Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects

Wing Lam
University of Illinois
Urbana, IL USA
winglam2@illinois.edu

Stefan Winter
TU Darmstadt
Darmstadt, Germany
sw@stefan-winter.net

Angello Astorga
University of Illinois
Urbana, IL USA
aastorg2@illinois.edu

Victoria Stodden
University of Illinois
Urbana, IL USA
vcs@stodden.net

Darko Marinov
University of Illinois
Urbana, IL USA
marinov@illinois.edu

***Abstract*—Flaky tests are tests that can non-deterministically pass and fail. They pose a major impediment to regression testing, because they provide an inconclusive assessment on whether recent code changes contain faults or not. Prior studies of flaky tests have proposed tools to detect flaky tests and identified various sources of flakiness in tests, e.g., order-dependent (OD) tests that deterministically fail for some order of tests in a test suite but deterministically pass for some other orders. Several of these studies have focused on OD tests.**

We focus on an important and under-explored source of flakiness in tests: non-order-dependent tests that can non-deterministically pass and fail even for the same order of tests. Instead of using specialized tools that aim to detect flaky tests, we run tests using the tool configured by the developers. Specifically, we perform our empirical evaluation on Java projects that rely on the Maven Surefire plugin to run tests. We re-execute each test suite 4000 times, potentially in different test-class orders, and we label tests as flaky if our runs have both pass and fail outcomes across these reruns. We obtain a dataset of 107 flaky tests and study various characteristics of these tests. We find that many tests previously called "non-order-dependent" actually do depend on the order and can fail with very different failure rates for different orders.

***Index Terms*—flaky tests, regression testing, reproducibility**

I. Introduction

Flaky tests are tests that can non-deterministically pass and fail in different test runs, even for the same code under test and the same test environment that the developers can easily control [1], [2]. Such tests are a major impediment to regression testing, because they provide an inconclusive assessment on whether recent code changes contain faults or not [3] as reported by many companies, including Apple [4], Facebook [2], Google [5], [6], [7], [8], Huawei [9], Microsoft [10], [11], [12], and Mozilla [13]. When developers run regression tests after code changes, they expect tests that flip the outcome, i.e., fail after the changes but passed before the changes, to indicate faults in the changes. Unfortunately, flaky tests can fail for reasons that are unrelated to the changes. A common way to handle flaky tests is to *rerun* tests after failures to see if they would pass. For example, Google reports rerunning failed tests ten times [6]. However, little is known about failures of flaky tests, making it difficult for developers to determine an appropriate number of reruns.

Prior work [1], [14], [15] has identified various reasons for the non-deterministic outcomes of flaky tests. These reasons can be broadly split into two groups. One group was called *order-dependent (OD)* tests [16], [17], [18], [19], [20], [21], [22], which deterministically fail for some order of tests in a test suite, but deterministically pass for some other orders. The other group of reasons include timing behavior for tests that depend on real time, concurrency for tests that involve multiple threads, asynchronous wait for tests that involve message passing, random seeds for tests that involve random choices [23], unspecified order of GUI events for GUI tests [14], I/O operations for tests that involve network or disk operations, and more. These tests were called *non-order-dependent (NOD)* [20], but we will argue that the name is imprecise as such tests can still depend on the order.

Although some previous studies [1], [15], [24] show that NOD tests are more frequent than OD tests, several studies of flaky tests have focused on OD tests, proposing several tools specialized to detect [16], [17], [18], [19], [20], automatically fix [21], or tolerate [22] OD tests. Some example tools [16], [20] dynamically run the entire test suite, while intentionally randomizing the order of tests to increase the chance of detecting OD tests. Other tools [17], [18], [19] identify which individual tests or pairs of tests may be OD.

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

Our study is organized around four main research questions (RQs) that aim to improve our understanding of how to rerun, detect, debug, and prioritize flaky tests. By answering these questions, we aim to provide actionable guidelines and practical suggestions for developers and researchers.

To perform our study, we (re)run tests using the default test runner configured by the developers, following what developers typically do in their development practice. In other words, we do *not* use any research tools [16], [17], [18], [19], [20], [25] or emerging approaches [26], [27] that aim to detect flaky tests. While our procedure does miss some flaky tests, it (1) gives a more *realistic assessment* of the impact that flaky tests have on regression testing; (2) exposes flaky tests that can be particularly difficult to debug, because they fail rarely even under the exact same build configuration; and (3) avoids false alarms that can be produced by some of the research tools (e.g., iFixFlakies [21] reports on some false alarms reported by iDFlakies [20]).

We perform our empirical evaluation on open-source Java projects that use the Maven build system [28] and have JUnit tests. We let Maven Surefire plugin [29] run these JUnit tests (by executing `mvn test`), as configured by the project developers. Maven structures Java projects into modules, and each module has its own test suite. Our experiments find flaky tests in 26 modules from the iDFlakies dataset [30]. While the iDFlakies evaluation [20] aimed to run each test suite 100 times, we run each test suite exactly 4000 times, potentially in different *test-class orders (TCOs)* as permitted by Surefire. We choose 4000 runs to balance the chance to detect flaky tests and the machine time of our experiments. Any test that have both pass and fail outcomes are marked flaky. We obtain a dataset of 107 flaky tests, where a "test" is defined by JUnit—a specific test method in a test class. Moreover, we run each of those 107 flaky tests 4000 times in isolation, running just the one test method without the rest of its test class or the test suite. Our dataset is publicly available on our website [31] and we use this dataset to address our four RQs.

