A Survey of Flaky Tests

OWAIN PARRY, University of Sheffield, UK
GREGORY M. KAPFHAMMER, Allegheny College, USA
MICHAEL HILTON, Carnegie Mellon University, USA
PHIL MCMINN, University of Sheffield, UK

Tests that fail inconsistently, without changes to the code under test, are described as *flaky*. Flaky tests do not give a clear indication of the presence of software bugs and thus limit the reliability of the test suites that contain them. A recent survey of software developers found that 59% claimed to deal with flaky tests on a monthly, weekly, or daily basis. As well as being detrimental to developers, flaky tests have also been shown to limit the applicability of useful techniques in software testing research. In general, one can think of flaky tests as being a threat to the validity of any methodology that assumes the outcome of a test only depends on the source code it covers. In this article, we systematically survey the body of literature relevant to flaky test research, amounting to 76 papers. We split our analysis into four parts: addressing the causes of flaky tests, their costs and consequences, detection strategies, and approaches for their mitigation and repair. Our findings and their implications have consequences for how the software-testing community deals with test flakiness, pertinent to practitioners and of interest to those wanting to familiarize themselves with the research area.

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

Additional Key Words and Phrases: Flaky tests, software testing

ACM Reference format:
Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1, Article 17 (October 2021), 74 pages.
https://doi.org/10.1145/3476105

1 INTRODUCTION

Software developers rely on tests to identify bugs in their code and to provide a signal as to their code's correctness [109]. However, should such signals have a history of unreliability and ambiguity, they not only become less informative but may also be considered untrustworthy [84, 166]. In the context of software testing, these unreliable signals are referred to as *flaky tests*, such as

This work is supported by a Facebook Testing and Verification award (2019). Owain Parry is funded by the EPSRC Doctoral Training Partnership with the University of Sheffield (Grant No. EP/R513313/1). Phil McMinn is supported, in part, by EPSRC Grant No. EP/T015764/1.
Authors' addresses: O. Parry and P. McMinn, Dept. of Computer Science, The University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP; emails: oparry1@sheffield.ac.uk, p.mcminn@sheffield.ac.uk; G. M. Kapfhammer, Department of Computer Science, Allegheny College, Box Q, 520 North Main Street, Meadville, Pennsylvania, USA 16335; email:gkapfham@allegheny.edu; M. Hilton, Institute for Software Research, Carnegie Mellon University, TCS Hall 430, 4665 Forbes Avenue, Pittsburgh, PA 15213; email; mhilton@cmu.edu.

1049-331X/2021/10-ART17 $15.00
https://doi.org/10.1145/3476105

```
1       def test_spinup_time(hook):
2           """Tests to ensure that virtual workers intialized with 10000 data points
3           load in under 1 seconds. This is needed to ensure that virtual workers
4           spun up inside web frameworks are created quickly enough to not cause timeout errors"""
5           data = []
6           for i in range(10000):
7               data.append(torch.Tensor(5, 5).random_(100))
8           start_time = time()
9           dummy = sy.VirtualWorker(hook, id="dummy", data=data)
10          end_time = time()
11          assert (end_time - start_time) < 1
```

Fig. 1. A flaky test from the PySyft project [57]. This test contains an assertion on line 11 (highlighted) regarding the length of time taken to complete an operation. Because the time taken may vary depending on the machine specification, it can fail regardless of any genuine bugs that may or may not be present in the source code. In fact, this test was so flaky that the developers decided to remove it entirely [56].

the test case listed in Figure 1. The exact definition of what constitutes a flaky test varies slightly from source to source but is generally considered to mean a test that can be observed to both pass and fail without changes to the code under test. Some seem to take a fairly conservative view, only considering a test flaky if the flakiness stems from timing or concurrency issues [171]. Most sources simply call a test flaky if it can be observed to produce an inconsistent outcome when only the test code and the code under test that it exercises remains constant [133]. This particular definition is inclusive of tests that are flaky due to factors external to the test, such as the prior execution of other tests, or the execution platform itself. The most common type of flaky test related to the former case are known as *order-dependent tests*, i.e., one whose outcome depends upon the test execution order, but is otherwise deterministic [185]. Recently, order-dependent tests, and the tests on which they depend, have been categorized by their manifestation [159], and their definition has been generalized by one study's authors, who considered how non-deterministic tests can also be order-dependent [124]. While one may initially consider it unlikely that the test order would change between test suite runs, several widely studied testing techniques do just that, making order-dependent tests an obstacle to their applicability and thus a genuine problem rather than just a peculiarity [68, 73, 123]. As for the latter situation, a test with a platform dependency may be considered flaky, since it might pass on one machine but fail on another. While some may consider such a test to be a deterministic failure rather than a form of flakiness, researchers have commented that when using a cloud-based continuous integration service, where different test runs may be executed on different physical machines in a manner that appears non-deterministic to the user, the test outcome may effectively become non-deterministic as well [79]. One reason why a test may be dependent on a particular platform is because it relies on the behavior of a particular implementation of an underdetermined specification, i.e., a specification that leaves certain aspects of a program's behavior undefined [96]. For example, a test that expects a particular iteration order for an object of an unordered collection type—with no particular iteration order mandated by its specification—may pass on one platform where the respective implementation happens to meet its expectations but fail on another [97, 158]. At least one source describes a test as flaky if its *coverage* (i.e., the set of program elements that it exercises) is also inconsistent [157]. Generalizing further, Strandberg et al. [165] defined *intermittent tests* as those with a history of passing and failing, regardless of software or hardware changes. These authors considered flaky tests to be a special case of intermittent tests that pass and fail in the absence of changes, as per the traditional definition.

Flaky tests challenge the assumption that a test failure implies a bug, constituting a leading cause of false alarm test failures [171, 185], posing problems for both developers and researchers. For developers, flaky tests may erode their trust in test suites and lead to time wasted debugging

spurious failures [84]. Furthermore, developers may be tempted to ignore flaky test failures, which has been demonstrated to have potentially detrimental effects on software stability [154]. A recent survey of industrial developers demonstrated the prevalence and prominence of flaky tests [79]. The results indicated that test flakiness was a frequently encountered problem, with 20% of respondents claiming to experience it monthly, 24% encountering it on a weekly basis and 15% dealing with it daily. In terms of severity, of the 91% of developers who claimed to deal with flaky tests at least a few times a year, 56% described them as a moderate problem and 23% thought that they were a serious problem. For researchers, flaky tests are a threat to the validity of any methodology assuming that the outcomes of tests are dependent purely upon the code they cover. Examples of such strategies and techniques negatively impacted include fault localization [75, 172], mutation testing [102, 157], automatic test suite generation [64, 149, 156], and test suite acceleration methods that retrieve useful information from test runs in less time [73, 88, 123, 126, 185]. These problems provide motivation for research into flaky tests and for this survey. Furthermore, this research area is of rapidly growing interest. We found that 63% of all examined papers on this topic were published between 2019 and 2021. In 2020, Zolfaghari et al. [189] provided a review on some of the main techniques with regards to the detection, repair, and root causes of flaky tests. However, to date there has been no comprehensive survey published that covers not only this core literature but also the wider periphery, such as the influence that flaky tests have in the wider software engineering field (see Section 4 for additional details). It is thus our position that doing so at this time of increasing attention will be the most valuable, particularly to those researchers and developers wishing to familiarize themselves with the field of flaky test research.

In this survey, we examine the peer-reviewed literature directly concerning, or within the periphery of, flaky tests, according to the definitions previously described in this section. We make available a BibTeX bibliography of the examined literature in a public GitHub repository [59]. Overall, our survey makes the following contributions:

(1) An evaluation of the literature of the growing research area of flaky tests, the first survey to do so.
(2) A comparative analysis on the causes of flaky tests.
(3) A summarization of the costs and impacts of flaky tests upon the reliability and efficiency of testing.
(4) A comparison of the available tools and strategies for automatically detecting flaky tests.
(5) An analysis of existing techniques to mitigate and repair flaky tests.
(6) An identification of research threads, trends and future directions in the field of test flakiness.

In Section 2, we present our research questions, describe the scope of our survey and explain our methodology for collecting relevant studies. In Section 3, we discuss the underlying mechanisms and factors identified as the causes of flakiness, both in general and in more application-specific domains, and present a list of flakiness categories that emerge from the examined literature. In Section 4, we consider the negative impacts that flaky tests impose upon the reliability and efficiency of testing, as well as upon a range of specific testing-related activities. In Sections 5 and 6, we consider approaches for detecting, mitigating and repairing flaky tests, including insights and general techniques from the literature as well as a tour of the automated tools available. In Section 7, we take a high-level view of all the sources we consulted and examine their demographics and emergent research trends before summarizing in Section 8.

2 METHODOLOGY

In this section, we present the four research questions that this article aims to address through an analysis of relevant studies. To that end, we go on to describe the scope of our survey and give

our inclusion and exclusion criteria. Following this, we describe our full paper collection approach, including our search query, and present the results. Our four research questions, with their relevant sections, are as follows:

- **RQ1: What are the causes and associated factors of flaky tests?** Answered in Section 3, this question explores the mechanisms behind test flakiness. Our answer explains the common patterns and categories, as identified by previous research, and then goes on to explore the more domain-specific factors.
- **RQ2: What are the costs and consequences associated with flaky tests?** Answered in Section 4, this question addresses the costs incurred by flaky tests and their severity as a problem for both developers and researchers.
- **RQ3: What insights and techniques can be applied to detect flaky tests?** Answered in Section 5, this question catalogues the techniques that have been applied to identify flaky tests, many of which have been implemented as automated tools and scientifically evaluated with real-world test suites.
- **RQ4: What insights and techniques can be applied to mitigate or repair flaky tests?** Answered in Section 6, this question examines the range of literature presenting attempts at limiting the negative impacts of flaky tests, as we identified by answering **RQ2**, or repairing them outright.

2.1 Survey Scope

The scope of our survey is limited to peer-reviewed publications relevant to test flakiness as defined in Section 1, including those regarding order- and implementation-dependent tests. We included papers that have flakiness as their main topic or are tangentially related, for example, those describing a cause of flaky tests or presenting a method that is impacted by them. Some other testing topics share similar terminology to test flakiness but are not directly related, and thus we took care to exclude them from our survey. For example, when discussing order-dependent tests, we did not include works regarding redundant tests, which are sometimes described as *dependent tests* [115, 167, 168, 173]. In that context, the term is used to refer to tests whose outcome can be inferred from the outcomes of one or more other tests, as opposed to tests having nondeterministic outcomes with respect to the test run order. Furthermore, by only considering peer-reviewed sources, we excluded preprints, theses, blog posts, and other "grey" literature. We also took care to exclude position papers if their proposed work has been completed and published (as in the case of Bell et al. [66, 68]) and cases where the same (or a similar) study is presented in multiple publications (such as Luo et al. [132, 133]). We list our inclusion and exclusion criteria in Table 1. We consider a study relevant to our survey if it meets at least one of our inclusion criteria and none of our exclusion criteria.

