build with Maven [9] and use JUnit. 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 recent related work [14], [46], (2) 150 most popular Java projects from GitHub [6] up until October 2018, and (3) 500 most popular Java projects from GitHub that were updated within the month of November 2018. We determine the popularity of GitHub projects using the number of stars.

The projects from related work [14], [46] are prominent Java projects that have flaky tests. Instead of using the same, mostly old, versions of the projects used in the prior work [14], [46], 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 [14], [46]. 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 our tool 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, iDFlakies is able to explore 945 modules for flaky tests. iDFlakies 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 its `pom.xml` file or have no tests, 476 modules' tests do not pass in any of three rounds in the original order, and 458 modules encounter some limitations of our tool. We plan to improve the tool to handle more modules in the future.

In summary, among the 945 modules that iDFlakies 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 [7] provides more details for all projects used in our study.

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

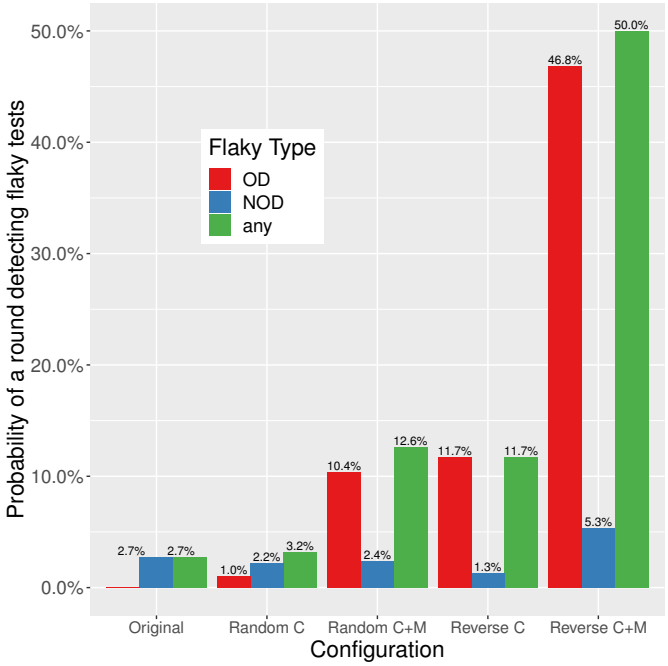

Fig. 5. Probability of a configuration to detect at least one flaky test.

As described in Section V, 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 VI-A) uses both sets of our dataset, while RQ2 and RQ3 (Sections VI-B and VI-C, respectively) use only the comprehensive set, because they compare the configurations.

A. 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 VI-C 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.

B. 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 5 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 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".

Figure 5 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 the reverse-class-method configuration. 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 or a "good" test is not run before it. 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 [12], [13], [24]. 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: 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 plan to further explore the notions of "bad" and "good" tests in the future.

C. RQ3. Configurations detecting most flaky tests

Table I 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, our tool runs each of those two configurations for only one round in one experiment, but we performed multiple experiments while developing our tool and kept most of the logs to provide the largest dataset for analysis of flaky tests.

OD tests: As shown in Table I, 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 opportunities for different reorderings to expose OD tests. In general, Table I 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 configuration. 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, and we plan to explore them in the future.

TABLE I
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**	**NOD**	**OD**	**NOD**	**Round**	**OD**	**NOD**	**OD**	**NOD**	**OD**	**NOD**	**OD**	**NOD**	**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**

NOD tests: Table I and Figure 5 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 configuration 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 our tool.

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 our tool classifies 91 as NOD tests using the classification step; the remaining 31 unique NOD tests need no classification step because our tool 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 from the (33) tests including both comprehensive and extended sets. We plan to explore in the future whether running the tests in different reorderings leads to differences that can more easily expose flakiness in the NOD tests due to various causes.

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.

D. Discussion

Running iDFlakies: Currently, our tool 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), our tool 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 our tool on every module once before running our tool on every module again for the second round, and so on. However, breadth-first would invoke our tool, and consequently Maven, each time it needs to run through all modules for one round. Invoking our tool 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 our tool can finish running tests for one module before proceeding to the next one. The advantage of breadth-first is that it allows for the usage scenario where a developer runs 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 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: Our tool 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 our tool 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. We plan to explore automated debugging of OD tests in the future.

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 [38] classifying fixed flaky tests found a much lower percentage of flaky tests being OD, 12%. Our tool uses random orderings to focus on detecting OD tests. Our tool likely misses many NOD tests and can be improved by adding more variations to test runs in order to detect more NOD tests.

FixMethodOrder: We find that 23 of the OD tests detected by our tool 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. Our tool 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 ones are OD. For example, we observe that while our tool detected several OD tests in `@FixMethodOrder`-annotated test classes from the `Activiti/Activiti` project [1] 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`.

VII. Threats to validity

Our 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 core tool but less confident in the summarizing step of the framework due to its complexity.

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 our tool 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 [7].

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 our tool 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 we study may be (much) higher than what we report. We likely miss some NOD flaky tests. 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, 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.

