A Large-Scale Longitudinal Study of Flaky Tests

WING LAM, University of Illinois at Urbana-Champaign, USA
STEFAN WINTER, TU Darmstadt, Germany
ANJIANG WEI, Peking University, China
TAO XIE, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
DARKO MARINOV, University of Illinois at Urbana-Champaign, USA
JONATHAN BELL, Northeastern University, USA

Flaky tests are tests that can non-deterministically pass or fail for the same code version. These tests undermine regression testing efficiency, because developers cannot easily identify whether a test fails due to their recent changes or due to flakiness. Ideally, one would detect flaky tests right when flakiness is introduced, so that developers can then immediately remove the flakiness. Some software organizations, e.g., Mozilla and Netflix, run some tools—*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 combat the high cost of applying detectors, these organizations typically run a detector solely on newly added or directly modified tests, i.e., not on unmodified tests or when other changes occur (including changes to the test suite, the code under test, and library dependencies). However, it is unclear how many flaky tests can be detected or missed by applying detectors in only these limited circumstances.

To better understand this problem, we conduct a large-scale longitudinal study of flaky tests to determine when flaky tests become flaky and what changes cause them to become flaky. We apply two state-of-the-art detectors to 55 Java projects, identifying a total of 245 flaky tests that can be compiled and run in the code version where each test was added. We find 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 always applied to all tests. Our study is the first to empirically evaluate when tests become flaky and to recommend guidelines for applying detectors in the future.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging*; *Software evolution.*

Additional Key Words and Phrases: flaky test, regression testing

ACM Reference Format:
Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A Large-Scale Longitudinal Study of Flaky Tests. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 202 (November 2020), 29 pages. https://doi.org/10.1145/3428270

Authors' addresses: Wing Lam, University of Illinois at Urbana-Champaign, USA, winglam2@illinois.edu; Stefan Winter, TU Darmstadt, Germany, sw@cs.tu-darmstadt.de; Anjiang Wei, Peking University, China, weianjiang@pku.edu.cn; Tao Xie, Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China, taoxie@pku.edu.cn; Darko Marinov, University of Illinois at Urbana-Champaign, USA, marinov@illinois.edu; Jonathan Bell, Northeastern University, USA, bellj@gmu.edu.

2475-1421/2020/11-ART202
https://doi.org/10.1145/3428270

1 INTRODUCTION

To find bugs and regressions during software development, developers run tests that execute the software under test (SUT) and check the results. If all tests pass, the new changes are allowed to be merged to the SUT. However, if some test fails, then the developers typically debug their changes to locate and remove the cause of the failure. Unfortunately, not all tests provide a consistent result, and may fail for reasons other than a recent change. A test that can both pass and fail in repeated runs, on the same SUT (even without new changes), is known as a *flaky test* [Luo et al. 2014].

When a flaky test fails, the developers, unaware of the flakiness at first, can start debugging for the cause of the failure within their changes. In particular, during regression testing, if a test that passed on a previous code revision fails on the new code revision, the developers would typically debug for the cause in the changes between the two revisions. Unfortunately, this debugging can take substantial time and effort, because the new changes are not the (sole) cause of failure for flaky tests, essentially (mis)leading developers to debug the failure in the wrong place.

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 [Harman and O'Hearn 2018] and recently ran a call for research projects focused on flaky tests [FacebookFlakyTestCall 2019]. Several papers and blog posts from Google [AvoidingFlakeyTests 2019; FlakinessDashboardHOWTO 2020; Micco 2020; Ziftci and Reardon 2017] and Microsoft [Harry 2019; Lam et al. 2019a, 2020a] have reported several challenges with flaky tests, even estimating the monetary cost of flaky tests on developer productivity [Herzig et al. 2015]. Huawei test logs were analyzed for flaky test failures by Jiang et al. [2017]. Other organizations, including Mozilla [Eck et al. 2019; Rahman and Rigby 2018], Netflix [NetflixAutomationTalk 2017], Salesforce [SalesforceFlakyTests 2016], and ThoughtWorks [Sudarshan 2012], also publicly report their problems with flaky tests.

Much existing work [Coverity 2014; Infer 2020; Larus et al. 2004; Memon et al. 2017; Muşlu et al. 2015; Sadowski et al. 2015; Saff and Ernst 2003] 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 [2020] at Facebook, Harman and O'Hearn [2018] found that bugs reported to developers at *"post land"* (after code has been merged into the SUT) had a close to 0% fix rate, yet bugs reported at *"diff time"* (when the developer submits the code change) had a fix rate of over 70%. In the case of flaky tests, it is likely far more useful to notify developers that a change they are in the process of making introduces test flakiness, rather than to notify developers weeks or months after they made the change that a test is flaky. In fact, the standard testing practice at organizations, such as Mozilla [MozillaChaosMode 2019] and Netflix [NetflixAutomationTalk 2017], already aims to find whether newly added tests are flaky as soon as possible.