2.2 Collection Approach

To collect the papers for our survey, we conducted query searches on the 15th of April 2021 using the ACM Digital Library [1], IEEE Xplore [16], Scopus [60], and SpringerLink [33]. We selected query terms that would cover both the general test flakiness and the order-dependent test literature. Our full search query was *("flaky test" OR "test flakiness" OR "intermittent test" OR "order dependent test" OR "test order dependency")*. While the subject area of the ACM Digital Library and IEEE Xplore is limited to computing/technology, Scopus and SpringerLink are far more general search engines including medicine, the social sciences, and other unrelated areas. For these two search engines, we made sure to restrict the results to computer science publications only. When evaluating papers against our inclusion and exclusion criteria, we read their titles, abstracts, and

Table 1. The Inclusion and Exclusion Criteria for Papers to Be Considered in This Survey

Inclusion	Exclusion
Discusses the causes of flaky tests	Not a peer-reviewed publication
Discusses the factors associated with flaky tests	Presents the same study as another included paper
Presents a strategy for detecting or analyzing flaky tests	Discusses redundant tests using the term dependent tests
Presents a strategy for repairing or mitigating flaky tests	A position paper, if the proposed work has been published
Describes an impact of flaky tests upon some technique or concept	

Table 2. Top Five Most Common Publication Venues

Venue	Papers
Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering	14
International Conference on Software Engineering	13
International Symposium on Software Testing and Analysis	9
International Conference on Automated Software Engineering	8
International Conference on Software Testing, Verification and Validation	5

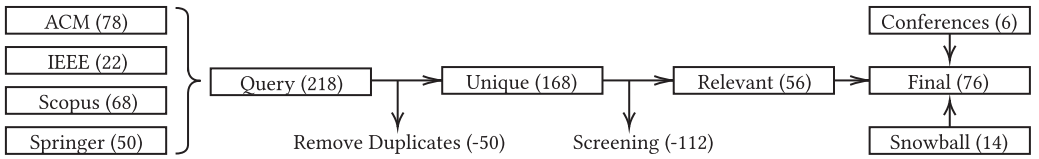


Fig. 2. An illustration of our paper collection approach, with the number of papers given at each stage.

introductions, and we evaluated their further sections if necessary. This constituted our screening process for selecting relevant papers. It is our position that the peer-review process constitutes a sufficient quality bar. For this reason, we did not include an additional quality assessment stage in our methodology. To collect more publications beyond our query searches, we checked the accepted papers of relevant 2021 conferences (such as those in Table 2) that had not yet published their proceedings, and thus would not have been found as part of our query searches. In addition, we applied a technique known as *backward snowballing* [177]. This involved reading the related work sections and references of each of the collected papers, from which we extracted additional studies, subject to our inclusion and exclusion criteria.

2.3 Collection Results

Figure 2 illustrates our paper collection process and gives the number of papers at each stage. Following our query searches across each of our four selected search engines, we ended up with a total of 168 unique results, that is, after removing duplicates. After our screening process, with respect to our inclusion and exclusion criteria as previously explained, we selected 56 papers. A further 6 relevant papers were identified from the 2021 accepted papers of the *International Conference on Software Engineering* (ICSE) [17], the *International Conference on Software Testing, Verification and Validation* (ICST) [40], and the *International Conference on Mining Software Repositories* (MSR) [39]. Finally, an additional 14 papers were collected via backward snowballing, making for a total of 76 relevant papers.

As shown in Figure 3, interest in flaky tests from a research point of view appears to be increasing over time, as evidenced from the growing number of collected papers published through the years. Until 2014, research in the area was fairly limited, at which point several now heavily cited papers emerged. Namely, these were Luo et al.'s *An Empirical Study of Flaky Tests*, that introduced a widely used set of flakiness categories [133] (see Table 4), and Zhang et al.'s *Empirically Revisiting the Test Independence Assumption*, which presented their tool DTDetector for detecting

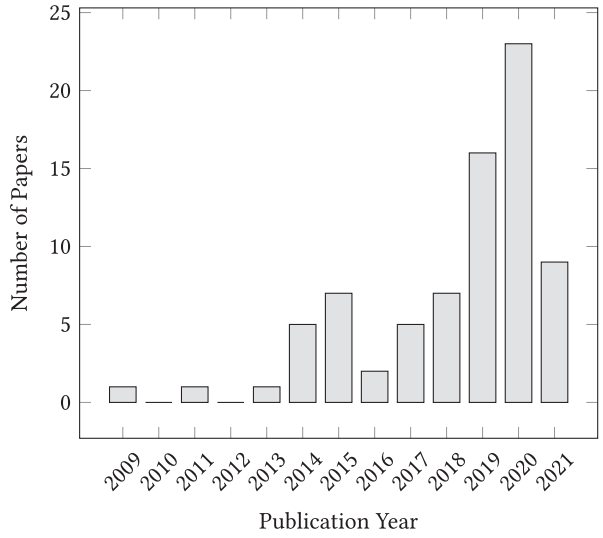


Fig. 3. Number of papers by publication year. This survey was conducted part-way through 2021, hence why the bar for 2021 is unexpectedly short.

- 2009 First reference to flaky tests in the research literature [119].
- 2011 First attempt at detecting order-dependent tests [143].
- 2014 First empirical evaluation of flaky tests. Ten categories of flakiness defined [133].
  First empirical evaluation of order-dependent flaky tests. First tool for their detection, DTDetector [185].
  First tool for mitigating order-dependent flaky tests, VmVm [69].
- 2015 First tool for explicitly detecting state-polluting tests, PolDet [98].
- 2016 First tool for detecting implementation-dependent flaky tests, NonDex [97, 158].
- 2017 First empirical studies of flaky tests in the context of continuous integration [104, 118, 138].
- 2018 First tool for detecting flaky tests based on differential coverage, DeFlaker [69].
- 2019 First tool for detecting both order-dependent and non order-dependent flaky tests, iDFlakies [122].
  First tool for automatically fixing order-dependent flaky tests, iFixFlakies [159].
  First tool to apply natural language processing techniques for the static prediction of flaky tests, FLAST [70].
- 2020 First tool for detecting flaky tests in machine learning applications, FLASH [78].
- 2021 First large-scale empirical study of general flaky tests in Python [95].

Fig. 4. Timeline of research milestones in the field of flaky tests.

order-dependent tests [185] (see Section 5.2.3). We consider these studies to be important research milestones, as shown in Figure 4. The drop between 2015 and 2016 appears to be something of an anomaly, and may be due to the fact that this data is produced from the sample of studies examined in this article (i.e., those passing our inclusion criteria in Section 2), as opposed to all of those published in the research literature. Overall, the trend indicates that flaky tests are an area of increasing interest, with just under 63% of all sampled studies published between 2019 and 2021.

Table 2 shows the top five conferences in which our collected papers were published. The *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE) [51] is first, followed closely by ICSE. For the field of computer science, both are very mature conferences, with the former dating back to the year 1987 and the latter to 1975. Three of the top five conferences are in the field of software engineering, of which software testing is a subfield. This result suggests that flaky tests are prominent enough as a topic to be

relevant in the more general field of software engineering, as opposed to being just a peculiarity of software testing. Together, these top five conferences account for approximately 64% of the studies examined in this article. Thus, we recommend that interested researchers regularly check the proceedings of these conferences for new publications about flaky tests.

3 CAUSES AND ASSOCIATED FACTORS

This section examines the sources that aim to understand the underlying causes and mechanisms that lead to flakiness within test suites. Some of these studies, like Luo et al. [133], have identified common patterns between flaky tests and have categorized them by their specific causes. Other studies examined more general factors, such as Shi et al. [158], that identified incorrect assumptions regarding library specifications, like the iteration order of unordered collection types, as a potential avenue for flakiness. Specifically, this section answers **RQ1**, which we achieve by splitting it into three sub-research questions, each with their own associated subsections. To that end, we consult a variety of relevant papers, as we identified according to the methodology described in Section 2. The answers to each sub-research question are summarized at the end of their respective subsections, and their combined findings and implications, constituting the overall answer to **RQ1**, are summarized in Table 3. Our three sub-research questions are as follows:

- **RQ1.1: What are the general causes of flaky tests?** Answered in Section 3.1, this question gives a summary of the breakdown of the different general causes of test flakiness as categorized by various studies. To that end, we examined four studies that analysed flaky tests in projects of no specific type or application with the aim of establishing different categories of flakiness and identifying their respective frequencies.
- **RQ1.2: What are the causes of order-dependent tests?** Answered in Section 3.2, and motivated by their particularly negative impacts (as examined further in Section 4.2), this question specifically investigates, in the context of test flakiness, the causes and factors associated with test order dependencies.
- **RQ1.3: What are the application-specific causes of flaky tests?** Answered in Section 3.3, this question addresses the factors specific to particular types of applications or testing strategies that are associated with flakiness that would not necessarily generalize to all domains. For this research question, we examine studies that focused on certain types of software and identified the causes of flaky tests specific to them.

3.1 General Causes

Studies have examined and categorized the causes of flaky tests in general software projects, that is, not constrained to a particular platform or purpose. To that end, objects of study such as historical commit data [133], bug reports [171], developers' insights from flaky tests that they've previously repaired [79], pull requests [121] and execution traces [95] have been analysed, resulting in a commonly used set of flakiness categories. Two of these studies consider subjects from the same source, namely, the Apache Software Foundation [3], yet interestingly present different findings [133, 171]. The union of the categories used in each of these sources is presented and described in Table 4. Inspired by previous work [175], we also present each category as a member of one of three families to illustrate their relatedness. The *intra-test* family describes flakiness wholly internal to the test, in other words, stemming from issues isolated to the execution of the test code itself, or the direct code under test. The *inter-test* family contains tests that are flaky with respect to the execution of other tests, for example, those with test order dependencies. The flakiness of tests in the *external* family stems from factors outside of the test's control, such as the time taken to receive a response

Table 3. Summary of the Findings and Implications Answering RQ1: What Are the Causes and Associated Factors of Flaky Tests?

	Finding	Implications	Source
🎓	Issues regarding **asynchronicity and concurrency** appear to be the leading causes of flaky tests.	Techniques for mitigating or repairing flaky tests ought to be able to address **asynchronicity and concurrency** to be the most impactful.	[79, 121, 133, 171]
</>	**Platform dependencies** represented 34% of sampled bug reports indicative of flaky tests.	Developers should ensure that test outcomes are **consistent across the platforms** targeted by their software, especially when using cloud-based continuous integration.	[79, 171]
🎓	**Order-dependent tests** were found to constitute up to 16% of flaky test bug reports and 9% of previous flaky test repairs.	Test order dependency appears to be a **prominent category** with specific negative impacts (see Section 4.2) and thus deserving of specific attention.	[79, 133, 171]
🎓 </>	**The majority of order-dependent tests are victims**, passing in isolation but failing after the execution of certain polluter tests.	Techniques to eliminate state-pollution during the execution of tests would **indirectly repair most order-dependent tests**. Developers ought to be especially careful that they do not introduce test cases that may induce a failure in other tests in the test suite.	[95, 159]
🎓	The dependency of 76% of order-dependent tests was related to only **one other test**.	Techniques for detecting order-dependent tests should consider that **complex networks of dependencies are an unlikely occurrence**.	[185]
🎓	Shared access to **in-memory resources via static fields** was the facilitator of 61% of order-dependent tests in the Java programming language.	Approaches for dealing with test order dependency should consider that a significant minority of cases cannot be identified by **analyzing program state alone**.	[185]
🎓 </>	Insufficient waiting for elements to be rendered in **user interface testing** appears to be a strong factor associated with flakiness in this domain.	Due to their similarities as **timing-based dependencies**, insights into mitigating asynchronous wait flakiness could be useful in this context.	[90, 153, 155]
🎓 </>	Algorithmic non-determinism accounts for 60% of flaky tests in **machine learning projects**.	Strategies for dealing with flakiness applied to more general projects, where asynchronous waiting and concurrency are the leading causes of flakiness, **may not be so applicable** to machine learning projects.	[78]
🎓 </>	The distribution of causes of flaky tests within **Android applications** appears to roughly correspond to that of more general projects created in languages like Java.	Insights gained from studying flaky tests in general are **likely to be relevant** to Android projects.	[170]

Findings relevant to researchers are marked with 🎓. Findings relevant to developers are marked with </>.

from a web sever. Table 5 presents the prevalence of these categories as determined by each source, along with their type of subject.