VIII. Related work

Luo et al. [38] performed an extensive study of flaky tests by looking through historical commits with fixed flaky tests and classified these tests into several types, including OD tests. OD tests were among the top three causes of flaky tests, with 12% of the fixed flaky tests being OD. Labuschagne et al. [34] studied regression testing in continuous integration and encountered many flaky tests as well, reporting that 13% of the historical failed builds rerun on Travis CI are due to flaky-test failures. Gao et al. [22] studied system tests and found that 96% of test failures can be due to flaky tests and that the same tests can have as much as 184 lines of code-coverage difference between runs. Others [29], [39] also report non-deterministic code coverage.

There is a growing body of work on detecting flaky tests. Our work follows Zhang et al. [56] who detected OD tests through random ordering of all test methods in the test suites, with follow-up work [35] that also reported NOD tests through rerunning the same order of tests many times. We also rely on random ordering to detect flaky tests and on reruns to classify them into OD and NOD tests. However, unlike Zhang et al. [56], we build our dataset across a much larger number of projects (683 projects vs. 4 projects). Furthermore, our random-class-method configuration does not interleave the test methods across different test classes, respecting how JUnit actually runs tests. Our results still confirm Zhang et al.'s finding that running a high number of randomized orders can detect more OD tests than running once the reverse order.

Gyori et al. [24] proposed the PolDet technique for detecting tests that "pollute" the state such that tests that run after polluters may have different outcomes. PolDet can proactively report potential test-order dependencies before they even occur, because no other test in the current test suite may actually depend on the polluted state. Dually, Huo and Clause [31] studied tests with brittle assertions that depend on the values derived from inputs not controlled by the tests themselves. Such tests can be order-dependent on other tests that conceptually pollute the state that affects the test with the brittle assertions. Bell et al. [13] proposed to monitor test executions to dynamically detect test dependencies, but these dependencies may not necessarily lead to different test outcomes if tests run in different orders. Building on that work, Gambi et al. [21] developed a technique to more precisely detect test dependencies and used it to find different test orders to manifest OD tests. They reported 27 previously unknown OD tests; we report 213 OD tests and also 209 NOD tests. In future work, we can leverage the ideas from Gambi et al. [21] to detect OD tests faster.

Palomba and Zaidman [46] considered the relationship between code smells and flaky tests, reporting that fixing certain types of code smells can also fix certain flaky tests. They reported finding flaky tests through reruns and classifying them into the types introduced by Luo et al. [38]. Palomba and Zaidman reported 11% of flaky tests to be OD and a large number of flaky tests from a small number of projects. However, they did not provide us the logs of test runs from their experiments, and thus we cannot directly compare their results against ours or investigate the differences.

Bell et al. [14] leveraged code evolution and code coverage to determine whether new test failures between two commits are due to flaky tests; they automatically detected flaky tests that cover no changed parts of the code but have a different outcome from the last time the tests run. In contrast, we detect flaky tests through reruns on the same commit, but instead of just running the tests in the Maven-specified order, we apply various configurations to reorder the tests. Both Palomba and Zaidman [46] and Bell et al. [14] released datasets of flaky tests but for older code versions; our dataset [7] is for the most recent code versions, includes classification of flaky tests into OD and NOD, and contains a collection to artifacts to help others reproduce these flaky tests.

IX. Conclusion

We have presented our *iDFlakies framework*, which automates experimentation to detect and partially classify flaky tests using our tool 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 [7], 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.

Acknowledgments

We thank Angello Astorga, Liia Butler, and Owolabi Legunsen for their discussions about flaky tests. This work was partially supported by National Science Foundation grants CCF-1421503, CNS-1513939, CNS-1564274, CNS-1646305, CNS-1740916, CCF-1763788, CCF-1816615, and OAC-1839010. We acknowledge support for research on flaky tests and test quality from Facebook, Google, Huawei, and Microsoft.

References

[1] Activiti. https://github.com/activiti/activiti.

[2] Docker. https://www.docker.com.

[3] Elastic-Job. https://github.com/elasticjob/elastic-job-lite.

[4] Executing unit tests in parallel on a multi-CPU/core machine in Visual Studio. http://blogs.msdn.com/b/vstsqualitytools/archive/2009/12/01/executing-unit-tests-in-parallel-on-a-multi-cpu-core-machine.aspx.

[5] Flakiness Dashboard HOWTO. http://www.chromium.org/developers/testing/flakiness-dashboard.

[6] GitHub. https://github.com.

[7] iDFlakies: Flaky test dataset. https://sites.google.com/view/flakytestdataset/home.

[8] Java WebSockets. https://github.com/tootallnate/java-websocket.

[9] Maven. https://maven.apache.org.

[10] Maven Surefire plugin. https://maven.apache.org/surefire/maven-surefire-plugin.

[11] TotT: Avoiding Flakey Tests. http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html.

[12] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *ICSE*, pages 550–561, Hyderabad, India, June 2014.

[13] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*, pages 770–781, Bergamo, Italy, Sept. 2015.