Due to the non-deterministic nature of flaky tests, detecting them is quite challenging, and has become an increasingly active area of research. The simplest approach to detect flaky tests is to just rerun tests many times: if different outcomes are observed on two runs, then the test is flaky. However, if we can guess the underlying source of non-determinism that causes a test to be flaky, it can be easier to detect the test by purposefully modifying some relevant environmental properties. For instance, some flaky tests might be *order-dependent*, where some test(s) may write to a shared state (e.g., memory or disk), and this state "pollution" can cause another test to fail depending on the order in which the tests run. Other flaky tests may be *implementation-dependent*, where a test is flaky due to an assumption that an API is deterministic, when that API is not (e.g., the order of iteration over a `HashSet`). Several tools have been proposed for detecting these and other categories of flaky tests [Gyori et al. 2015; Huo and Clause 2014; Lam et al. 2019b; MozillaChaosMode 2019;

NetflixAutomationTalk 2017; Shi et al. 2016; Zhang et al. 2014]. We refer to these tools as *detectors*. Although they make it easier to detect some flaky tests, detectors are costly to apply as they rerun some or all tests in a test suite many times to check for potential failures. At the same time, many organizations are reporting a desire to *decrease* the amount of time taken to run tests [Yoo and Harman 2012], thereby rendering the option of running a test suite multiple times for every code change to be an infeasible option for detecting flaky tests.

One solution to decrease the time to run detectors is not to check for flaky tests on *every* code change, but instead, only when flakiness might be introduced. Unfortunately, it is not easy to determine such checking points. For instance, to better optimize flaky-test detection at "diff time", Mozilla [MozillaChaosMode 2019] and Netflix [NetflixAutomationTalk 2017] treat newly added or directly modified tests differently from existing tests that are not modified. Specifically, these organizations run each newly added or directly modified test some number of times in various environments in isolation from all other tests (e.g., for Mozillia, the default is 10 isolated runs and 5 other additional isolated runs, each in a new execution environment). Although the idea of applying detectors on just newly added tests would substantially reduce the cost of flaky-test detection (i.e., rerunning *only* new tests many times, and not rerunning the entire test suite many times), one important question is how often flaky tests are already flaky when they are added.

Currently, developers have little guidance in terms of predicting *when* a test becomes flaky—is it when the test is added, when the test is modified, or even when the test is not directly modified but something else changes (the test suite, the code under test, or some library dependency)? Our literature search reveals only two research papers [Gambi et al. 2018; Luo et al. 2014] that mention checking whether tests are flaky when they are added, but both studies are quite limited to just a handful of flaky tests that were manually studied, and in at least one study [Luo et al. 2014] tests were not even executed at all. To shed light on the situation, we conduct a large-scale longitudinal study to answer precisely this question: when do tests become flaky and what implications does that answer have for how developers should apply detectors to effectively detect flaky tests?

To understand the potential effectiveness of applying detectors on only newly added tests versus at other points in each test's lifecycle, we perform a study to find flaky tests' *flakiness-introducing commit (FIC)*, the commit in which a flaky test first becomes flaky. We find that a nontrivial number of flaky tests are not flaky in their *test-introducing commit (TIC)*, the commit in which the test was first added, i.e., the test's TIC is not the same as the FIC. Our study uses the Maven-based, Java projects from a dataset provided by our prior work [Lam et al. 2019b].

We apply two flaky-test detectors—NonDex [Shi et al. 2016] and iDFlakies [Lam et al. 2019b]—to these projects and detect 684 potentially flaky tests in 55 projects. We use these two detectors because they can together detect four categories of flaky tests and are state-of-the-art tools for Maven-based, Java projects. The NonDex detector has some similarities with the Mozilla Chaos mode [MozillaChaosMode 2019] in that they both randomize various non-deterministic choices (specifically for standard library methods with non-deterministic specifications). They differ in that NonDex is for Java, while the Mozilla Chaos mode is for C/C++. (Moreover, our personal communication with developers from Google reveals that they also have a detector very similar to NonDex and the detector is also for Java.) In contrast, the iDFlakies detector differs from the Mozilla Chaos mode in that iDFlakies can randomize the order of tests in the entire test suite. We apply each detector on each project's *iDFlakies-commit*, the same commit from the original iDFlakies dataset. As we intend to find all FICs for every flaky test in our study, we do not directly use the iDFlakies dataset of flaky tests because certain test suites were run up to 16,503 times in the iDFlakies dataset, and it would substantially increase the cost of our experiments to run test suites that many times for many commits.