3.1.1 Commits. As part of an empirical evaluation, Luo et al. [133] sampled historical commit data from 51 projects of varying size and language (mostly Java) from the Apache Software

Table 4. Categories and Descriptions as Used to Classify Flaky Tests in the Various Studies Given in the Source Column

Family	Category	Description	Source
Intra-test	Concurrency	Test that invokes multiple threads interacting in an unsafe or unanticipated manner. Flakiness is caused by, for example, race conditions resulting from implicit assumptions about the ordering of execution, leading to deadlocks in certain test runs.	[79, 95, 121, 133, 171]
	Randomness	Test uses the result of a random data generator. If the test does not account for all possible cases, then the test may fail intermittently, e.g., only when the result of a random number generator is zero.	[79, 95, 121, 133]
	Floating Point	Test uses the result of a floating point operation. Floating point operations can suffer from a variety of discrepancies and inaccuracies such as precision over and under flows, non-associative addition, etc., which if not properly accounted for can result in inconsistent test outcomes, e.g., by comparing the result of a floating point operation to an exact real value in an assertion.	[79, 121, 133]
	Unordered Collection	Test assumes a particular iteration order for an unordered collection type object. Since no particular order is specified for such objects, tests that assume they will iterate in some fixed order will likely be flaky due to a variety of reasons, e.g., implementation of the collection class.	[95, 121, 133]
	Too Restrictive Range	Test where some of the valid output range falls outside of what is accepted in its assertions. This test is flaky, since it does not account for corner cases and thus it may intermittently fail when they arise.	[79, 95]
	Test Case Timeout	Test specified with an upper limit on their execution time. Since it's usually not possible to precisely estimate how long a test will take to run, by specifying an upper time limit a developer may run the risk of creating a flaky test, since it could fail on certain executions that are slower than anticipated.	[79, 95]
Inter-test	Test Order Dependency	Test that depends on some shared value or resource that is modified by another test that impacts its outcome. In the case where the test run order is changed, these flaky tests may produce inconsistent outcomes, since the dependencies upon tests previously executed beforehand are broken.	[79, 121, 133, 171]
	Resource Leak	Test that improperly handles some external resource, e.g., failing to release allocated memory. Improperly handled resources may cause flakiness in subsequently executed test cases that attempt to reuse that resource.	[79, 95, 121, 133, 171]
	Test Suite Timeout	Test is part of a test suite with a limited execution time. Intermittently fails, because it happens to be running once the test suite hits an upper time limit.	[79]
External	Asynchronous Wait	Test makes an asynchronous call and does not explicitly wait for it to finish before evaluating assertions, typically using a fixed time delay instead. Results may be inconsistent in executions where the asynchronous call takes longer than the specified time to finish, leading to the flakiness.	[79, 95, 121, 133, 171]
	I/O	Test that is flaky due to its handling of input and output operations. For example, a test that fails when a disk has no free space or becomes full during file writing.	[79, 95, 121, 133]
	Network	Test that depends on the availability of a network connection, e.g., by querying a web server. In the case where the network is unavailable or the required resource is too busy, the test may become flaky.	[79, 95, 121, 133]
	Time	Test relies on local system time and it may be flaky due to discrepancies in precision and timezone, e.g., failing when midnight changes in the UTC timezone.	[79, 95, 121, 133]
	Platform Dependency	Test depends on some particular functionality of a specific operating system, library version, hardware vendor, etc.. While such tests may produce a consistent outcome on a given platform, they are still considered flaky, particularly with the rise of cloud-based continuous integration services, where different test runs may be executed upon different physical machines in a manner that appears non-deterministic to the user.	[79, 95, 171]

Table 5. The Breakdown of the Causes of Flaky Tests as Categorized by Various Studies

Source	Subjects	Concurrency	Randomness	Floating Point	Unordered Coll.	Too Res. Range	Case Timeout	Test Order Dep.	Resource Leak	Suite Timeout	Async. Wait	I/O	Network	Time	Platform Dep.
[133]	Commits	16%	2%	2%	1%	—	—	9%	5%	—	37%	2%	5%	3%	—
[171]	Bug reports	19%	—	—	—	—	—	16%	8%	—	18%	—	—	—	34%
[79]	Fixed flaky tests	26%	1%	3%	0%	17%	8%	9%	6%	2%	22%	0%	0%	2%	4%
[121]	Pull requests	8%	5%	2%	0%	—	—	0%	5%	—	78%	5%	14%	4%	—
[95]	Execution traces	3%	37%	—	1%	0%	1%	—	2%	—	3%	7%	42%	4%	0%

Dashes indicate that the study did not consider that category. Percentages are rounded and do not sum to 100%, since not all flaky tests could be categorized by the authors of the referenced source. See Table 4 for a description of each category.

Foundation. They identified 201 individual commits that were evidence of a flaky test being repaired by a developer and studied these to determine the most common causes of test flakiness. Specifically, they categorized the type of flakiness that each commit repaired under 11 causes, including a miscellaneous or "hard to classify" category. They categorized 37% of commits under the *asynchronous wait* category, meaning a developer repaired flaky tests that were caused by not properly waiting upon asynchronous calls. Such tests typically employed a fixed time delay following the invocation of some asynchronous operation instead of explicitly waiting for it to complete before evaluating assertions, potentially leading to an erroneous outcome in the executions where the time delay was insufficient. Examples include waiting for a response from a web server or waiting for a thread to finish. Figure 5 shows a concrete example of a flaky test of the *asynchronous wait* category taken from the Home Assistant project [12]. They categorized 16% of their commits under the *concurrency* category, pertaining to flaky tests whose non-determinism was due to undesirable interactions between multiple threads, including issues such as data races and deadlocks. The main difference between this category and *asynchronous wait* is that the latter refers to synchronization issues explicitly concerning external or remote resources. They categorized a further 9% under the *test order dependency* category, in other words, a developer fixed order-dependent tests, and 5% under *resource leak*. They distributed the remainder among the other eight categories, which included causes related to networking, time, floating point operations, assumptions regarding the iteration order of unordered collection types [158] and input/output operations. The authors suggested that techniques for detecting and fixing flaky tests ought to focus on those related to asynchronicity, concurrency and test order dependency, the three most commonly observed categories.

3.1.2 Bug Reports. Vahabzadeh et al. [171] categorized the root causes of test bugs, that is, bugs in the code of tests as opposed to the code under test, which they mined from the bug repository of the Apache Software Foundation. In total, they systematically categorized 443 test bugs. They considered five categories: *semantic bugs*, *environment*, *resource handling*, *flaky tests*, and *obsolete tests*. Based upon their descriptions, we would consider the *environment* and *resource* categories, along with *flaky tests* of course, to fall under the more inclusive definition of flakiness given in Section 1. From this perspective, they categorized 51% of test bugs to be flaky tests, 34% of which under the *environment* category, which was sub-categorized into tests that were inconsistent across operating systems and those that were inconsistent across third-party library or Java Development Kit versions and vendors. For the purposes of Table 5, we consider these as part of the *platform dependency* category. The *resource* category was sub-categorized into *test order dependency*, accounting

```
1     async def test_get_image(hass, hass_ws_client, caplog):
2         """Test get image via WS command."""
3         await async_setup_component(
4             hass, "media_player", {"media_player": {"platform": "demo"}}
5         )
6         await hass.async_block_till_done()
7
8         client = await hass_ws_client(hass)
9
10        with patch(
11            "homeassistant.components.media_player.MediaPlayerEntity."
12            "async_get_media_image",
13            return_value=(b"image", "image/jpeg"),
14        ):
15            await client.send_json(
16                {
17                    "id": 5,
18                    "type": "media_player_thumbnail",
19                    "entity_id": "media_player.bedroom",
20                }
21            )
22            msg = await client.receive_json()
23        assert msg["id"] == 5
24        assert msg["type"] == TYPE_RESULT
25        assert msg["success"]
26        assert msg["result"]["content_type"] == "image/jpeg"
27        assert msg["result"]["content"] == base64.b64encode(b"image").decode("utf-8")
28        assert "media_player_thumbnail is deprecated" in caplog.text
```

Fig. 5. A flaky test from the Home Assistant project [12]. This test was flaky until the addition of line 6 [13] (highlighted), which blocks the calling thread until all pending work to setup the components under test has been completed. Previously, the test might fail if this had not been completed before the assertions starting on line 23 were evaluated.

for 16% of flaky tests, and *resource leak*, accounting for 8%. Finally, the *flaky tests* category consisted of the *asynchronous wait*, *race condition* and *concurrency bugs* subcategories. Considering the latter two as part of the more general *concurrency* category of Table 5, they categorized 18% of flaky tests under *asynchronous wait* and 19% under *concurrency*. The results of this study are different from the previous work of Luo et al. [133], with *asynchronous wait* being only the third most common category, for example. This may be surprising given that they both examined subjects from the same source (i.e., the Apache Software Foundation). However, this study takes bug reports as objects of study, which developers may not have addressed, whereas Luo et al. explicitly analysed commits that *fixed* flaky tests. Therefore, this difference in the results reported by Luo et al. and Vahabzadeh et al. suggests that there may be certain categories of test flakiness that developers are more likely or more able to repair.

3.1.3 Developer Survey. Eck et al. [79] asked 21 software developers from Mozilla to classify 200 flaky tests that they had previously fixed, using the categories of Luo et al. [133] as a starting point, but being allowed to create new ones if appropriate. After reviewing the developers' responses, they identified four new emergent categories. The first was *test case timeout*, which refers to the circumstance in which a test is flaky due to sometimes taking longer than some specified upper limit for single test case executions. Another related category was *test suite timeout*, in which a test times out non-deterministically as before but due to a time limit on the whole test suite execution. Another was *platform dependency*, where a test unexpectedly fails when executed on different platforms, such as across Linux kernel versions, even if the failure is consistent. The final new category was *too restrictive range*, in which some valid output values lie outside of assertion

ranges, making the test fail in these corner cases. Their results showed that the top three categories observed by the developers were *concurrency*, *asynchronous wait* and *too restrictive range*, accounting for 26%, 22%, and 17% of cases, respectively. The *test order dependency* category, one of the top three categories in the previous work of Luo et al., came in fourth, responsible for 9% of cases. Since these insights were also derived from previously *fixed* flaky tests, one could argue that their results are more directly comparable to those of Luo et al. than those of Vahabzadeh et al. [171]. This statement is supported by the fact that the most commonly identified causes of flakiness in this study are similar to that of Luo et al. [133], albeit with *test order dependency* not quite making the top three.

3.1.4 Pull Requests. Lam et al. [121] categorized the type of flakiness repaired in 134 pull requests regarding flaky tests in six subject projects that were internal to the Microsoft corporation. These projects used Microsoft's distributed build system CloudBuild, which additionally contains Flakes, a flaky test management system. By re-executing failed tests, Flakes identifies those that are flaky and generates a corresponding bug report. By examining the bug reports that had been addressed by developers via a pull request, they were able to categorize the respective fixes, using the categories from previous work [133, 147]. They identified *asynchronous wait* as a leading category, though with a significantly higher prominence of 78% of pull requests, in comparison to 37% of commits as found previously by Luo et al. [133]. Furthermore, they found no cases of the *test order dependency* category, which had previously been identified as being common. One possible reason for this is that Flakes only re-executes tests in their original order and does no reordering, therefore making it unlikely to identify, and generate bug reports for, any order-dependent tests.