One important finding of our study affects the nomenclature of flaky tests. Namely, many flaky tests that were labeled "non-order-dependent" actually do depend on the order, e.g., across different TCOs, they have very different *failure rates*—the ratio of the number of failed runs over the number of total runs for a particular test order. We therefore introduce two subcategories of NOD tests—*non-deterministic, order-dependent (NDOD)* and *non-deterministic, order-independent (NDOI)*. To the best of our knowledge, we are the first to study the relationship of test orders on NOD tests.

The RQs we study also suggest the following guidelines:
RQ1 & RQ3: To check if test failures are due to flaky tests, one can use ≤ 5 reruns (50% less than Google's default ten reruns [6]) and still check correctly $\geq 82\%$ of the time.
RQ2 & RQ3: To detect flaky tests, one should run tests in a test suite instead of in isolation, and run tests in different orders with fewer times each, rather than fewer orders with more times each.
RQ3: To debug flaky tests, one should first run the tests in isolation, because tests tend to run faster and fail more often, before running the tests in a test suite.
RQ4: To prioritize the debugging of flaky tests, one can consider debugging tests that often fail together before the ones that do not fail together.

TABLE I
CATEGORIZATION OF TESTS. $\mathbb{F}$ ARE FAILURE RATES PER ORDER FOR A TEST. DIF IS TRUE IFF ANY RATE SIGNIFICANTLY DIFFERS FROM OTHERS. NOD AND OD ARE FROM IDFLAKIES' CATEGORIZATION [20].

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

II. CATEGORIES OF FLAKY TESTS & A REAL EXAMPLE

Our categorization of flaky tests builds on DTDetector [16] and iDFlakies [20], which automatically categorized all flaky tests into two categories based on the empirical results from running the tests in various test orders. We redefine the existing categories with tests' failure rates across test orders, refine an existing category by introducing two new subcategories, and show an example test from a new subcategory.

A. Categories

DTDetector and iDFlakies partition flaky tests into two categories. One category are *order-dependent (OD)* tests that can *deterministically* pass or fail based on the order in which the tests are run. We clarify that such tests are deterministic in that their failure rates are either 0% or 100% for each order, and they have at least two orders whose failure rates differ. For completeness of categorization, we also define tests that are *not flaky*—they either always pass (all orders have 0% failure rate) or always fail (all orders have 100% failure rate).

The other category are *non-order-dependent (NOD)* tests that are flaky but not OD. We clarify that such tests have at least one order where the test fails non-deterministically (failure rate is neither 0% nor 100%). We further break down these NOD tests to non-deterministic, order-dependent (NDOD) tests and non-deterministic, order-independent (NDOI) tests. Specifically, *NDOD* tests are NOD tests where at least one order's failure rate significantly differs from other orders' failure rates, e.g., a test that has a 99% failure rate in one order but 0% in another. A statistical test can be used to determine if any one order's rate significantly differs from the others. Conversely, *NDOI* tests are NOD tests where all failure rates do *not* significantly differ. Table I shows precise definitions.

Some previous studies of flaky tests [1], [14], [15], [24] categorized these tests into fine-grained categories, but the authors *manually* examined the source code of the tests or conducted surveys of developers after the tests were fixed and the causes of flakiness were removed. Moreover, a study [24] showed that manually identified fixes for flaky tests (and, hence, their corresponding categorizations) can be quite error-prone. We aim to *automatically* categorize flaky tests, even if in coarse-grained categories.

We further argue in this paper that it is important to differentiate between NDOD and NDOI tests, ideally automatically. In fact, we find that the majority of NOD tests are actually NDOD, not NDOI. Developers and tool builders should take test orders into account. For example, a flaky-test management system [24], [32], [33] can save and prioritize test order(s) that

```
1  @Test(timeout = 1500)
2  public void shouldRetryWithDynamicDelayDate() {
3    ... // test setup
4    atLeast(Duration.ofSeconds(1),() -> unit.get("/baz").
       dispatch(...).join());
5  }
```

Fig. 1. Example NDOD test that fails substantially less often when run as part of a test suite than when run in isolation.

developers should use for debugging based on the category of the test determined with the failure rates of the observed test orders. For NDOD and OD tests, developers could debug with the test orders whose failure rates differ the most, while for NDOI tests, any order could suffice.

B. Example NDOD Test

One example NDOD test from our study is shown in Figure 1. The test `shouldRetryWithDynamicDelayDate` is from a project [34] that implements client-side response routing. This test is flaky because some runs take longer than the timeout limit of 1500ms. We find that the test is NDOD because Line 4 launches a server, and in runs where the test passes, other tests running before this test trigger the Java just-in-time (JIT) compiler, thereby reducing the latency to start the server. On the other hand, in runs where the test fails, the JIT compiler is not triggered, thereby increasing the latency to start the server and resulting in the test exceeding 1500ms. We confirmed our understanding by logging the time it takes to run the test and then running the test multiple times. With our changes, we see that after ∼5 runs, the runtime of this test would decrease from ∼1500ms to ∼1000ms.