To determine whether a flaky test is flaky at any particular commit, we use the following approaches to rerun and *categorize* each test:

(1) Isolation, where every test is repeatedly run by itself, as a representative technique for detecting order-dependent brittle (OD Brit) tests [Shi et al. 2019] and for how developers [MozillaChaosMode 2019] typically detect flaky tests;
(2) One-By-One (OBO), where each flaky test is run with every other test, as a representative technique for detecting order-dependent victim (OD Vic) tests [Shi et al. 2019]; and
(3) NonDex [Shi et al. 2016] as a representative technique for detecting implementation-dependent (ID) tests.

Besides the three categories of tests that these three approaches focus on, all three approaches can also detect, through multiple reruns of the tests, non-deterministic (ND) tests, which can non-deterministically pass or fail because of other causes. We also use all three of these approaches on the iDFlakies-commit to categorize each flaky test into one of the four categories so that we can evaluate how different categories of flaky tests may affect when one should run detectors (RQ2).

Our study finds that 245 out of 684 flaky tests can be compiled and run on the test's TIC. Some of the tests cannot be compiled or run on older commits for various reasons (Section 4), e.g., the code cannot compile because an old library dependency is no longer available. Of these 245 tests, we find that 184 (75%) are flaky when they are first added to the project. That is, if developers used detectors solely on newly added tests, and not on existing modified or unmodified tests, they could detect 75% of the flaky tests that we study.

To understand how one may detect the remaining 25% of flaky tests, we proceed to study when such tests become flaky and what changes cause them to become flaky. Specifically, for all tests that are not identified as flaky in the TIC, we traverse the commit history between TIC and iDFlakies-commit to find the FIC. We find that tests that are not flaky when added can become flaky at a varying point in the future, from the immediate next commit of the TIC to tens of thousands of commits after the TIC. With the FICs, we then study how many of them involve changes to the code of the flaky test. We find that if developers were to apply detectors on newly added tests and tests that have been modified, then the detectors can detect 85% of the flaky tests that we study (a 10pp increase from just applying detectors on newly added tests). For the remaining 15% of flaky tests in our study, we find that these tests are not flaky in their TIC, and the changes that cause these tests to be flaky are not directly made to the test itself.

The topic of flaky test research is relatively new, and this study provides quantitative empirical evidence for the hypotheses that underpin much work on this topic. Our foundational study can inspire researchers to build new detectors that focus on finding flaky tests early, and we confirm developers' practices for running detectors. We also contribute a dataset of flaky tests with labeled TICs and FICs on our website [FlakyTestFICWebsite 2020], so future research can use them for evaluations and to develop techniques to better detect and categorize these tests. To help researchers and developers on this topic, this paper 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?

We additionally discuss (Section 7) several examples of flaky tests that are not flaky when introduced (TIC) but become flaky in a later commit (FIC).

2 STUDY SETUP

Flaky tests should be detected as soon as the change that introduces the flakiness is made. Developers at Mozilla [MozillaChaosMode 2019] and Netflix [NetflixAutomationTalk 2017] 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 [Gambi et al. 2018; Gyori et al. 2015; Huo and Clause 2014; Lam et al. 2019b; Shi et al. 2016; Zhang et al. 2014]. We next describe the detection approaches that we use to obtain a dataset of flaky tests (Section 2.1), the flaky-test dataset that we use for our study (Section 2.2), and how we categorize flaky tests to reproduce their failures (Section 2.3).

2.1 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) Gambi et al. [2018]; Gyori et al. [2015]; Huo and Clause [2014]; Lam et al. [2019b]; Zhang et al. [2014] have proposed various ways to detect flaky tests whose test result depends on the order the tests are run; such tests are known as *order-dependent* (OD) flaky tests.
(2) MozillaChaosMode [2019]; Shi et al. [2016] have 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 for detecting OD tests and NonDex for detecting ID tests. We refer to flaky-test detection tools as *detectors*. 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, all of the detectors that we find [Gambi et al. 2018; Gyori et al. 2015; Huo and Clause 2014; Lam et al. 2019b; Zhang et al. 2014] are for Java projects and most are for Maven-based projects. We select iDFlakies [Lam et al. 2019b], because it is the most recently published detector and co-authored by us (we are thus familiar with it). For detecting ID tests, we select NonDex [Shi et al. 2016] because it is the only detector for Maven-based, Java projects.

2.1.1 iDFlakies. iDFlakies [Lam et al. 2019b] 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* (ND) – tests that non-deterministically pass or fail with no changes to test execution order or implementation of test dependencies.