3.1.5 Execution Traces. Gruber et al. [95] performed an empirical analysis of flaky tests in the Python programming language. To collect subjects, they automatically scanned the Python Package Index [58] and selected projects with a test suite that could be executed by PyTest, a popular Python test runner [28] on which their experimental infrastructure depends. This resulted in 22,352 subject projects containing a total of 876,186 test cases. They executed each test suite 200 times in its original order and 200 times in a randomized order to identify flaky tests including order-dependent tests. Introducing the concept of *infrastructure flakiness*, Gruber et al. split each of these 200 runs into 10 iterations of 20 runs, each executed on the same machine in an uninterrupted sequence. They considered a flaky test to be due to infrastructure flakiness if it was flaky between iterations but not within an iteration—for example, a test case that fails in every run of one iteration but consistently passes every run of every other iteration. They reasoned that such instances were due to non-determinism in the testing infrastructure. For a sample of 100 flaky tests that were neither order-dependent nor due to infrastructure non-determinism, the authors classified their causes based on keywords in the execution traces of the flaky test failures. For instance, they considered keywords such as *thread* and *threading* to be indicative of flaky tests of the *concurrency* category. In total, they identified 7,571 flaky tests among 1006 projects of their subject set. Of these, they found 28% to be due to infrastructure flakiness, 59% due to test order dependencies and 13% due to other causes. Of the 100 automatically classified flaky tests (randomly sampled from the latter 13%), they found the most common category to be *network*, accounting for 42%. This was followed by *randomness* at 37%. At odds with the general trend of previous studies [79, 121, 133, 171], *asynchronous wait* and *concurrency* were minority categories, responsible for 3% of the categorized flaky tests each. The authors put this down to use case differences between the Python and Java programming languages, since these previous studies focused predominantly on subjects of the latter.

Conclusion for RQ1.1: What are the general causes of flaky tests? The overarching picture painted by studies examining commit data, bug reports, developer survey responses, and pull requests suggests that timing dependencies related to asynchronous calls is the leading cause of test flakiness [79, 121, 133, 171]. One study that categorized pull requests of automatically generated bug reports of flaky tests found the prevalence of this category to be as high as 78%. Therefore, techniques for mitigating or automatically identifying such cases would likely address the most significant share of flaky tests. However, another study examining execution traces specifically from projects implemented in the Python programming language [95] found this to be a minority category. Other emergent leading causes include concurrency related issues, other than asynchronicity and platform dependency. The latter of these, accounting for up to 34% of bug reports indicative of flaky tests in one study [171], is particularly relevant in the age of cloud-based continuous integration, where test runs may be scheduled across heterogeneous machines in a seemingly non-deterministic manner depending on availability [79]. Another prominent cause of flaky tests are test order dependencies (i.e., order-dependent tests), representing 9% of flaky test repairs in two studies [79, 133].

3.2 Test Order Dependencies

Multiple sources have specifically considered the causes of, and the factors associated with, the *test order dependency* category of flaky tests. Some of these have discussed the difficulties associated with implementing and executing procedures between test case runs for resetting the program state, specifically setup and teardown methods [67, 101]. Others have examined the impact of global variables, such as *static fields* in the Java programming language, as a vector for conveying information between test cases and thus establishing order-dependent tests [67, 68, 143, 185]. One study posited that failing to control all the inputs that may later impact assertions leaves tests open to establishing dependencies on one another [106]. An example of an order-dependent test, and the test that can cause it to fail, is given in Figure 6, taken from Facebook's Hydra project for the configuration of Python applications [15].

3.2.1 Classification. Shi et al. [159] introduced the terminology of *victim* and *brittle* for describing order-dependent tests. A victim test passes in isolation but fails when executed after certain other tests. The tests that cause a victim to fail are known as the victim's *polluters*. Conversely, a brittle test fails in isolation and requires the prior execution of other tests to pass, known as *state-setters*. Shi et al. performed an evaluation using 13 modules of open-source Java projects containing 6,744 tests in total. By executing the respective test suites in randomized orders, they identified 110 order-dependent tests. Of these, 100 were victims and 10 were brittles.

As part of their empirical evaluation of flaky tests in Python projects, Gruber et al. [95] followed a similar procedure upon 22,352 Python projects from the Python Package Index [58]. They identified 4,461 order-dependent tests, 3,168 of which were victims and 738 of which were brittles. The authors were unable to categorize the remaining 555 due to limitations in their testing framework. Overall, the combination of these two studies indicates that the vast majority of order-dependent tests would pass in isolation but may fail due to the action of other tests (i.e., they are polluters).

3.2.2 Setup and Teardown. Haidry et al. [101] reasoned that test order dependencies are a result of interactions between components of the software under test. Giving an example of a hypothetical piece of software for running an automated teller machine (ATM) machine, they explained that, since the machine requires the user to enter their personal identification number (PIN) before inputting the desired amount of cash, the test case that covers the PIN component must be executed before the test case that covers the cash withdrawal component to ensure that the system

```
1   @pytest.mark.parametrize(
2       "config_dir", ["../hydra/test_utils/configs"],
3   )
4   @pytest.mark.parametrize(
5       "config_file, overrides, expected",
6       [
7           (None, [], {}),
8           (None, ["foo=bar"], {"foo": "bar"}),
9           ("compose.yaml", [], {"foo": 10, "bar": 100}),
10          ("compose.yaml", ["group1=file2"], {"foo": 20, "bar": 100}),
11      ],
12  )
13  class TestCompose:
14      def test_compose_decorator(
15              self, hydra_global_context, config_dir, config_file, overrides, expected
16          ):
17              with hydra_global_context(config_dir=config_dir):
18                  ret = hydra.experimental.compose(config_file, overrides)
19                  assert ret == expected
20
21      def test_strict_failure_global_strict(
22          self, hydra_global_context, config_dir, config_file, overrides, expected,
23      ):
24          overrides.append("fooooooooo=bar")
25          with hydra_global_context(config_dir=config_dir, strict=True):
26              with pytest.raises(KeyError):
27                  hydra.experimental.compose(config_file, overrides)
```

Fig. 6. An example of an order-dependent test from the Hydra project [15]. The test `test_strict_failure_global_strict` appends a string to the list passed as the `overrides` argument on line 24, which is part of a class-wide test parametrization given on line 4. The test `test_compose_decorator` expects `overrides` to be of its initial value and would fail if executed after `test_strict _failure_global_strict`, as may other tests in `TestCompose` [35] that unexpectedly receive the modified version of `overrides`.

is in the required state. A developer could write each of these test cases to be independent of one another, but that would require some amount of additional setup for each of them, resulting in a less efficient test suite. Therefore, test order dependencies may be a consequence of developers prioritizing efficiency and convenience over test case independence.

Having studied the prevalence of per-test process isolation as a mitigation for test order dependencies, which involves executing each test in its own process to prevent side effects, Bell et al. [67] suggested that the burden of writing setup and teardown methods could be a factor in the existence of test order dependencies. Setup methods, which ensure that the state of the program is as expected before a test is executed, and teardown methods, which reset the state of the program to as it was before the test run, are a means of avoiding side effects between tests, meaning isolation should not be necessary. Bell et al. remarked, however, that these methods may be difficult to implement correctly, perhaps explaining their observation that 41% of 591 sampled open-source projects used isolated test runs as a catch-all attempt to eliminate side effects and thus test order dependencies.

3.2.3 Static Fields. Muşlu et al. [143] remarked how the implicit assumptions of developers regarding the way in which their code is executed, such as the ordering of method calls and data dependencies, leads to a class of bugs that, when testers are unaware of such assumptions, are likely to be overlooked. They explained how this can manifest itself as test cases that depend on

the execution of other tests in a test suite and behave differently when executed in isolation. Such tests violate the *test independence assumption* that all tests are isolated entities, something that they stated is rarely made explicit, thus exacerbating the problem. The authors went on to describe and evaluate a technique for manifesting these implicit dependencies and found the improper use of Java static fields to be a potential vector.

Of the 96 instances of order-dependent Java tests identified by keyword searches in various issue tracking systems, Zhang et al. [185] found a majority of 73 requiring only one other test to manifest the test order dependency. In other words, 76% of their sample were observed to produce a different outcome if executed after only one other test in isolation from the rest of the test suite, compared to the outcome when running the test suite as normal. They found that 61% of order-dependent tests were facilitated by a static field, for example, one test modifies some object pointed to by a static field somewhere, which is later used by a subsequent test, with the remainder caused by side effects within external resources such as files, databases or some other unknown cause. This finding roughly concurs with that of Luo et al. [133], who found that 47% of order-dependent tests were caused by dependencies on external resources.

When introducing their test virtualization approach, Bell et al. [67] described static fields as potential points of "leakage" between test runs, becoming possible avenues for tests to share information or resources and thus become dependent upon one another. Through the evaluation of their order-dependent test detection approach in a later study, Bell et al. [68] discovered that the test order dependencies identified by their tool were facilitated by a very small number of static fields. In other words, many order-dependent tests were caused by shared access to the same static fields in a Java program. Further examination revealed that most of these were within internal Java runtime code, as opposed to application or test code, and all of them were of a primitive data type.

3.2.4 Brittle Assertions. Huo et al. [106] reasoned that *brittle assertions*, or assertion statements that check values derived from inputs that the test case does not control, can give rise to order-dependent tests. Tests containing brittle assertions generally make an implicit assumption that the program state that constitutes the uncontrolled inputs will always be of some particular value, often their default value. This assumption can facilitate a test order dependency if another test modifies this program state before the evaluation of the brittle assertion. The authors gave an example of a Java test class containing a test method that assumes various instance fields of an object under test are set to their default values. In the case where a previously executed test method modifies the values of these fields, the former test becomes order-dependent, stemming from the uncontrolled, assumed default, field values.

Conclusion for RQ1.2: What are the causes of order-dependent tests? Studies have attributed test order dependencies to the difficulty of implementing, and the added computational cost of executing, setup and teardown procedures between test case runs [67, 101]. In terms of how they are facilitated, one source found 61% of order-dependent tests to have been caused by global variables, specifically static fields, as opposed to external factors such as files, and that the majority, 76%, appeared to depend on only one other test [185]. Furthermore, another study found that most test order dependencies are conveyed via a relatively small number of static fields, mostly found within internal runtime code as opposed to application or test code [68]. An additional cause that has been associated with order-dependent tests is related to the assumption that certain parts of the program state will always be of some particular value. This gives rise to tests that do not fully control all the inputs evaluated within their assertions, which may go on to become order-dependent if other tests modify such inputs [106].

3.3 Application-specific Causes

As well as the causes of flakiness in general projects, studies have examined the factors associated with flaky tests specific to particular applications or domains. There exists a line of work explicitly concerned with flakiness within the test suites exercising user interfaces [89, 90, 153, 155]. In addition, given the rise in popularity of machine learning techniques, studies have begun to consider the particularities of flaky tests within such projects [78, 145]. Furthermore, following a similar methodology to Luo et al. [133], one source categorized flaky tests specific to Android projects [170].

3.3.1 User Interface Testing. Gao et al. [89] discussed the causes of flakiness in GUI regression testing. The process of GUI regression testing requires the test harness to take measurements about particular GUI state properties, such as the positions of various elements, which are then compared with measurements taken of a later version to identify unexpected discrepancies. During the execution of test cases, GUI test oracles evaluate an extensive range of properties, including those that may be tightly coupled with the hardware on which they are executed, such as screen resolution, meaning test outcomes can be inconsistent across machines. Furthermore, since it is difficult to predict the amount of time required to render a GUI, or due to instability within the GUI state itself, the point in time at which to take these measurements may be unclear. In other words, if measurements are taken too early, before the GUI has been fully drawn, then the values of these properties may be inaccurate. All these factors contribute to the potential for these properties to become highly unstable, something which has been quantified in terms of entropy [90].