We refer to the number of times a flaky test fails consecutively as the test's *burst length*. In our 4000 TSRs, we find that this test's maximal burst length is 7 in one order (TCO), with the average maximal burst length being 3.1 across all TCOs. We also find that this test has a failure rate of 39.1% when run by itself in isolation, while its failure rate is 10.9% when it is run in its test suite, with the failure rate ranging from 0% to 20.1% depending on the TCO. This test's failure rate varies a lot because some other tests running before it can exercise code that triggers the JIT compiler, in which case the test often passes because it then takes closer to 1000ms.

III. Research Questions

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

RQ1: What is the failure rate and the maximal burst length of flaky tests across all test-suite runs?

Why it matters: A common way to separate test failures into regression failures and flaky failures is through test reruns. For example, Google [6] and Microsoft [24] report how they rerun tests after a failure to check whether the test's result would change to pass (indicating a flaky failure instead of a regression failure). The number of reruns is typically chosen ad-hoc, e.g., a round number of ten. This RQ aims to provide empirical evidence for how many reruns can suffice to separate regression failures and flaky failures.

RQ2: How do failure rates and maximal burst lengths differ across TCOs, and how many tests are NDOD and NDOI?

Why it matters: Developers have a limited budget of test runs to detect NOD tests. Is it better for those runs to be used through (1) many unique test orders with fewer runs each, or (2) fewer orders with many runs each? If NDOD tests are more prevalent than NDOI tests, then (1) can be better, otherwise (2) can be better. This RQ aims to provide empirical evidence on the prevalence of these two subcategories, so that developers can better detect NOD tests.

RQ3: How likely can NOD-test failures from running the test suite be reproduced by running the NOD test in isolation?

Why it matters: When developers encounter a test failure by running a test suite, a common next step is to try reproducing a test failure by running the test in isolation. For a regression failure, the test continues to deterministically fail in isolation. For OD flaky tests, the test also deterministically passes or fails in isolation [21], and it deterministically fails when run in the same order that produced the test failure. However, for NOD tests, it is unclear how likely test runs in isolation can reproduce the test failure from running the entire test suite, e.g., the test may fail more or less often when run in isolation than in the test suite. This RQ aims to show empirical evidence to clarify this issue.

RQ4: How do failure rates of individual flaky tests relate to failure rates of test-suite runs?

Why it matters: Whether developers can merge their recent changes to a project typically depends on whether the entire test suite passes or fails with their recent changes. RQ1 and RQ2 study the failure rate of each individual flaky test, while RQ4 aims to show how often a test-suite run has at least one failing flaky test. RQ4 is important because it provides empirical evidence on how often (1) developers encounter a test-suite run failure due to flaky tests and (2) failures of different flaky tests are symptoms of the same underlying problem, which developers can use to prioritize their debugging efforts.

IV. Experimental Methodology

A. Modules Used in Our Study

To investigate our research questions, we use open-source code from the iDFlakies dataset [30]. We obtained this dataset in our previous work with the iDFlakies tool, which detects flaky tests by perturbing the execution order of test methods (not just test classes) across repeated test-suite runs and marking tests that both pass and fail in various runs as flaky. Due to the perturbing of test order, many tests that iDFlakies detected are OD tests, i.e., they pass or fail deterministically depending on which other tests have (not) run before them in the test suite.

The iDFlakies dataset [30] consists of Java projects that use the Maven build system [28]. Maven organizes projects around modules, so we present our analysis in terms of modules. We do not use all modules (111) from the iDFlakies dataset for our study, because we focus on NOD tests, and iDFlakies detected NOD tests in only 62 modules (and only OD tests in the other 49 modules). We focus on NOD tests because

TABLE II
STATISTICS FOR THE MODULES WHERE WE DETECTED SOME FLAKY TEST BY FIRST RUNNING ENTIRE TEST SUITES (TSO) AND THEN RUNNING THE FLAKY TESTS DETECTED BY TSO IN ISOLATION (ISO); '=' INDICATES VALUE SAME AS IN THE CELL TO THE LEFT.

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

many past studies of flaky tests focused on OD tests [16], [19], [20], [21]. We find that only 48 of the 62 modules could be compiled "out-of-the-box" (i.e., Maven could not download some dependencies for these 14 modules), and for 44 of the 48 modules we could run the test suite 4000 times (i.e., the test suite deadlocks) and easily control the order in which Maven Surefire [29] runs test classes (i.e., the module uses Maven Surefire version 2.7 or higher).

B. Configurations in Our Study

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

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

We run TSO and ISO 4000 times each to balance the chance to detect flaky tests and the machine time used for our experiments. In total, our experiments used 2148 hours (~90 days) of CPU time. To the best of our knowledge, this is the largest number of runs in any published study of flaky tests. We ran our experiments on Microsoft Azure [35] using the `Standard_D11_v2` virtual machines with 2 CPUs, 14 GB of RAM, and 100 GB of hard-disk space each.

C. RQ1 and RQ2: Failure Rate and Burst Length

To answer RQ1 and RQ2, we obtain and compare failure rates (defined in Section I) and maximal burst lengths (introduced in Section II-B) of flaky tests across all TCOs (RQ1) and across different TCOs (RQ2). Of the 44 modules from Section IV-A, we find at least one flaky test in 26 modules when they are run in TSO. Table II shows the statistics of these 26 modules. Figure 2 shows an overview for how we run the test suites to collect our dataset.