2.1.2 NonDex. NonDex [Shi et al. 2016] is a tool for detecting incorrect assumptions by developers on specifications. Some APIs have underdetermined specifications [Liskov and Guttag 2000], 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 not 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* (ND) – tests that non-deterministically pass or fail with no changes to test execution order or implementation of test dependencies.

2.2 Flaky Tests Used in Our Study

As described in Section 2.1, 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 [Lam et al. 2019b]. 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 82 projects that are all selected from GitHub [GitHub 2020] 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. When we refer to *modules* in this paper, we do not refer to the Java Platform Module System [JavaModules 2020], 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.

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 [Lam et al. 2019b], 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). 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 paper, 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 16,503 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 3.2, is the number of runs that we use for each applicable categorization step to reproduce flaky-test failures.

[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 [fastjsonGitIssue 2020].

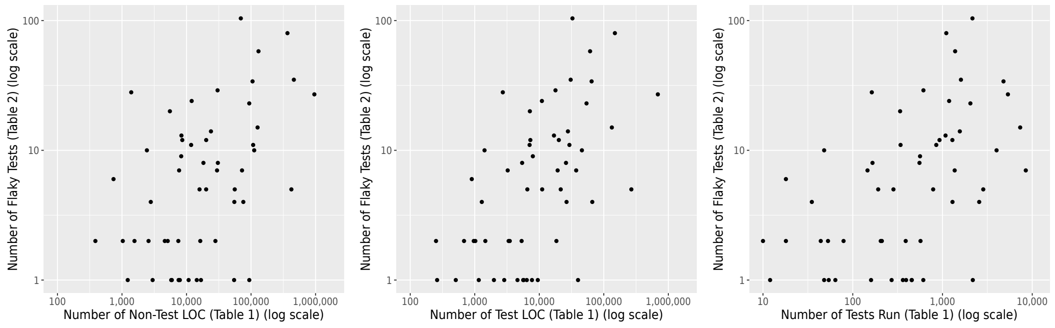


Fig. 1. Relation of project characteristics and flaky test counts for projects with one or more flaky tests.

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 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 684 flaky tests from 55 projects. Table 1 shows the overall characteristics of these 55 projects, and Table 2 shows the number of flaky tests detected per project. The projects have a wide range of sizes (from only 387 to 958,112 lines of Non-Test code[2]), and their test suites can be quite big (up to 8,471 tests[3]).

Using Kendall's rank correlation coefficient, we investigate the relationship between the project characteristics (Table 1) and the number of flaky tests detected (Table 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 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 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 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.

[2]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.
[3]Tests have been counted as test methods run according to Maven Surefire reports.

Table 1. Characteristics of the projects used in our study.

	iDFlakies	**Lines of Code**		**Test**
Project Slug from GitHub	**SHA**	**Non-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**

2.3 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 flakiness-introducing commit 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 2.1.2) 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 ND tests. As described in Section 3.2, 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).

2.3.1 Isolation. Bell [2014]; Bell and Kaiser [2014]; MozillaChaosMode [2019]; Muşlu et al. [2011] have 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 ND tests, because the tests can be run more times in a limited amount of 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 [MozillaChaosMode 2019] and Netflix [NetflixAutomationTalk 2017]. 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 [MozillaChaosMode 2019] developers use by default. For running a specific test in isolation, we simply use an already existing feature in Maven Surefire [Maven 2020], namely `mvn test -Dtest=TestClass#testMethod`.

2.3.2 One-By-One. While iDFlakies can effectively detect flaky tests, it 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. Not covering all permutations is particularly important to our study. 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 work [Shi et al. 2019], 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.

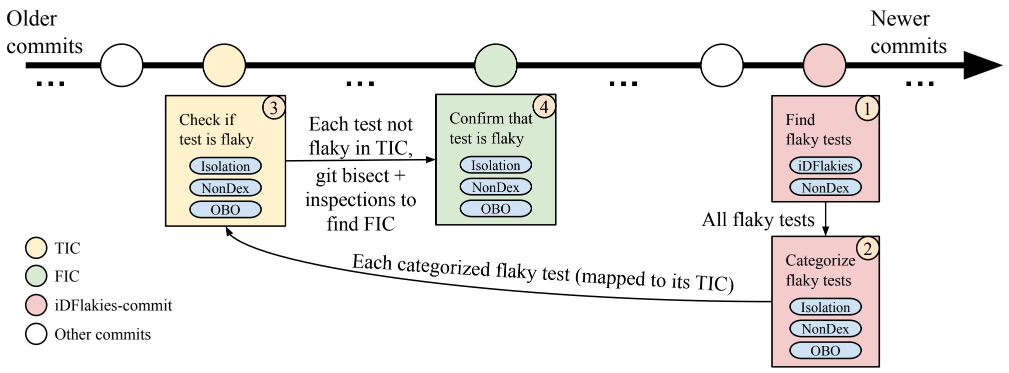


Fig. 2. High-level overview of our study methodology. For Step 1, we use two different tools to detect flaky tests. For Step 2, we categorize each of these flaky tests. For Step 3, we identify the test-introducing commit (TIC) and confirm whether the test was flaky in that commit. For Step 4, we identify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

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[4]. By default, Maven Surefire [Surefire 2020] 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.