A later study by Presler-Marshall et al. [153] evaluated the impact of various environments and configurations of Selenium [32], a framework for automating UI tests in web apps, upon test flakiness. Specifically, they analysed the effects of varying five properties of the execution platform. The first was the waiting method, used by Selenium to wait for the various UI elements to appear before evaluating test oracles. As previously explained, tests may be flaky if they are too eager to evaluate assertions before the web browser has fully rendered the page. The second was the web driver, Selenium's interface for controlling a web browser to execute tests. In the context of this study, the term web driver was used interchangeably with web browser, since they have a one-to-one correspondence with respect to their evaluation. The web drivers they evaluated were Chrome [10], Firefox [25], PhantomJS [26], and HtmlUnit [14], the latter two being *headless*, meaning that they do not render a graphical display and are used primarily for testing and development purposes. The third factor was the amount of memory available to the testing process, doubling from 2 GB up to 32 GB. The fourth was the processor, which they varied between a notebook CPU, supposed to represent what a software engineering student might use, and a much more powerful workstation processor. The final factor was the operating system, specifically Windows 10 or Ubuntu 17.10. The object of their analysis was an on-going software engineering student project containing nearly 500 Selenium tests. Under five configurations of the five described properties, they counted the number of test failures after 30 test suite executions, knowing that tests ought to pass and so a failure would be indicative of flakiness. The authors found that Java's `Thread.sleep` [19] with a fixed time delay as a waiting method yielded the fewest test failures of all the methods evaluated, compared with not waiting at all or using more dynamic, conditional waiting methods. They also found that a faster CPU resulted in substantially fewer flaky tests, presumably since it would render web pages fast enough to ensure that the waiting method would have less of an impact.

Romano et al. [155] performed an empirical evaluation examining, among other things, the causes of flakiness in user interface testing. As subjects, they took both web and Android applications, and, following a methodology inspired by Luo et al. [133], manually inspected 3,516 commits

from 7,037 GitHub repositories. Eventually, they narrowed down their objects of analysis to 235 distinct flaky tests and categorized them by their root causes into four broad categories, each broken down into subcategories. Their first broad category, *asynchronous wait*, contained 45% of the 235 flaky tests and consisted of three subcategories. The first subcategory was *network resource loading* and describes flaky tests attempting to manipulate remote data or resources that, depending on network conditions, are not fully loaded (or have failed to load at all). The second was *resource rendering* and refers to the case where a flaky test attempts to perform an action upon a component of the user interface before it is fully drawn. The third was *animation timing*, containing flaky tests that rely on animations in the user interface and are thus sensitive to timing differences across different running environments. Their second broad category was *environment*, consisting of two subcategories, and containing 19% of their dataset. The first subcategory was *platform issue*, regarding issues with one particular platform that may cause flakiness, and the second was *layout difference*, pertaining to differences in layout engines in different internet browsers. The third broad category was *test runner API issue*, accounting for 17% of the flaky tests. This was broken down into *DOM selector issue*, problems interacting with the Document Object Model (DOM) due to differences in browser implementation, and *incorrect test runner interaction*, incorrect behavior within APIs provided by the test runner. Finally, the last broad category was *test script logic issue* and contained 19% of the flaky tests. This was broken down into the familiar *unordered collections*, *time*, *test order dependency*, and *randomness*, and contained the additional subcategory *incorrect resource load order*, describing flaky tests that load resources after the calls that load the tests, causing the tested resources to be unavailable at test run time.

3.3.2 Machine Learning. Nejadgholi et al. [145] studied *oracle approximations* in the test suites of deep learning libraries. An oracle approximation in this context refers to the range of acceptable output from some deep learning-based software under test, which are naturally probabilistic in nature, as estimated by a developer. These ranges are expressed as assertion statements within test cases that, due to their probabilistic nature, often take the form of a range of acceptable numerical values or some approximate equality operator with a defined tolerance. This is in contrast to, for example, a simple equality test with a pre-defined value that might be applicable when testing a deterministic algorithm but may be inappropriate in this context. They proposed a set of assertion patterns that express oracle approximations and identified instances of them across four popular deep learning libraries implemented in the Python programming language. Across these projects, between 5% and 25% of all assertions were deemed to be oracle approximations. Since an oracle approximation may be overly conservative in certain circumstances, this may lead to tests failing inconsistently for corner cases and thus being flaky tests as part of the *too restrictive range* category [79].

Dutta et al. [78] argued that machine learning and probabilistic applications suffer especially from test flakiness as a result of non-systematic testing specific to their domain. They examined 75 previously fixed flaky tests from the commit and bug report data of four popular open-source machine learning projects written in Python and categorized their causes and fixes. They devised their own categories to be more specific regarding the origin of the flakiness in their particular subject set, explaining that under the categories of Luo et al. [133], they would have mostly been categorized under *randomness*, which would be too general for this more specific study. They categorized the majority of flaky tests, 60% of those sampled, as being caused by *algorithmic nondeterminism*, that is, non-determinism inherent in the probabilistic algorithms under test common in machine learning projects, such as Bayesian inference and Markov Chain Monte Carlo among others. For instance, they described the situation in which a developer wishes to test a statistical model, selecting a small dataset upon which to train it. If the developer does not account for the

Table 6. Leading Causes of Flakiness in Specific Domains

Source	Domain	Leading Cause
[89, 90, 153, 155]	User interface testing	Asynchronous wait
[78, 145]	Machine learning	Randomness
[170]	Mobile applications	Concurrency

non-zero chance that the training algorithm does not converge after some number of iterations, then any assertions regarding the parameters of the trained model may fail in some cases. They associated five of their sampled tests with incorrect handling of floating point computations (e.g., rounding errors and the mishandling of corner cases such as `NaN`). A further four were categorized under incorrect API usage. Unsynchronized seeds—in which the program inconsistently set the seeds of different random number generators within different libraries—were blamed for two flaky tests. The remainder were split between concurrency or hardware related causes, or were labelled by the authors as *other* or *unknown*.

3.3.3 Mobile Applications. Thorve et al. [170] claimed that Android apps have a particular set of characteristics that may make them specifically vulnerable to test flakiness. These include *platform fragmentation*—the vast number of versions and variants of the Android operating system available—and the *diversity of interactions*—the functionality within Android apps, which typically involves many third-party libraries, networks and hardware with highly variable specifications (e.g., screen size and processing power). With a methodology inspired by Luo et al. [133], they searched through the historical commit data of Android projects on GitHub for particular keywords associated with flaky tests. In total, they identified 77 commits across 29 Android projects that appeared to fix flaky tests. They categorized 36% as related to concurrency. Unlike Luo et al. [133], they considered the *asynchronous wait* category as part of the *concurrency* category. They categorized 22% as based upon assumptions or dependencies regarding specific hardware, the Android version or the version of particular third-party libraries. A further 12% they categorized as being attributable to erroneous program logic, specifically regarding corner-case program states. The remaining flaky tests were categorized as being caused by either network or user interface related non-determinism, or were too difficult to classify.

Fazzini et al. [83] highlighted the significant number of interactions between mobile apps and their software environment as a cause of test flakiness. Specifically, they referred to external calls to application frameworks that collect data from the network, location services, camera, and other sensors. They explained that a common strategy to mitigate such flakiness is to create test mocks for these sorts of external calls, providing tests with fictitious but deterministic data [163].

Conclusion for RQ1.3: What are the application-specific causes of flaky tests? In the context of user interface testing, an insufficient waiting mechanism to allow the interface to be fully rendered and stable before executing tests appears to be strongly associated with flakiness in this domain, in a manner comparable to that of flaky tests related to asynchronous waiting [89, 153, 155]. For machine learning projects, overly conservative approximations regarding the valid output of probabilistic algorithms and the high degree of algorithmic non-determinism in general appears to be a major cause of flaky tests [78, 145]. The distribution of causes of flaky tests within Android applications appears to roughly correspond to what has been observed in more general efforts to categorize flakiness [79, 121, 133, 171], with concurrency and dependencies upon the highly variable execution platform identified as major causes [170].

4 COSTS AND CONSEQUENCES

This section presents evidence of the negative consequences of test flakiness to illustrate its prominence as a problem. Following the same methodology as outlined in Section 2, the goal of this section is to answer **RQ2** by posing three sub-research questions, each answered in their own respective subsections, with findings summarized in Table 7. These three questions aim to capture the variety of discussions and analyses performed in the literature regarding the impacts and costs of flaky tests, and are as follows:

- **RQ2.1: What are the consequences of flaky tests upon the reliability of testing?** Answered in Section 4.1, this question provides insights into the extent to which test flakiness detracts from the value of testing by casting doubt on the meaning of a test's outcome. Our answer examines how flaky tests manifest themselves, with respect to incorrectly indicating the presence of a bug or allowing one to go unnoticed, consulting studies that analysed issue trackers and bug reports. We go on to consider the impact that ignoring flaky tests can have on the volume of crashes experienced by users, demonstrating that, while unreliable, flaky tests may still provide some testing value and thus should not always be immediately deleted by developers.
- **RQ2.2: What are the consequences of flaky tests upon the efficiency of testing?** Answered in Section 4.2, this question studies how flaky tests could impose costs on the time taken to receive useful feedback from test suite executions. By reviewing several empirical studies, our answer specifically examines how a particular category of flaky test poses a threat to the validity of a variety of techniques designed to accelerate the testing process. We concluded the response to this question by examining how this category impacts research in the area.
- **RQ2.3: What other costs does test flakiness impose?** Answered in Section 4.3 and acting as a catch-all for all of the relevant sources not consulted in answering the previous questions, our answer to this question aims to identify all other testing-related areas that are negatively affected by flaky tests.

4.1 Consequences for Testing Reliability

Since the goal of a software test is to provide a signal to the developer indicating the presence of faults, a flaky test that may pass or fail, regardless of program correctness, represents a signal with some amount of noise. In a test suite containing multiple flaky tests, this noise accumulates and may result in an unreliable test suite. Luo et al. [133] explained three deleterious effects that flaky tests can have on the reliability of testing. First, since non-determinism is inherent to test flakiness, attempting to reproduce failures of flaky tests can be more difficult and less valuable. One study empirically evaluated the reproducibility of flaky tests on a large scale [124]. Second, because flaky tests can fail without modifications to the code under test, they can waste developer time via debugging a potentially spurious failure that is unrelated to any recent changes. Multiple studies have examined the manifestation of flaky tests in terms of indicating the presence of non-existent bugs, or conversely, missing real bugs [171, 185]. Last, despite flaky tests being an unreliable signal, they may still convey some useful information as to the correctness of the code under test, however, a frequently failing flaky test may eventually be ignored by a developer, thus potentially causing genuine bugs to be missed. To that end, one source found that ignoring flaky test failures may have a negative effect on software stability, measured in the number of crash reports received for the Firefox web browser [154].

4.1.1 Reproducibility. Lam et al. [124] posited that, when encountering a failing test during a test suite run, a developer is likely to run that test in isolation to reproduce the failure and thus

Table 7. Summary of the Findings and Implications Answering RQ2: What Are the Costs and Consequences Associated with Flaky Tests?