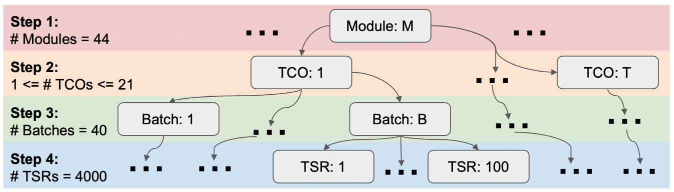

Fig. 2. Overview of running test suites to obtain flaky tests in our study.

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

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

We design our experiment to use multiple batches with 100 TSRs per batch rather than (1) just one batch to run all 4000 TSRs, or (2) 4000 batches, each with just one TSR. Compared to (1), our design helps with cases where a test may deadlock in a batch (i.e., we would only need to rerun one small batch). Compared to (2), our design helps by controlling machine cost and providing control over the number of TSRs for each TCO (which we use to calculate failure rates for RQ2).

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

2) Flaky Tests: In total, we detected 107 flaky tests in 26 modules. Compared to iDFlakies, which detected 124 flaky tests in these 26 modules, 64 tests are in common with our 107 tests, 60 tests are detected only by iDFlakies, and 43 tests are detected only by our runs. In 18 modules where we did not detect any flaky test, iDFlakies detected 40 NOD tests. We detect some more flaky tests than iDFlakies, because the iDFlakies study ran most test suites 100 times, while we run them 4000 times. We detect some fewer flaky tests than iDFlakies because of three reasons: (1) iDFlakies uses a custom Maven plugin that does not respect some configuration options from Surefire, e.g., `exclude` and `runOrder`; (2) iDFlakies does produce false alarms, e.g., it may even run test methods annotated with `@Ignore` [36], which should be skipped; and (3) iDFlakies can randomize not just the test-class order but also the test-method order. These features of iDFlakies have been designed to maximize the number of potentially flaky tests it can detect, while our study aims to more closely understand actual flaky tests in developers' typical test-suite runs.

D. RQ3: Reproducing TSO Failures in ISO

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

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

E. RQ4: Effect of Flaky Tests on TSRs

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

V. Results

We next present the results for our four research questions.

A. RQ1: Overall Failure Rate and Burst Length

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

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

If test failures are temporally correlated and tend to occur in consecutive reruns with large burst lengths, they appear more deterministic to developers and may mislead them in

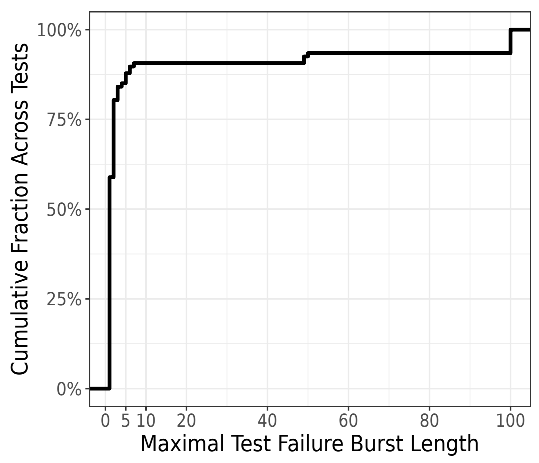

Fig. 3. Distribution of maximal burst lengths across 107 tests for TSO.

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

Figure 3 shows the cumulative distribution function (CDF) of *maximal* burst lengths we observed across all test reruns. The number cannot exceed 100, because each batch executed 100 TSRs. We discuss the maximal burst length, rather than the average, to obtain a *worst case estimate* of the negative impact of flaky tests. The CDF reaches a first plateau at 90.6% for a burst length of 7, which means that over 90% of flaky tests in our study failed at most 7 times. Even a burst length of 5 already has 87.8% of tests. The later increases are for two tests with a maximal burst length of 49, one of 50, and seven OD tests of 100. (Of these seven tests, four are in M1 and fail in all ISO runs, so strictly considering only their ISO failure rate of 100%, they would be marked as always failing and not flaky in ISO, even if they are flaky in TSO—we mark this with '*' in the table.) Thus, the commonly used number of ten reruns by Google [6] to detect flaky tests does not appear well justified: after one failure, 87.8% of tests can be found to pass in 5 reruns, with minor increases at 6 and 7 reruns. The remaining tests require many more than ten reruns.
Guideline: When rerunning tests after failures to check if they are flaky, our results suggest ≤ 5 reruns.

B. RQ2: Effect of Order on Failure Rate and Burst Length

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

We first consider *failing TCOs*, which have at least one failing run. For each test we count the number of failing TCOs and divide it by the total number of TCOs. If the ratio of failing TCOs is low, then the distribution of failures across TCOs is not uniform. Figure 4 shows a boxplot for the ratio of failing TCOs with a minimum of 4.8% (1 out of 21), maximum of 90.5% (19 out of 21), median of 19%, and interquartile range of 10% to 48%. From the skew in the distribution, TCOs do affect test failures often. The failures appear in a small ratio of the TCOs for a vast majority of tests. In fact, for 75% of tests, more than half of their TCOs have no failures, which would be unexpected if the failures were independent and uniformly distributed across TCOs. Our manual inspection (Section VI) finds some tests that are definitely OD, NDOD, and NDOI, and the ratio of failing TCOs for these categories is 4.8%–19%, 4.8%–58.3%, and 66.7%, respectively. A higher ratio indicates that a test may be NDOI.