[14] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. DeFlaker: Automatically detecting flaky tests. In *ICSE*, pages 433–444, Gothenburg, Sweden, May 2018.

[15] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on UML designs. *IST*, 51(1):16–30, Jan. 2009.

[16] G. Brown. Test Verification. https://developer.mozilla.org/en-us/docs/mozilla/qa/test_verification.

[17] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

[18] S. Elbaum, A. Mclaughlin, and J. Penix. The Google dataset of testing results. https://code.google.com/p/google-shared-dataset-of-test-suite-results.

[19] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, pages 235–245, Hong Kong, Nov. 2014.

[20] M. Fowler. Eradicating non-determinism in tests. http://www.chromium.org/developers/testing/flakiness-dashboard.

[21] A. Gambi, J. Bell, and A. Zeller. Practical test dependency detection. In *ICST*, pages 1–11, Vasteras, Sweden, Apr. 2018.

[22] 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*, pages 55–65, Florence, Italy, May 2015.

[23] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, pages 211–222, Baltimore, MD, USA, July 2015.

[24] A. Gyori, A. Shi, F. Hariri, and D. Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*, pages 223–233, Baltimore, MD, USA, July 2015.

[25] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*, pages 1–23, Madrid, Spain, Sept. 2018.

[26] B. Harry. How we approach testing VSTS to enable continuous delivery. https://blogs.msdn.microsoft.com/bharry/2017/06/28/testing-in-a-cloud-delivery-cadence.

[27] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *ICSE*, pages 483–493, Florence, Italy, May 2015.

[28] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *ICSE*, pages 39–48, Florence, Italy, May 2015.

[29] M. Hilton, J. Bell, and D. Marinov. A large-scale study of test coverage evolution. In *ASE*, pages 53–63, Montpellier, France, Sept. 2018.

[30] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*, pages 426–437, Singapore, Sept. 2016.

[31] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, pages 621–631, Hong Kong, Nov. 2014.

[32] 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*, pages 712–723, Buenos Aires, Argentina, May 2017.

[33] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*, pages 437–440, San Jose, CA, USA, July 2014.

[34] 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*, pages 821–830, Paderborn, Germany, Sept. 2017.

[35] W. Lam, S. Zhang, and M. D. Ernst. When tests collide: Evaluating and coping with the impact of test dependence. Technical Report UW-CSE-15-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Mar. 2015.

[36] T. Lavers and L. Peters. *Swing Extreme Testing*. Packt Publishing, 2008.

[37] 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*, pages 583–594, Seattle, WA, USA, Nov. 2016.

[38] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, Hong Kong, Nov. 2014.

[39] P. Marinescu, P. Hosek, and C. Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *ISSTA*, pages 93–104, San Jose, CA, USA, July 2014.

[40] A. M. Memon and M. B. Cohen. Automated testing of GUI applications: models, tools, and controlling flakiness. In *ICSE*, pages 1479–1480, San Francisco, CA, USA, May 2013.

[41] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. Taming Google-scale continuous testing. In *ICSE SEIP*, pages 233–242, Buenos Aires, Argentina, May 2017.

[42] J. Micco. Continuous Integration at Google scale. https://eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf.

[43] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. In *ESEC/FSE NIER*, pages 496–499, Szeged, Hungary, Sept. 2011.

[44] A. Nanda, S. Mani, S. Sinha, M. J. Harrold, and A. Orso. Regression testing in the presence of non-code changes. In *ICST*, pages 21–30, Berlin, Germany, Mar. 2011.

[45] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, Newport Beach, CA, USA, Nov. 2004.

[46] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *ICSME*, pages 1–12, Shanghai, China, Sept. 2017.

[47] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu. On test suite composition and cost-effective regression testing. *TOSEM*, 13(3):277–331, July 2004.

[48] 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*, pages 34–43, Bethesda, MD, USA, Nov. 1998.

[49] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous deployment at Facebook and OANDA. In *ICSE-Companion*, pages 21–30, Austin, TX, USA, May 2016.

[50] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, Hong Kong, Nov. 2014.

[51] A. Shi, A. Gyori, S. Mahmood, P. Zhao, and D. Marinov. Evaluating test-suite reduction in real software evolution. In *ISSTA*, pages 84–94, Amsterdam, Netherlands, July 2018.

[52] A. Shi, S. Thummalapenta, S. Lahiri, N. Bjorner, and J. Czerwonka. Optimizing test placement for module-level regression testing. In *ICSE*, pages 689–699, Buenos Aires, Argentina, May 2017.

[53] P. Sudarshan. No more flaky tests on the Go team. http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.

[54] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *ICSE*, pages 41–50, Sorrento, Italy, Apr. 1995.

[55] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *ISSTA*, pages 331–341, Minneapolis, MN, USA, July 2012.

[56] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA*, pages 385–396, San Jose, CA, USA, July 2014.

[57] C. Ziftci and J. Reardon. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale. In *ICSE SEIP*, pages 113–122, Buenos Aires, Argentina, May 2017.