	Finding	Implications	Source
</>	Of 107 tests identified as flaky by repeatedly executing whole test suites, only the flakiness of 50 could be **reproduced by repeatedly executing them in isolation**.	Flaky tests seem more consistent when run in isolation, meaning attempts to reproduce their flakiness this way might be challenging, but **reproducing their failure for debugging purposes may still be effective**.	[124]
</>	Of 96 sampled order-dependent tests, 94 reportedly caused a test failure in the absence of a bug, i.e., a **false alarm**, the remainder caused a bug to go unnoticed. Of a sample of 443 test bug reports, 97% led to a false alarm, and of these, 53% were flaky tests.	The dominant consequence of flaky tests appears to be false alarms and flaky tests also appear to be their leading cause. This indicates that flaky tests may be a considerable **waste of a developer's time**, since they may attempt to go looking for a bug that does not exist.	[171, 185]
</>	Production builds of the Firefox web browser with one or more flaky test failures were associated with a median of 234 **crash reports**. Builds with one or more test failures in general were associated with a median of 291.	Despite being less reliable, **flaky test failures should not be ignored**, in the same way that any test failure should not be ignored.	[79]
</>	Of the test execution logs of over 75 million builds on Travis, 47% of previously failing, **manually restarted builds** subsequently passed, indicating that they were affected by flakiness. Projects with restarted builds experienced a **slow down of their pull request merging process** of 11 times compared to projects without.	Flaky tests can threaten the **efficiency of continuous integration** by intermittently failing builds and requiring manual intervention, hindering the process of merging changes and stalling a project's development.	[77]
🎓	At **Google**, of the 115,160 test targets that had previously passed and failed at least once, 41% were flaky. Of the 3,871 distinct builds sampled from **Microsoft**'s distributed build system, 26% failed due to flakiness.	Test flakiness has been reported within **all sectors of the software engineering field**, from independent open-source software projects to the proprietary products of some of the world's largest companies.	[120, 138]
</>	Over a 30-day period of test execution data, 0.02% of test executions resulted in flaky failures, though had these not been automatically identified by the testing infrastructure, **5.7% of all failed builds would have been due to these flaky tests**.	A relativity small number of flaky tests in a test suite can have a **significant impact on the number of subsequently failed builds** as part of a continuous integration system.	[121]
🎓 </>	Across the developer-written test suites of 11 Java modules, 23% of **previously passing order-dependent tests failed** after applying test prioritization, 24% after applying test selection and 5% after parallelizing tests.	Order-dependent tests in particular **limit the applicability of test acceleration techniques** by causing inconsistent outcomes in the test runs they produce.	[123]
🎓	Flaky tests are a **threat to the validity** of experimental test acceleration algorithms, with researchers taking steps to filter them from their evaluation methodologies.	Researchers should understand that flaky tests are **inevitable in real-world test suites** and should give forethought to their implications when creating methods that reorder or otherwise modify test suite runs.	[129, 134, 150, 181]
🎓	Flaky tests have been shown to detract from the applicability of **mutation testing, fault localization, automatic test generation, batch testing and automatic program repair**.	Approaches for dealing with flaky tests would likely have **far-reaching impacts**, beyond the obvious benefits of improving test suite reliability.	[136, 144, 156, 157, 172, 179]
🎓	Of 89 **student submissions** of a software engineering assignment, 34 contained at least one flaky test.	Flaky tests may hinder software engineering education, which, while representative of industry experience, may place an **undue burden** upon students.	[158]

Findings relevant to researchers are marked with 🎓. Findings relevant to developers are marked with </>.

debug the code under test. When trying to reproduce the inconsistent outcome of a flaky test, they demonstrated that this technique may not be effective. They executed the test suites of 26 modules of various open-source Java projects 4,000 times and calculated each test's failure rate—the ratio of failures to total runs. For each test they found to be flaky, meaning a failure rate greater than zero or less than one, they re-executed it 4,000 times again in isolation. They found that, of the 107 tests identified as flaky, 57 gave totally consistent outcomes in isolation. Furthermore, they found that the failure rates of flaky tests appear to differ when executed in isolation. Of the 50 flaky tests that were reproduced in isolation, 19 had lower failure rates and 28 had higher ones. A Wilcoxon signed-rank test demonstrated a statistically significant difference between the two samples, suggesting that the failure rates when executed within the test suite were significantly different to the failure rates when executed in isolation. This result shows that flaky tests appear to behave more consistently when executed in isolation and so this may be a good strategy if trying to reproduce the *failure* of a flaky test, but not its *flakiness*.

4.1.2 Manifestation. Zhang et al. performed an empirical study of test order dependencies [185]. They pointed out that, unlike other sorts of flaky tests, order-dependent tests would only be manifested when the test suite changed by adding, removing or reordering test cases. They described how order-dependent tests can mask bugs, since the order in which the containing test suite is executed might determine whether or not the dependent test is able to expose faults or not. They also explained how test order dependencies might lead to spurious bug reports should the test run order be changed for any reason, potentially manifesting the order-dependent tests and exposing an issue in the test code (e.g., improper setup and teardown) rather than a bug in the code under test, something that a developer may have to allocate time to identify themselves. They went on to search for four phrases related to test order dependency in five software issue tracking systems to understand the manifestations of order-dependent tests in Java projects. In total, they found 96 such tests, 94 of which caused a false alarm, a test failure in absence of a bug, and two caused a missed alarm that was, in other words, a bug in absence of a test failure. The ramifications of a missed alarm are potentially more serious than a false alarm, since they may allow a bug to go unnoticed and thus persist after testing.

A later and more general investigation into the prevalence and nature of bugs in test code was performed by Vahabzadeh et al. [171]. Through their manual analysis, they identified 5,556 unique bug reports across projects of the Apache Software Foundation [3], 443 of which they systematically categorized. They categorized these by manifestation, that is, whether the test bug led to a false alarm or to a missed alarm, or a "silent horror" as they termed them, and by their root cause. They found that 97% of test bugs manifested as a false alarm, with test bugs categorized as flaky tests, or due to environmental reasons or mishandling of resources (also types of flaky test according to the definition in Section 1), responsible for 53% of these. False alarm test bugs, while potentially less serious than "silent horrors," can still take a considerable amount of time and effort for developers to debug. Since flaky tests are likely the dominant root cause of such bugs this means that they are potentially a significant waste of a developer's time. Ultimately, the findings of both of these studies suggest that flaky tests are more likely to manifest as a spurious test failure rather than an unnoticed bug, though as the operative word in the latter case suggests, it might simply be that such instances are under reported, because they are, by definition, unnoticed by developers and researchers.

4.1.3 Ignoring Flakiness. Rahman et al. [154] examined the impact of ignoring flaky test failures on the number of crash reports associated with builds of the Firefox web browser [25] in both the beta and production release channels. To identify flaky tests, they searched for records of test executions marked with the phrase "RANDOM," a term commonly used by Firefox developers, in

the testing logs. In the median case, they found that builds with one or more flaky test failures were associated with a median of 514 and 234 crash reports for the beta and production channels, respectively. Furthermore, a Wilcoxon signed-rank statistical test comparing the numbers of crashes between the two channels showed a statistically significant difference, with Rahman et al. conjecturing that developers were more conservative about releasing production builds with known flaky tests. For those with failing tests in general, these figures were 508 crash reports for the beta channel and 291 for the production channel. In the case of builds with all tests passing, they recorded a median of only two crash reports for each channel. These findings indicate that ignoring test failures, flaky or otherwise, appears to lead to a considerably higher volume of crashes due to missed bugs.

Conclusion for RQ2.1: What are the consequences of flaky tests upon the reliability of testing? One study found that, of 107 flaky tests identified by repeatedly executing their test suites, only 50 could be reproduced as flaky by repeatedly executing them in isolation. These findings suggest that reproducing the flakiness of a flaky test by executing it in isolation may be difficult, though it may still be effective for witnessing the failing case for debugging purposes [124]. In terms of their manifestation, one source identified 94 of 96 sampled order-dependent tests to have caused a false alarm, meaning that they failed in the absence of a real bug [185]. Another found that 97% of bugs in test code manifested as a false alarm, with flaky tests representing 53% of these [171]. Overall, these results indicate that flaky tests may be a leading cause of test failures that are false alarms. One study found that builds of the Firefox web browser with one or more flaky test failures were associated with a median of 234 crash reports in the production channel. This is compared to a median of 291 crash reports associated with builds containing one or more failing tests in general and two reports associated with builds that had all passing tests. These results suggest that ignoring test failures, even if they are flaky, can lead to a higher incidence of crashes [154].

4.2 Consequences for the Efficiency of Testing

Projects can benefit from using continuous integration to automate the process of running tests and merging changes [105]. However, since continuous integration may result in a greater volume of test runs, it can highlight the true extent of flakiness in a test suite [119]. In a continuous integration system, when a developer has been working on their own branch and goes to merge their changes, their code is remotely built and a test suite run is triggered to ensure their changes have not introduced bugs. Should any tests fail, their changes will be rejected. Naturally, given the association between flaky tests and false alarm failures [171, 185], flaky tests can limit the efficiency of the continuous integration process by incorrectly causing builds to fail [77, 118, 120, 121, 138]. In addition, since the majority of software build time is spent on test execution [68], numerous test acceleration approaches have been developed for quickly retrieving the useful information from test runs [110]. Test selection is a technique for speeding up a test run by only executing tests deemed likely to indicate a bug, for example, by only executing those test cases that cover recently modified code [129, 134, 160]. Test prioritization is used to reorder a test suite such that the test cases likely to reveal faults are executed sooner, achieved by sorting test cases by, for instance, the most modified code elements covered, for [101, 150, 181]. Test parallelization is an approach for taking advantage of a multi-processor architecture by splitting the execution of a test suite over multiple processors, potentially resulting in significant speedups [68, 73, 100]. Since all of these techniques shrink, reorder or split up a test suite (i.e., change the test execution order), they run the risk of breaking test order dependencies, resulting in inconsistent outcomes for the order-dependent tests that have them [68, 123, 185]. This is problematic, since, even though these

techniques may reduce testing time, if they cause inconsistent test outcomes, then the results of the optimized test run may become unreliable by causing false alarm failures, as previously explained in Section 4.1.2. Studies have shown that flaky tests do impact the results and deployment of these techniques [73, 123, 185], to the extent that researchers have taken steps to filter them as part of their evaluation methodologies [129, 134, 150, 181]. A recent mapping study of test prioritization in the context of continuous integration found that 13% of sampled publications on the topic identified flaky tests as a problem [152].

4.2.1 Continuous Integration. Lacoste [119], a Canonical developer working on the collaborative development service Launchpad [22], described his experiences of transitioning to a continuous integration system in 2009. His account provided an insight into how the constant execution of tests in a continuous integration system can expose flakiness. He remarked how an intermittent failure caused by a "malfunctioning test," his description for a flaky test, was a potential reason for a branch being rejected by their old "serial" integration system, thereby causing the team to miss a release deadline. Upon transitioning to a continuous integration system, developers witnessed the magnitude of the flakiness in their test suites, since tests were being executed more often. In particular, he found that the "integration-heavy" tests, that covered multiple processes, were the most problematic ones in the studied test suite.

At a software developed company, called Pivotal Software, a survey deployed by Hilton et al. [104] asked developers to estimate the number of continuous integration builds failing each week due to genuine test failures and those due to flaky test failures. They found no statistically significant difference between the two distributions of estimates after performing a Pearson's Chi-squared test. This indicated that developers at Pivotal experienced similar numbers of both genuine and flaky failures, something that the manager at Pivotal indicated was a surprising finding.

As part of a study into the cost of regression testing, Labuschagne et al. [118] examined the build history of 61 open-source projects that were implemented in the Java programming language and used the Travis continuous integration service [36]. In each case where a build had previously transitioned through *passed*, *failed* (indicating that the build had failed tests) and back to *passed* again consecutively, they re-executed the build at the failing stage three times. They found that just under 13% of 935 such failing builds in their dataset had failed due to flaky tests.