3 METHODOLOGY

This section describes how we use the flaky-test dataset described in Section 2.2 and the flaky-test categorization approaches described in Section 2.3 to answer our RQs.

3.1 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 such as iDFlakies [Lam et al. 2019b] and NonDex [Shi et al. 2016]. 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 2.2, we apply the targeted approaches described in Section 2.3 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 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.

3.1.1 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: (1) the location of a test could have changed, e.g., to a different module or package, and (2) test

[4]Zhang et al. [2014] 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.

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.

3.1.2 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` [GitBisect 2020] 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)[5]; 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 ND 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).

3.2 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

[5]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.

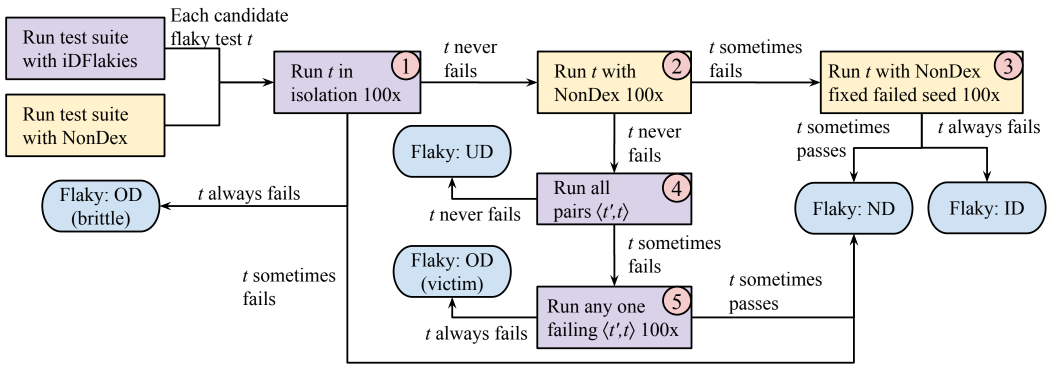


Fig. 3. Procedure for categorizing flaky tests on the iDFlakies-commit.

Figure 3. We perform this categorization on the iDFlakies-commit for each test (before using the approach outlined in Section 3.1 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 2.3). 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 3 shows the five steps we perform to categorize each flaky test into these 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* (ND) – 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 ND. 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 ND. 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 ND. 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 [Shi et al. 2019] 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 [testrunner 2020] so that it can control the order of all test methods, while all of our targeted approaches use Maven Surefire [Surefire 2020] 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 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.

3.3 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 3.1.2, 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 [Zeller 1999] 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.

3.3.1 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 RQ1: HOW EFFECTIVE ARE FLAKY-TEST DETECTORS IF RUN ONLY WHEN TESTS ARE INTRODUCED?

Table 2 shows an overview of the number and category of flaky tests that our experimental procedure (described in Sections 3.1 and 3.2) 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 3.2) 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 3.1.1, and we then attempted to compile that revision of the module under test and to run that test. We did not attempt to run unknown-dependent 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 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 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

Table 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.../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

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 7.1, 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 Sections 6 and 7.

5 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 provide a better understanding of this problem, our experimental procedure (as illustrated in Figure 3 and described in Section 3.2) 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 3.2.

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 (ND), 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 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 NDs, 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, NDs, 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.

Table 3. Flaky tests not detected as flaky on their test-introducing commit (TIC) and the information about their flakiness-introducing commit (FIC). For each FIC, we show which kinds of files are modified by the change: test class containing the flaky test (TC), test suite (TS), code under test (CUT), or a build configuration file (B). For each intervening commit between the TIC and FIC, we tally how many of those commits change each of these kinds of files.

			Files Changed by FIC				**Commits from TIC-FIC Changing**					**Days**
Project	**Test**	**Category**	**TC**	**TS**	**CUT**	**B**	**TC**	**TS**	**CUT**	**B**	**Any**	**TIC-FIC**
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

6 RQ3: WHEN SHOULD ONE RUN FLAKY-TEST DETECTORS?

To better understand when a test becomes flaky, we analyze in detail (mostly manually with some automated tool support) *all* 61 tests that are not flaky on their TIC. In Section 7, 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.