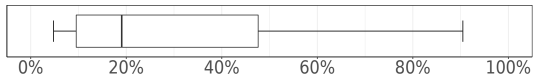

Fig. 4. Ratios of failing TCOs for 98 tests with more than 2 TCOs.

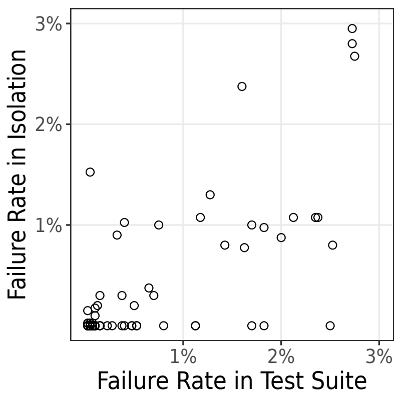

Fig. 5. Failure rates in TSO and ISO for 80 tests that TSO detected, with both rates $\leq 3\%$; details of the other tests are in Table III.

We also conduct a χ^2 test of independence to identify significant differences in failure rates across different TCOs. Out of 98 tests, 70 have a p-value lower than 0.05. For these tests, the null hypothesis that the failure rates per TCO are the same is rejected. Their failure rates significantly differ across TCOs, and they are likely NDOD. For the remaining 28 tests, the null hypothesis cannot be rejected: while there is no clear evidence that the tests are NDOD, it does not necessarily imply that the tests are NDOI, as we show in Section VI-B3.

We finally consider differences of the maximal burst length across TCOs for each test. We observe that 23 out of 107 tests fail in only one order. We consequently exclude these from our analysis of differences across failing TCOs. Of the remaining 84 tests, 52 have an identical maximal burst length across all TCOs, which is 1 (i.e., no consecutive failures) for 48 tests and 100 (i.e., all repetitions fail) for 4 tests. As we discuss in Section VI-A, the latter tests are confirmed to be OD tests by our manual inspection. Of the remaining 32 tests, the maximal burst length alternates between 1 and 2 for 21 tests and varies by small numbers (1–3, 1–4, 1–7, 2–6, 3–5, 3–6) for the remaining 11 tests.
Guideline: We find that a lot of tests may be NDOD; to detect them, it is better to run tests in more TCOs with fewer times each than in fewer TCOs with more times each.

TABLE III
27 TESTS WITH $> 3\%$ TSO OR ISO FAILURE RATE.

MID	Test name	Failure rate [%] TSO	Failure rate [%] ISO
M1	Issue1298.test_for_issue_1	5.0	*100.0
M1	Issue1298.test_for_issue	5.0	*100.0
M1	DefaultExtJSONParser_parseArray.test_7	2.5	*100.0
M1	DefaultExtJSONParser_parseArray.test_8	2.5	*100.0
M3	PortTelnetHandlerTest.testListAllPort	10.0	0.0
M3	PortTelnetHandlerTest.testListDetail	10.0	0.0
M6	TestWriter.shouldWriteToElasticPort	11.4	0.0
M9	TestLRMovingSimMain.testSim	3.0	3.7
M11	TestParallelMatcher.testParallelSlowLog	0.3	58.6
M11	TestPartitionMatcher.testParallelSlowLog	0.8	68.2
M13	TestGetFunction.test	0.7	4.2
M13	TestMemoryLimit.test_no_abuse	0.6	57.1
M20	ZKTest.testBulkClusterJoining	49.0	0.0
M20	ZKTest.testDiscoveryListener	49.0	0.0
M20	ZKTest.testMembershipJoinAndLeave	50.0	0.0
M23	Issue256Test.runReconnectBlocking...9	3.2	0.5
M23	Issue256Test.runReconnectScenario0	5.1	16.5
M23	Issue256Test.runReconnectScenario1	1.6	16.5
M23	Issue256Test.runReconnectScenario2	1.2	15.9
M23	Issue256Test.runReconnectScenario3	1.6	16.1
M23	Issue256Test.runReconnectScenario4	1.6	15.8
M23	Issue256Test.runReconnectScenario5	1.6	16.6
M23	Issue256Test.runReconnectScenario6	1.9	15.3
M23	Issue256Test.runReconnectScenario7	1.9	14.9
M23	Issue256Test.runReconnectScenario8	2.7	16.3
M23	Issue256Test.runReconnectScenario9	2.3	15.7
M26	RetryAfterDelay...Test.shouldRetry...Date	10.9	39.1

C. RQ3: Reproducing TSO Failures in ISO

After encountering a test failure from running a test suite (TSO), developers are likely to debug the test by running it in isolation (ISO), because ISO runs faster. However, it is unclear how many flaky tests can be reproduced in ISO. Moreover, even when both TSO and ISO detect a test, the failure rates in TSO and ISO can greatly differ, as shown in Figure 5 and Table III. The figure shows a scatterplot for the 80 tests where both failure rates are below 3%, and the table lists the actual failure rates for the other tests.