Durieux et al. [77] performed an empirical study into the prevalence of, and the reasons behind, manually restarted continuous integration builds. Under normal circumstances, a build would be triggered following a code change. If the build fails, due to compilation errors or failed tests, then the change is rejected. If a developer decides to manually trigger a build, without making any changes, then it is referred to as a *restarted build*. They analysed the logs from over 75 million builds on the Travis system. They found that 47% of previously failing builds that were restarted subsequently passed, indicating that they were affected by some non-deterministic behavior, since no code change was involved. They went on to identify inconsistently failing tests to be the main reason for developers restarting builds. Furthermore, they found that projects with restarted builds experienced a slow down of their pull request merging process of 11 times compared to projects without restarted builds. This result suggests that flaky tests have the potential to hinder the continuous integration process by spuriously failing builds and requiring manual developer intervention.

Additionally, Memon et al. [138] performed a study into reducing the cost of continuous testing at Google. They described flaky tests as a reality of practical testing in large organizations, citing them as a constraint to the deployment of the results of their work. Specifically, they identified approximately 0.8% of their total dataset of over five million "test targets," a term used by Google to describe a build-able and executable code unit labeled as a test, as flaky. Of the 115,160 test targets that had historically passed at least once and failed at least once, flaky tests constituted 41%.

Microsoft's distributed build system, CloudBuild, was the object of analysis in a study by Lam et al. [120]. By examining the logs from repeated test executions, these authors identified 2,864 unique flaky tests across five projects. Overall, of the 3,871 individual builds that they sampled, 26% presented flaky test failures. Furthermore, data from Flakes, the flaky test management system integrated into CloudBuild, was used in a later study [121]. Over a 30-day period, the authors found that Flakes identified a total of 19,793 flaky test failures across six subject projects, representing just 0.02% of the sampled test executions. However, had the Flakes tool not identified such cases, they reported that flaky test failures would have been responsible for a total of 1,693 failed builds, which would have represented 5.7% of all the failed builds sampled. This finding suggests that, even when flaky tests are not particularly prevalent, the percentage of builds that may be affected by flaky test failures can be relatively significant.

4.2.2 Test Acceleration. As part of an empirical study of test order dependence, Zhang et al. [185] assessed how many test cases gave inconsistent outcomes when applying five different test prioritization schemes across the test suites of four open-source Java projects. Their results showed that, for one subject in particular with 18 previously identified order-dependent tests, up to 16 gave a different outcome compared to a non-prioritized test suite run. This finding indicated that the soundness of test prioritization was, at least for this project, significantly impacted by order-dependent tests—even though test prioritization methods are not supposed to affect the outcomes of tests at all.

A later evaluation into the impact of test parallelization on test suites for Java programs was conducted by Candido et al. [73]. They measured the speedup of the test suite run time and the percentage of previously passing tests, which now failed, indicating flakiness, under five parallelization strategies across 15 subjects. Their results showed that flaky tests were manifested within four projects under every strategy and that 11 projects had totally consistent outcomes for at least one. Their results also indicated that the finer-grained techniques that involved parallelization at the test method level, as opposed to just the test class level, manifested the most flaky tests. These techniques were also the most effective, indicating that the speedup achieved and the reliability of the test outcomes could be a trade-off for developers to consider when order-dependent tests are present in the test suite.

Lam et al. [123] evaluated the consequences of order-dependent tests upon the soundness of regression test prioritization, selection and parallelization. They evaluated a variety of algorithms upon 11 modules of open-source Java projects with both developer-written and automatically generated test suites with the order-dependent tests already identified in previous work [122]. After applying test prioritization, they found a total of 23% of order-dependent tests in their subjects' developer-written test suites failed and 54% of such tests in their automatically generated test suites failed, when they were previously passing. For test selection, this was 24% and 4% for developer and automatic test suites, respectively, and, for parallelization, 5% and 36%, respectively.

Several studies have revealed that researchers are aware of the negative impacts that flaky tests can have on attempts to improve testing efficiency, as they often take steps to remove them in their evaluation methodologies. Leong et al. [129] compared the performance of several test selection algorithms using one month's worth of data from Google's Test Automation Platform. They demonstrated that, for one particular algorithm that prioritized tests based on how frequently they had historically transitioned between passing and failing, its performance was better when evaluated without flaky tests. Specifically, when evaluated with a dataset containing flaky tests, for up to 3.4% of historical commits, the algorithm did not select one or more tests whose outcome had changed, compared to up to 1.4% when flaky tests were filtered out. Since the goal of test

selection is to only execute informative tests, by missing more tests whose outcome had changed (suggesting that a particular commit may have introduced/fixed a bug), the algorithm performed worse at its intended purpose. They used this as justification for filtering flaky tests in their wider evaluation methodology. Similarly, Peng et al. [150] found that, when comparing test prioritization approaches based on information retrieval techniques, they mostly appeared to perform better when evaluated with a dataset filtered of flaky tests.

When evaluating a data-driven test selection technique deployed at Facebook, Machalica et al. [134] remarked how flaky tests represented a significant obstacle to the accuracy of their approach. Their test selection technique used a classification algorithm, that was trained with previous examples of failed tests and their respective code changes, to only select tests for execution when it predicted that they might fail given a new code change. Finding that the set of flaky tests in their dataset was four times larger than the set of deterministically failing tests, they explained that if they did not filter flaky tests from their training data then they would run the risk of training their classifier to capture tests that failed flakily rather than those that failed deterministically due to a fault introduced by a code change. Yu et al. [181] faced similar problems with their machine learning-driven approach for test case prioritization in the context of user interface testing. They identified flaky tests as a threat to the internal validity of their research, admitting that they did not filter flaky tests from their dataset and thus could have limited the accuracy of their trained model. As future work, these authors planned to explore ways to tackle test flakiness.

Conclusion for RQ2.2: What are the consequences of flaky tests upon the efficiency of testing? One study found that 13% of failing builds across 61 projects using the Travis system, which had previously passed and then immediately went on to pass again, were caused by flaky tests [118]. Similarly, of a sample of over 75 million builds, 47% were manually restarted builds that previously failed and subsequently passed, indicating that they were affected by non-deterministic behavior such as flaky tests [77]. At Google, one study found that 41% of "test targets" that had previously passed and failed at least once were flaky [138]. At Microsoft, another study identified flaky test failures within 26% of all the builds they sampled [120]. Another Microsoft study demonstrated that the 0.02% of flaky test failures from over 80 million test executions, over a 30-day period, could have been responsible for what would have been 5.7% of all the failed builds in that period, had they not been identified [121]. These results indicate that flaky tests can limit the efficiency of continuous integration by spuriously failing builds, thereby requiring manual intervention from developers. Evaluating several test parallelization strategies across 15 projects, one study found that order-dependent flaky tests were manifested within 11 of the 15 under at least one strategy [73]. Across a range of developer-written test suites with previously identified test order dependencies [122], another source found that 23% of order-dependent tests failed, when previously passing, after applying test prioritization, 24% after applying test selection and 5% after parallelizing tests [123]. These findings suggest that order-dependent tests in particular are an obstacle to the applicability of test acceleration techniques. Flaky tests may also hinder the formulation and evaluation of new test acceleration methods. One study found that when evaluating a transition-based test selection algorithm, it failed to perform correctly in 3.4% of cases when flaky tests were present in the evaluation dataset, compared to 1.4% once they were removed [129]. Other sources have described how flaky tests were a threat to the validity of their techniques, with most opting to filter them as a part of their methodologies [134, 150, 181].

4.3 Other Consequences

Beyond the general consequences for testing already examined in this section, various sources have demonstrated that flaky tests can pose a considerable problem to a diverse range of specific testing activities. The effectiveness of techniques such as mutation testing [108, 157], fault localization [172], automatic test generation [149, 156], batch testing [144], and automatic program repair [136, 179] have all been shown to be impacted by the presence of test flakiness. Furthermore, in the domain of software engineering teaching, studies have revealed that flaky tests can detract from the educational value of various teaching activities and possibly become a burden for students [153, 158, 164].

4.3.1 Mutation Testing. Mutation testing is the practice of assessing a test suite's bug finding capability by generating many versions of a software under test with small syntactical perturbations, to resemble bugs, known as *mutants* [107]. A test is said to *kill* a mutant if it fails when executed upon it, thus witnessing the artificial bug. If a test covers mutated code but does not kill it, then the mutant is said to have *survived*. At the end of a test suite run, the percentage of killed mutants is calculated and is called a *mutation score*, commonly used as a measurement of test suite quality. Demonstrating the extent to which flakiness can impact mutation testing, Shi et al. [157] conducted an experiment with 30 Java projects. In the context of this study, the flakiness they were specifically referring to relates to non-determinism in the coverage of tests rather than necessarily the test outcome. As an initial motivating study, they examined the number of statements that were inconsistently covered across their subjects when repeatedly executing their test suites. Overall, they found that the coverage of 22% of statements was non-deterministic. They used a mutation testing framework, called PIT [27], to generate mutants for these same subjects. The framework first runs the test suite to analyse each test's coverage so that it generates mutants that the tests have a change of killing. Given the level of flakiness in the coverage of these tests, however, the authors explained that when they are executed again to measure mutant killing, they may not even cover their mutants, such that their killed status is unknown. Investigating this phenomenon's extent by applying PIT to their subject set, they found that over 5% of mutants ended up with an unknown status due to inconsistent coverage, leading to uncertainty in the mutation score. Assuming that all unknown mutants were killed, the overall mutation score would have been 82% and—further assuming that they all survived—this score would have been 78%, showing that non-deterministic coverage can limit the reliability of this test effectiveness metric.

4.3.2 Fault Localization. Vancsics et al. [172] studied the influence of flaky tests on automated fault localization techniques. Fault localization uses the outcomes and coverage of tests to identify the location of faulty program elements. This is motivated by the reasoning that a group of program statements, for example, that is disproportionately covered by failing tests is likely to contain a bug. Fault localization techniques associate program elements with *suspiciousness rankings*, calculated in various ways [178], which capture the probability that they contain a fault. As the subjects of their investigation, they took multiple versions of a single project from the *Defects4J* dataset, each version containing a reported bug, and evaluated the effectiveness of three popular fault localization techniques in locating each respective bug. They executed the test suites of each buggy version 100 times, collecting test outcomes and method-level coverage data. To simulate the presence of flaky tests, they repeatedly invoked each studied technique to compute suspiciousness rankings for each method, increasingly adding noise to the test outcomes. Specifically, they artificially changed test outcomes from pass to fail, or vice versa, at random with an increasing probability, up until the point where each test appeared to pass or fail with a 50/50 ratio. This required them to modify the suspiciousness ranking formulae of each technique to consider a

test outcome as a probability of passing or failing rather than a binary value, since typically fault localization does not require multiple test suite runs and would thus only have one recorded outcome for each test. To measure the impact of the artificial flaky tests, they measured the extent to which the suspiciousness rankings changed with increasing flakiness. Their results indicated that, in general, each technique was affected by increasing flakiness. In other words, the magnitude of the difference in the suspiciousness rankings of each technique for many of the subject bugs was positively correlated with the magnitude of the artificial flakiness.

4.3.3 Automatic Test Generation. Shamshiri et al. [156] studied the effectiveness of automatically generated test suites for the testing of Java programs. They applied three unit test generation tools, Randoop [29], EvoSuite [8], and AgitarOne [2], to a dataset of over 300 faults across five open-source projects, assessing how many bugs the automatically generated unit tests could detect. Through this process they also identified the number of flaky tests that were generated by each of the three tools. Of the tests generated by Randoop, which uses feedback-directed random test generation, an average of 21% exhibited non-determinism in their outcomes. The EvoSuite tool, which leverages a genetic algorithm, produced flaky tests at an average rate of 3%. Only 1% of the tests generated by the commercial, proprietary tool called AgitarOne were found to be flaky. These findings demonstrate that automated tools, as well as developers, are also capable of producing flaky tests, threatening the overall reliability of automatically generated test suites.