6.1 TIC-FIC Characteristic Differences for All Flaky-Test Categories

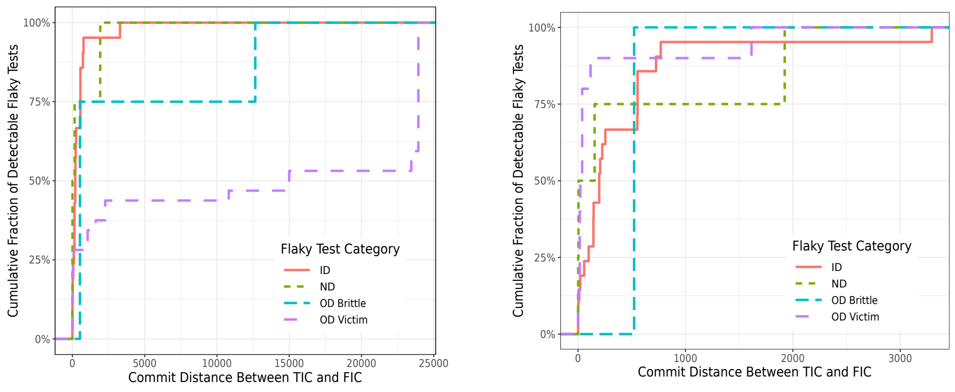
Table 3 shows the results of the analysis of what files the FIC changed (and where the change was that introduced the flakiness). In particular, we distinguish the following types of commits: (1) the commit changed the code of the test itself (note that this need not be just the test method body but 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" [Burckhardt et al. 2010]), so projects typically do not run these detectors on all tests for each and every commit. For example, Mozilla runs its "test verification" [MozillaChaosMode 2019] only on tests newly added or modified in a commit.

As described in Section 4, we find that 75% of flaky tests in our study are detected as flaky on their TIC. From column "TC" (Test Class) in Table 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 3.1.2. 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 the number of days (Section 3.3.1) 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 7,089 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 organizations, such as Mozilla [MozillaChaosMode 2019] and Netflix [Netflix-AutomationTalk 2017], 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,



(a) Including wildfly/wildfly. (b) Excluding wildfly/wildfly.

Fig. 4. Flaky-test detectability over commit distance between TIC and FIC.

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 be detected only 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 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.

6.2 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 TIC), 97% of NDs (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, ND, and ID tests become flaky right after their test code has been added or changed, which shows that how Mozilla [MozillaChaosMode 2019] and Netflix[NetflixAutomationTalk 2017] 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 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 4a includes all data, Figure 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 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 15,000 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 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 ND 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 ND 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 ND 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).

7 CASE STUDIES OF TESTS NOT FLAKY AT TIC

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.

7.1 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).

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 7.2. 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.

7.2 Order-Dependent Brittle

Flakiness introduced due to a dependency change. Consider the test WritableServiceBased-NamingStoreTestCase.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 only 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" [Shi et al. 2019] 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 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.

7.3 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 ND 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. [2014]), 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. [2014]). 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.

7.4 Implementation-Dependent

Resolving compilation problems to identify precise FIC. Consider the test KubeConfigContentServiceTest.testGetKubeConfigWithBearerToken from the project `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 precisely 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. [2014] call this phenomenon *manifest flakiness* and argue that one should report only such cases where a test fails. In contrast, Huo and Clause [2014] and Gyori et al. [2015] 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.

8 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 5), 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 industry 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 [FlakyTestFICWebsite 2020].

8.1 Threats to Validity

We next discuss threats to validity of our study following the classification by Wohlin et al. [2012].

8.1.1 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 [Croux and Dehon 2010]. We conduct a test for the equality of proportions 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 1) and find that its results confirm the validity of the proportionality test's conclusion.

8.1.2 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 [Shi et al. 2019]. 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. Two authors of the paper 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 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.

8.1.3 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 2.2, 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 [Lam et al. 2019b] (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.