From Table II we see that ISO detected only 50 of the 107 tests that TSO detected, despite both having the same number of runs (4000). That is, ISO did not detect 57 tests by itself, but note that four tests from M1 (shown in Table III) do have failing runs in ISO—in fact, these are OD tests that failed for all ISO runs (marked with '*' in the table)—but because they have no passing runs, we do not consider them detected in ISO. The difference in the number of detected tests already shows that reproducing passing and failing runs in TSO and ISO can greatly differ. Failure rates can also differ: of the 50 tests that ISO detected, 19 tests have their failure rates lower for ISO, 3 have it equal, and 28 have it higher. For the tests where the ISO failure rate is lower, the maximum and median difference is 2.6pp and 0.6pp, respectively. For the tests where it is higher, the maximum and median difference is 67.4pp and 7.4pp, respectively. We analyze in detail some of the tests with large differences in Section VI.

Because the observed failure rates are based on a sample of runs, we also perform statistical tests to check whether the differences are statistically significant. We conduct a paired Wilcoxon signed-rank test and a Kolmogorov-Smirnov (KS) test on the failure rate distributions obtained in our experiments. We chose the Wilcoxon signed-rank test because (1) it is based on a pair-wise comparison of the failure rate for individual tests, rather than an overall statistic on the failure rate distribution; and (2) it intuitively captures how often either TSO or ISO yields a higher failure rate, weighted by the rank of the difference magnitude. By mapping the actual magnitude of the difference to a rank, the test statistic is robust to outliers in the differences of failure rates. On our dataset, the test results in a p-value of 0.041. Therefore, the difference between the observed failure rates for TSO and ISO test executions is significant at the 0.05 level. However, the Wilcoxon test's mapping to ranks loses information on the magnitude of difference, which is captured by the CDFs on which the KS test is based. For our dataset, the KS test results in a p-value of 2.9×10^{-12}, which strongly rejects the null hypothesis that the samples come from the same distribution.

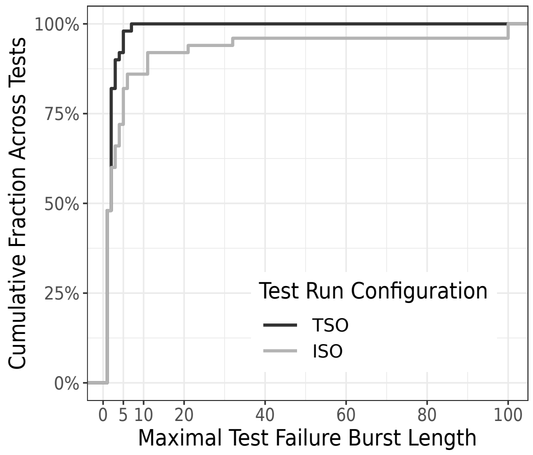

Fig. 6. Distribution of maximal burst lengths across 50 tests for TSO & ISO.

We also compare the maximal burst length for TSO and ISO runs. Figure 6 shows the CDFs of the maximal burst length per test for all tests detected by both TSO and ISO. From the plot, we observe that TSO reaches 82% for a maximal burst length of 2 test failures. However, for ISO the maximal burst length tends to be longer, e.g., ISO would reach 82% for a maximal burst length of 5 test failures instead. Due to the larger maximal burst length for ISO test executions, if one aims to find whether a test can pass, it could be beneficial to rerun failing tests in their test suites rather than in isolation, especially when the number of tests in the test suite is low. While rerunning in isolation has the advantage of reducing test runtime to an individual test, it entails the overhead for a potentially longer burst length before a passing run can be observed. In contrast, if one aims to get a failure, e.g., for debugging, it appears more beneficial to run the tests in ISO.

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

D. RQ4: Effect of Flaky Tests on TSRs

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

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

From Table II we can also see that 12 modules have only one flaky test detected, so their minimum, maximum, and sum failure rate is the same as the TSR failure rate. For the other 14 modules, we find that the TSR failure rate is the same as the maximum failure rate for 21% (3 / 14) of modules (M12, M16, M20). Thus, the flaky tests in these modules are related to one another (i.e., when one fails, more also fail). When we manually investigated M20's three flaky tests, we do indeed find that two tests always failed together, and the third test fails with them, but the third test fails in one more run and is responsible for M20's maximum failure rate. Specific details of these tests are in Section VI-B1. We further find that the TSR failure rate is the same as the sum failure rate for 43% (6 / 14) of modules. Thus, the flaky tests in these six modules are likely independent of one another (i.e., they always fail separately). For the remaining 36% (5 / 14) of modules, the TSR failure rate is in between the potential minimum and maximum TSR failure rates, suggesting that their flaky tests are a mix of related and independent flaky tests.
Guideline: At least 21% and up to 57% of test suites with multiple flaky tests have flaky tests related to one another, so flaky-test management systems [24], [32] could present the related flaky-test failures to help developers prioritize which tests to fix, e.g., first fix flaky tests that are related.

VI. Manually Inspected Flaky Tests

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

A. OD Tests

While this paper aims to study NOD tests, our experiments did encounter 7 tests categorized as OD, i.e., the failure rate for each TCO was either 0% or 100%. Our inspection confirmed that all these tests are indeed OD. 5 tests were already fixed [37], [38], [39] as part of our iFixFlakies work [21]; note that the iDFlakies dataset used in our experiments has older commits, not the latest `master`. Note also that the iDFlakies study [20] detected many more OD tests in the modules used in our experiments, because the iDFlakies tool perturbs the order of *all test methods* in a test suite, whereas our use of `mvn test` only perturbs the order of *some test classes*. The remaining 2 tests had not been fixed, so as a contribution of this work we provided a pull request [40] that the developers already accepted.

B. NDOD Tests

1) High TSO Failure Rate: Table III lists three tests that have a much higher failure rate for TSO than for ISO. All three tests are from the module M20 and class `ZKTest`. Our inspection shows that all three tests fail for the same reason. These tests check certain network operations, which require obtaining a port number. All three tests share the same port numbers, and when they use a port, they mark that by creating a file in the `/tmp` directory, which is never deleted. The test code allows these tests to use 152 different ports. As a result, after all three tests are run 50 times each, they mark 150 ports, and so at the 51st run two tests will pass and then one will fail, while from the 52nd run all three will start failing. In fact, these failures are deterministic. Strictly speaking, the tests are still NDOD because they both pass and fail for the exact same test order in TSO. In contrast, none of the tests fails in ISO. The reason is that we run each test 100 times on one virtual machine (VM) and then allocate a fresh VM for the next 100 runs. (Depending on the CI system, developers may also not encounter these failures, e.g., Travis [41] uses a fresh VM for every TSR.) Thus, each test marks only 100 ports in its 100 runs and does not reach the 152; had we run the test in ISO for 153 or more times on the same VM, we would have also encountered failures. These examples illustrate that running multiple tests can, in some circumstances, have a higher failure rate than running only each test in isolation.

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

Another test is from the module M13, class `TestMemoryLimit`. This test fails when some resource bound is exceeded. Specifically, the project provides a sandbox for executing JavaScript in Java, and this test checks that some execution of a JavaScript program does not exceed a certain amount of memory. The memory check does not consider the entire heap but only the amount of memory allocated by the thread that the test executes. When the test runs in isolation, it allocates all the memory and fails often, but not always, as the memory check is done every 50ms. Therefore, the test may pass the memory check at one point and then finish in less than 50ms, despite going over the memory limit after the check. When the test runs in the test suite, another test runs before it and allocates many shared objects. Thus, the second test can use these shared objects from the heap and allocates less, so it fails much less often.

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

3) High χ^2 p-Value: We finally discuss two example tests that have high p-values but our inspection still finds the failures to depend on the test order. One test is from the module M15, class `CronTest`. The test creates a pattern matcher for time which itself calls `DateUtil.date().second()` to initialize the matcher. The test also explicitly creates another time object calling `DateUtil.current(false)` to be matched with the matcher. Both calls get milliseconds and translate them into seconds, minutes, hours, and dates. The test fails if the two calls have a different value for seconds. The two calls are executed nearby, so the chance is small that the first call is executed right at the end of one second interval and the second call right at the start of the next second interval. The probability for the test to fail depends on how much time it takes between the two method calls. In our experiments, this time is $\sim$15ms for ISO runs, i.e., the test fails if the first call gets milliseconds that modulo 1000 give values 985–999, so the test fails in $\sim$15/1000=1.5% of runs. In contrast, the code runs faster in TSO (due to the already discussed effects of class loading and JIT compilation), so the test fails less frequently, only $\sim$1/1000=0.1% of runs. Moreover, the failure rates differ across the two test orders: in one order this class runs second, and the test never fails; in the other order the class runs first, so the test can fail but still less frequently than in ISO because other test methods run before this test.

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

4) Others: One test is already explained in Section II. Due to space limits, we omit descriptions of 5 tests: `TestWriter` from M6 is flaky because of timeout, whereas `CompletableThrowingSafeSubscriberTest`, `CompletableThrowingTest`, and `SingleThrowingTest` from M21 and `Issue621Test` from M23 are flaky because of concurrency.

C. NDOI Tests

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

The error message does not hint at the root cause but says that some averages of numeric values differ in two data structures. Our inspection shows that all of the tests populate these data structures with random numbers, and the random seed is based on the current time. The time is taken in milliseconds and translated into seconds, minutes, hours, and the date. A careful analysis of `testTimeSeriesDropOff` and `testMultipleTimeSeriesMovingAverage` shows that they fail when the time seed translates into the range of approximately 58min:20sec to 59min:55sec (for any hour or date); if a test is run earlier or later, it passes. (The reason is that each test initializes the two data structures based on the time using offsets of 5sec and 100sec.) Most precisely, in each hour there are 95000 millisecond values for which each test fails, so assuming that each test can be run uniformly for any millisecond, each test is expected to fail in 95000/(60*60*1000)=2.64% of runs. In our experiments, the tests indeed have similar failure rates: 2.95% in ISO and 2.72% in TSO for `testTimeSeriesDropOff`, and 2.80% in ISO and 2.72% in TSO for `testMultipleTimeSeriesMovingAverage`. The other two tests, `testManySparseTimeSeries` and `testAdditionalTimeSeries`, behave similarly.

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

D. NOD Tests Difficult to Classify

We inspected three NOD tests that are difficult to classify as NDOD or NDOI. We selected these tests based on high p-values (e.g., one test has p-value of 1), but some have a low number of failures (e.g., one test fails twice in 2000 TSRs of one TCO but does not fail in 100 TSRs for any of the other 20 TCOs). The root cause for all three tests is concurrency [42].