9 RELATED WORK

In recent years we have seen a number of studies on the characteristics of flaky tests [Eck et al. 2019; Gao et al. 2015; Jiang et al. 2017; Kowalczyk et al. 2020; Lam et al. 2020a, 2019b, 2020b; Luo et al. 2014]. Luo et al. [2014] appears to be the first academic study on the various categories of flaky tests. That study is based on a *manual investigation* of commits that are likely related to fixing flaky tests, with the commits found by searching for certain flaky-test related keywords. The study reported 78% of flaky tests being flaky in the commits in which the tests are added. In contrast, we *run flaky tests* using Isolation, NonDex, and OBO on the tests' TIC, and we find that 75% of flaky tests are flaky in the commits in which the tests are added. Our methodology for detecting flaky tests differs substantially from Luo et al. [2014] in that we use two state-of-the-art flaky-test detectors, while Luo et al. [2014] found flaky tests by reading changelogs. Also, Luo et al. [2014] reported that 12% of flaky tests are order-dependent (OD), while 88% are not OD. When we apply our three targeted approaches to categorize flaky tests, we find that 26% of flaky tests are OD and 74% are not OD. Note that we apply iDFlakies and NonDex on different projects and different versions from those studied by Luo et al. [2014], and the manual investigation from that study is highly costly to reproduce, so we do not further dissect the differences in our results.

A more recent study on flaky tests interviewed Mozilla developers after they fixed flaky tests in Mozilla-related projects [Eck et al. 2019]. This study extends the one from Luo et al. [2014] by identifying additional categories of flaky tests, based on developer surveys. We do not directly compare our results with this study because the projects substantially differ (e.g., programming languages, build systems, and testing frameworks used). Nevertheless, we believe that such interviews are highly valuable and could help minimize the threat of manual analysis that we performed on flaky tests, so we advocate for more future work with developer interviews.

Another recent work of ours [Lam et al. 2020b] studied flaky tests by rerunning test suites many times to observe whether non-deterministic flaky tests depend on the order in which the tests are run. The study presented in this paper differs in four main ways: (1) this study contains non-deterministic, implementation-dependent, and order-dependent flaky tests, while the other study focuses solely on non-deterministic tests, (2) this study considers multiple commits for each flaky test (e.g., TIC, FIC, and iDFlakies-commit), while the other study focuses solely on the iDFlakies-commit, (3) this study uses projects from the iDFlakies dataset regardless of whether the projects have OD or ND tests, while the other study focuses solely on projects with ND tests, and (4) this study uses iDFlakies and NonDex to detect the flaky tests, while the other study used only Maven Surefire [Surefire 2020] to run tests in various test-class orders.

Our prior work [Lam et al. 2019b] provided the iDFlakies tool for detecting flaky tests in Maven-based Java projects. Using our tool, we studied the frequency of OD and ND tests. We reported that 50.5% of flaky tests are OD, and 49.5% are not. The difference in the frequency of OD tests reported in this paper and our prior work is due to the different number of runs for each test suite, the use of NonDex for detection of flaky tests, and our use of targeted approaches to categorize flaky tests. Focusing on tests that this paper's experiments categorize as ND, while our prior work categorized as OD, we find only three such tests. Our manual inspection confirms that all three are indeed ND, and all have a substantially lower chance of failing when run in isolation than when run after some other tests. One of the three tests can fail when run after one of two polluters. The test always fails when run after one of its polluters, appearing to be OD Vic. However, the test can pass or fail when run after the other polluter, appearing to be ND. The other two tests behave similarly, i.e., either have polluters with substantially different chances for the test to fail, or running the test in isolation has a substantially different chance to fail than running it after polluters. We find no tests that this paper's experiments categorize as OD, while our prior work categorized as ND.

Several projects have proposed techniques for detecting flaky tests [Bell et al. 2018; Gambi et al. 2018; Pinto et al. 2020; Terragni et al. 2020; Zhang et al. 2014]. Specifically, Zhang et al. [2014] proposed randomizing test orders to detect OD tests. Bell et al. [2018] proposed monitoring the changes between revisions to determine whether test failures are due to flaky tests or not. Gambi et al. [2018] proposed using data-flow analysis to detect OD tests. Pinto et al. [2020] proposed using machine learning techniques based on tokens obtained from test-method code to detect flaky tests. Terragni et al. [2020] proposed using various containers, each specialized to detect and root cause a certain category of flaky tests. None of these detectors would be effective if applied immediately on the TIC if the test is not yet flaky. This observation is important because running these techniques 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.

Another line of prior work [Bell 2014; Bell and Kaiser 2014; Muşlu et al. 2011] has proposed isolating tests from each other during their runs. Although doing so may help reduce or even eliminate the likelihood of flaky-test failures, such techniques are typically much slower than running the test without such techniques, even with the recent work that describes how to reduce the overhead of test isolation in Maven Surefire [Nie et al. 2020]. Gyori et al. [2015] proposed PolDet, which monitors the heap and filesystem to detect tests that may become polluters later. Complementary, Huo and Clause [2014] proposed detecting "brittle assertions", which can detect tests that may become either victims or brittles later. These last two techniques could indeed be effective at detecting *potential* flaky tests starting at the TIC. However, these specific techniques focus on the shared heap, while flaky tests have many other sources of flakiness (e.g., random numbers, network, timing, concurrency, or I/O) that still remain to be addressed.