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

Another test is from module M14, class `WaiterTest`. This test creates one thread that executes `lock.wait(millis)`. The main thread has `Thread.sleep(20)` and then effectively calls `lock.notify()`. However, if `notify` is called before `wait`, the signal is missed, and `wait` would block forever if it were not for the timeout of `millis=1000`. We can make the test to fail deterministically by adding a delay in the right place in the code under test. We can also delete the existing `sleep(20)` in the test to make it fail determinstically. Unfortunately, reasoning precisely and analytically whether the probability that `notify` is missed due to TCO is again rather challenging because it requires determining the execution times of various events controlled by the JVM. As discussed in Section V-B, an empirical approach would be to just run the tests many more times to observe more failures and use a statistical analysis to check failure rates across TCOs.

VII. Threats to Validity

A threat to validity is that our study uses only 26 modules from 23 Maven-based, Java projects. These modules may not be representative, causing our results to not generalize well. We attempt to mitigate this threat by using modules from iDFlakies [30], selecting them as described in Section IV-A.

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

The findings from our RQs in Section V may be influenced by the types of statistical tests that we used to interpret the data. We attempt to mitigate this threat by considering two statistical tests for RQ3 (Section V-C). We also make all data and scripts that were used to generate the plots and figures in our paper publicly available on our website [31] so that others may interpret the data however they see fit.

VIII. Conclusion

Flaky tests are caused by various sources of non-determinism, and the research community can benefit from multiple studies to understand flaky tests and develop new solutions for them. Several studies of flaky test have keyed on one group of flaky tests, order-dependent tests. We show that the other group, called "non-order-dependent" tests, also has many tests that actually do depend on the test order, sometimes in complex ways. These tests have significantly different failure rates in different test orders and in isolated runs. To capture the complexity of these tests, we propose the term *non-deterministic, order-dependent (NDOD)* tests. We manually inspect a number of flaky tests to show concrete, real-world examples. We hope that our study motivates more researchers to tackle this practically important problem.

Acknowledgments

We thank Anjiang Wei for helping us debug some flaky tests, Jon Bell for extensive discussions about flaky tests, and Tianyin Xu for sharing Microsoft Azure credits. This work was partially supported by NSF grant nos. CCF-1763788 and OAC-1839010, GEM fellowship, and Supplemental Summer Block Grant (SSBG). We acknowledge support for research on flaky tests from Facebook and Google.

References

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

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

[3] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, “DeFlaker: Automatically detecting flaky tests,” in *ICSE*, 2018.

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

[5] Google, “Avoiding flakey tests,” 2008. [Online]. Available: http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html

[6] J. Micco, “The state of continuous integration testing at Google,” in *ICST*, 2017. [Online]. Available: https://bit.ly/2OohAip

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

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

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

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

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

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

[13] “Test Verification,” 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification

[14] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, “Making system user interactive tests repeatable: When and what should we control?” in *ICSE*, 2015.

[15] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, “Understanding flaky tests: The developer’s perspective,” in *ESEC/FSE*, 2019.

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

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

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

[19] A. Gambi, J. Bell, and A. Zeller, “Practical test dependency detection,” in *ICST*, 2018.

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

[21] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, “iFixFlakies: A framework for automatically fixing order-dependent flaky tests,” in *ESEC/FSE*, 2019.

[22] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, “Dependent-test-aware regression testing techniques,” in *ISSTA*, 2020.

[23] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic, “Detecting flaky tests in probabilistic and machine learning applications,” in *ISSTA*, 2020.

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

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

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

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

[28] “Maven,” 2020. [Online]. Available: https://maven.apache.org

[29] “Maven Surefire plugin,” 2020. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin

[30] “iDFlakies: Flaky test dataset,” 2020. [Online]. Available: https://sites.google.com/view/flakytestdataset

[31] “Flaky test statistics,” 2020. [Online]. Available: https://sites.google.com/view/flakyteststatistics

[32] “Manage flaky tests,” 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/pipelines/test/flaky-test-management

[33] “Flaky tests,” 2020. [Online]. Available: https://docs.gitlab.com/ee/development/testing_guide/flaky_tests.html

[34] “Riptide,” 2020. [Online]. Available: https://github.com/zalando/riptide

[35] “Microsoft Azure,” 2020. [Online]. Available: https://azure.microsoft.com

[36] “JUnit Ignore annotation,” 2020. [Online]. Available: http://junit.sourceforge.net/javadoc/org/junit/Ignore.html

[37] “Pull request #2148: Fixing flaky tests in DateTest4_indian and DateTest5_iso8601,” 2020. [Online]. Available: https://github.com/alibaba/fastjson/pull/2148

[38] “Pull request #2906: Fixing flaky tests in PortTelnetHandlerTest,” 2020. [Online]. Available: https://github.com/apache/incubator-dubbo/pull/2906

[39] “Pull request #592: Fixing flaky test ShutdownListenerManagerTest.assertIsShutdownAlready,” 2020. [Online]. Available: https://github.com/elasticjob/elastic-job-lite/pull/592

[40] “Pull request #3291: Fixing flaky tests in DefaultExtJSONParser_parseArray,” 2020. [Online]. Available: https://github.com/alibaba/fastjson/pull/3291

[41] “Travis CI - Test and deploy with confidence,” 2020. [Online]. Available: https://travis-ci.org/

[42] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in *ASPLOS*, 2008.