10 CONCLUSION

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 [MozillaChaosMode 2019] and Netflix [NetflixAutomationTalk 2017]. However, the remaining 15% of flaky tests become flaky due to other changes, providing motivation for studying how to better detect such tests in the future.

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 [Burckhardt et al. 2010]. 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 [RSpecIssue635 2020], 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 [FlakyTestFICWebsite 2020].

ACKNOWLEDGMENTS

We thank Pu Yi for helping us inspect specific flaky tests, and Reed Oei and August Shi for general discussions about flaky tests. This work was partially supported by NSF grant nos. CNS-1564274, CCF-1763788, CCF-1763822, CCF-1816615, and CCF-1844880. We acknowledge support for research on flaky tests from Facebook and Google.

REFERENCES

AvoidingFlakeyTests 2019. TotT: Avoiding flakey tests. http://goo.gl/vHE47r.

Jonathan Bell. 2014. Detecting, isolating, and enforcing dependencies among and within test cases. In *FSE*.

Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*.

Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*.

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*.

Coverity 2014. Static analysis in industry. http://popl.mpi-sws.org/2014/andy.pdf

Christophe Croux and Catherine Dehon. 2010. Influence functions of the Spearman and Kendall correlation measures. *Statistical methods & applications* (2010).

Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *ESEC/FSE*.

FacebookFlakyTestCall 2019. Facebook testing and verification request for proposals. https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019

fastjsonGitIssue 2020. fastjson - Git issue. https://github.com/alibaba/fastjson/issues/2584

FlakinessDashboardHOWTO 2020. Flakiness dashboard HOWTO. http://www.chromium.org/developers/testing/flakiness-dashboard

FlakyTestFICWebsite 2020. A Large-Scale Longitudinal Study of Flaky Tests - Tools and Dataset. https://sites.google.com/view/first-commit-flaky-test

Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*.

Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE*.

GitBisect 2020. Git bisect. https://git-scm.com/docs/git-bisect

GitHub 2020. GitHub. https://github.com

Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*.

Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*.

Brian Harry. 2019. How we approach testing VSTS to enable continuous delivery. https://blogs.msdn.microsoft.com/bharry/2017/06/28/testing-in-a-cloud-delivery-cadence

Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *ICSE*.

Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*.

Infer 2020. Infer static analyzer. https://fbinfer.com

JavaModules 2020. Java Platform Module System. https://www.oracle.com/corporate/features/understanding-java-9-modules.html

He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In *ICSE*.

Emily Kowalczyk, Karan Nair, Zebao Gao, Leopold Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *ICSE*.

Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019a. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*.

Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020a. A study on the lifecycle of flaky tests. In *ICSE*.

Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019b. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*.

Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020b. Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects. In *ISSRE*.

Jim Larus, Tom Ball, Manuvir Das, Rob DeLine, Manuel Fahndrich, Jon Pincus, Sriram Rajamani, and Ramanathan Venkatapathy. 2004. Righting software. *IEEE Software* (2004).

Barbara Liskov and John Guttag. 2000. *Program development in Java: Abstraction, specification, and object-oriented design.*

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.

Maven 2020. Maven. https://maven.apache.org

Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE SEIP*.

John Micco. 2020. Continuous integration at Google scale. https://eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf

MozillaChaosMode 2019. Test verification. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *ESEC/FSE*.

Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. 2015. Preventing data errors with continuous testing. In *ISSTA*.

NetflixAutomationTalk 2017. Netflix automation talks - Test automation at scale. https://youtu.be/FrBN94gUn_I?t=764

Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the performance of Maven's test isolation: Experience report. In *ISSTA*.

Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests?. In *MSR*.

Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE*.

RSpecIssue635 2020. RSpec-core issue 635. https://github.com/rspec/rspec-core/issues/635

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *ICSE*.

David Saff and Michael D. Ernst. 2003. Reducing wasted development time via continuous testing. In *ISSRE*.

SalesforceFlakyTests 2016. Flaky tests (and how to avoid them). https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60.

August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*.

August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*.

Pavan Sudarshan. 2012. No more flaky tests on the Go team. http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.

Surefire 2020. Maven Surefire plugin. https://maven.apache.org/surefire/maven-surefire-plugin

Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. 2020. A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests. In *ICSE NIER*.

testrunner 2020. TestingResearchIllinois/testrunner. https://github.com/TestingResearchIllinois/testrunner

Claes Wohlin, Per Runeson, Martin Höst, Magnus Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*.

Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification & Reliability* (2012).

Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *ESEC/FSE*.

Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*.

Celal Ziftci and Jim Reardon. 2017. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale. In *ICSE*.