A later study by Paydar et al. [149] examined the prevalence of flaky tests within the regression test suites generated specifically by Randoop. They explained how regression test suites are useful in as far as they capture the behavior of a program at a given point in time and are used to identify if a recent code change has any unintended effects. If a regression test suite contains flaky tests, then it does not accurately reflect the behavior of the program and is thus less informative for developers. These authors took between 11 and 20 versions of five open-source Java projects and used Randoop to generate regression test suites, which were the main objects of analysis. Overall, they found that 5% of their automatically generated test classes were flaky, and on average, 54% of the test cases within each of these were flaky, further demonstrating that automatically generated tests can contribute to the flakiness of a test suite.

4.3.4 Batch Testing. Najafi et al. [144] investigated the impact of flakiness on batch testing, a technique for improving efficiency within a continuous integration pipeline. Instead of testing each new commit individually, batch testing groups commits together to reduce test execution costs. Should the batch pass then each commit can proceed to the next stage of the pipeline. Should a batch fail, it has to be repeatedly bisected to identify the commit(s) that caused the failure, effectively performing a binary search for the culprit. They explained that flaky tests are a threat to the applicability of batch testing, since a spurious flaky failure may lead to unnecessary bisections. To mitigate this, smaller batch sizes can be selected, since the flaky failure will affect fewer commits, though this limits the potential efficiency gains from applying batch testing in the first place. An evaluation upon three unnamed projects at Ericsson demonstrated the following negative correlation: the more flakiness within a test suite, the smaller the most cost-effective batch size.

4.3.5 Automatic Program Repair. Test suite-based automatic program repair is a family of techniques for automatically generating patches for bugs [127]. The test suites of the subject programs are used to assess the correctness of the patch, such that if all the tests pass then the patch is deemed suitable. Like any technique based on the outcomes of tests, automatic program repair is sensitive to flaky tests, since they introduce unreliability into this assessment. Specifically, a flaky test failure could lead to a correct patch being spuriously rejected [136]. Ye et al. [179] presented an empirical evaluation of automated patch assessment, analyzing 638 automatically generated

Table 8. Summary of the Findings and Implications Answering RQ3: What Insights and Techniques Can Be Applied to Detect Flaky Tests?

	Finding	Implications	Source
🎓	Almost all flaky tests were **independent of the execution platform**, meaning, for example, that they could be flaky under multiple operating systems—even if they were dependent on the execution environment.	Techniques for detecting flaky tests ought to **consider environmental dependence with a higher priority** than platform dependence.	[133]
🎓	Of the *asynchronous wait* flaky tests, 34% used a simple time delay to **enforce a particular execution order**.	A considerable portion of asynchronous wait flaky tests may be manifested by changing—in particular by decreasing—a **simple time delay**.	[133]
🎓	The vast majority of flaky tests of the *concurrency* category involved, or could be reduced to, only **two interacting threads**, and 97% pertained to concurrent access to in-memory resources only.	Previous approaches to **increasing context probability** [80] may be applicable to detecting concurrency related flaky tests.	[133]
🎓	Dependency on **external resources** was blamed for 47% of flaky tests of the *test order dependency* category.	Not all order-dependent tests can be detected by considering the internal state of in-memory objects; **modelling of the external environment** may be required.	[133]
🎓 </>	**Repeating tests** as a method of identifying flakiness is a common practice. Between 15% and 18% of test methods were found to be flaky in open-source Java projects using this approach.	Repeating tests can be considered a **reliable baseline** for detecting flaky tests, since it directly witnesses inconsistent outcomes. It can also become very costly in terms of **execution time** when performing many repeats.	[69, 121, 124]
</>	One study found that 88% of flaky tests were found to consecutively fail up to a **maximum of five times** before passing, though another reported finding **new flaky tests even after 10,000 test suite runs**.	There appears to be **no clear, optimum number of reruns** for identifying flaky tests.	[62, 124]
🎓 </>	By identifying tests that **cover unchanged code** but whose outcome changes anyway, DeFlaker was able to detect 96% of flaky tests previously identified by repeatedly executing them up to 15 times.	By taking differential coverage into account, an automated tool can identify the vast majority of flaky tests with only a **single test suite run**.	[69]
</>	By **randomizing the implementations** of non-deterministic specifications, NonDex found 60 flaky tests across 21 open-source Java projects.	Developers should take care when dealing with **unspecified behavior**, such as the iteration order of unordered collections, so as not to introduce flaky tests.	[97, 158]
</>	The addition of **CPU and memory stress** during test suite reruns was shown to increase the rate at which flaky tests were detected.	Since stress-loading tools are readily available, this technique is easily accessible to developers and could **reduce the run-time cost** of detecting flaky tests.	[162]
🎓	By fitting a probability distribution over the inputs evaluated in assertion statements and estimating the probability that they would fail, FLASH identified 11 new flaky tests in **machine learning projects** and verified a further 11 previously fixed flaky tests.	Despite the **differences in the common causes** of flaky tests in machine learning projects (see Section 3.3), specific approaches for detecting them have demonstrated some success in this domain.	[78]
🎓	Through **static pattern matching** within test code, 807 instances of timing dependencies potentially indicative of flaky tests of the asynchronous wait category were identified across 12 projects. A sample of 31 of these were all identified as true positives via manual analysis.	This technique could prove useful as a **first indicator** of potential flakiness, given that it requires no test executions, but is naturally limited by the fact that it cannot verify that the tests it identifies are genuinely flaky.	[175]

(Continued)

Table 8. Continued

	Finding	Implications	Source
🎓 </>	The application of **machine learning techniques** to the detection of flaky tests has shown promising results, with one study reporting an overall F1 score of 0.86 across 24 open-source Java projects using a model based on features regarding particular identifiers in test code, the presence of test smells and other test characteristics.	Since these techniques **do not require repeated test suite runs** and are **not tied to a specific programming language**, they are potentially very useful to developers, who may welcome a more general-purpose technique and may not have the time or resources to repeatedly rerun their test suites.	[62, 70, 99, 150]
🎓 </>	Of 245 flaky tests, 75% were flaky from the **commit that introduced them**.	Running automatic flaky tests detection tools only after introducing tests may **identify the majority of cases**, but will exclude a significant portion.	[125]
🎓 </>	Of 61 flaky tests that were not flaky from their inception, the median number of commits between the commit that introduced them into the test suite and the **commit that introduced their flakiness** was 144.	Periodically running flaky tests detection tools after larger numbers of commits, i.e., 150 or so, is likely to achieve a **good cost-to-detection ratio**, as opposed to running them after every commit, which may be prohibitively expensive.	[125]
🎓 </>	**Executing a test suite in reverse** was able to manifest 72% of all the developer-written and automatically generated order-dependent tests that were identified as part of a larger evaluation of several strategies within DTDetector. This approach was also the fastest by at least one order of magnitude.	Executing tests in their reverse order and identifying order-dependent tests via inconsistent outcomes compared to their original order is a fast and reasonably **effective baseline approach**.	[185]
🎓	By monitoring reads and writes to Java objects between test runs, ElectricTest detected all the same order-dependent tests in a single instrumented test suite run as DTDetector would when it repeatedly executes the tests. It also detected hundreds more that were **not verified**.	Techniques of **instrumenting objects** to identify potential test order dependencies can be very efficient, since they require only a single test suite run. Yet, they cannot verify that a given order-dependent test is manifest and so could be considered prone to **false positives** and a low precision, even if they have a high recall.	[68]
🎓	By executing minimal test schedules to **verify possible order-dependent tests**, PraDeT filters false positives detected using an object instrumentation technique similar to ElectricTest. However, it detected fewer order-dependent tests overall than simple techniques based upon reversing or shuffling the test run order.	Despite their simplicity, techniques for detecting order-dependent tests based upon re-executing whole test suites in different orders may be **more efficient and effective** than more sophisticated approaches.	[88]
🎓	Filtering possible order dependencies that were **unlikely to be manifest**, using natural language processing techniques upon the leading verb and nouns of test case names, decreased the run time of TEDD by between 28% and 70% when detecting order-dependent tests.	There appears to be valuable information regarding the existence of test order dependencies present in the **names of test cases**, which may be extracted using natural language processing techniques.	[71]
🎓	An evaluation of iDFlakies found that, of 422 identified flaky tests, **50% were order-dependent**.	The prevalence of order-dependent tests as reported by automatic tools appears to be higher than as reported by developers (see Table 5), suggesting that developers may be **unaware of many order-dependent tests** or possibly do not consider them a priority for repair.	[122]

(Continued)

Table 8. Continued

	Finding	Implications	Source
🎓	A chi-squared statistical test of independence revealed that 70 of 96 flaky tests had **different failure rates** when executed in different test class orders, to a statistically significant degree.	The **binary distinction** of flaky tests into order-dependent or not **may be overly coarse**, since the probability that a flaky test fails may be dependent on the test run order, yet still be neither exactly zero or one.	[124]

Findings relevant to researchers are marked with 🎓. Findings relevant to developers are marked with </>.

patches. Specifically, they employed a technique known as Random testing based on Ground Truth (RGT) to determine if a generated patch was correct or *overfitting*. A patch is considered overfitting if it passes the developer-written tests it was generated from, yet is generally a poor solution to the bug in question and may fail on tests held out from the patch generation process. To that end, the RGT technique uses an automatic test generation technique such as Randoop [149] or EvoSuite [82, 85] to assess the generated patches. If a patch fails an automatically generated test, then it is labelled as overfitting. In this case, the test is also applied to a developer-written, *ground truth* patch to determine the behavioral difference with the overfitting patch. Given the potential for automatic test generation techniques to produce flaky tests [149, 156], the RGT technique includes a preprocessing stage where it repeatedly executes each generated test on the ground truth patch. Given that the ground truth patch is considered to be correct, any test failure at this stage is considered to be flaky and the test is discarded. Having discarded 2.2% of tests generated by EvoSuite and 2.4% generated by Randoop for this reason, Ye et al. concluded that flaky test detection is an important consideration for researchers in automatic program repair. The authors remarked how this was a threat to the internal validity of their study, explaining how the flaky tests they discarded may still have exposed behavioral differences between generated and ground truth patches. As such, this could have led to them underestimating the effectiveness of RGT in the context of automated program repair.

4.3.6 Teaching and Education. Using a contrived electronic health record system developed and tested by students of a software engineering course as an object of study, Presler-Marshall et al. [153] analysed the effect of various factors on the flakiness of user interface tests within web apps. As part of their motivation, they explained that, while it may well reflect real industry experiences, ambiguous feedback from flaky tests can introduce undue stress to students.

Shi et al. [158] investigated the impact of assumptions about non-deterministic specifications upon the incidence of flaky tests in both open-source Java projects and student assignments. They described how the student assignments in software engineering courses are typically graded with the assistance of automated tests. If it turns out that these tests are flaky, the authors explained, then incorrect solutions may have passing tests and, visa versa, correct solutions may have failing tests and thus may result in unfair grades being awarded. They went on to evaluate the incidence of flaky tests in student submissions of a software engineering assignment, where students were asked to implement a program and its test suite, and found 110 flaky tests across 89 submissions, with 34 containing at least one flaky test.

A study by Stahlbauer et al. [164] introduced a formal testing framework for the educational programming environment called Scratch [30]. The authors instantiated this framework with the Whisker tool, providing automatic and property-based testing for Scratch programs. Since Scratch is an educational platform, they reasoned that it would add educational value by assisting learners to identify functionality issues in their programs. Upon evaluation of their tool, they found that just over 4% of the combinations of tests and projects to exhibit flakiness if Scratch's

underlying random number generator was not seeded. This result suggests that flaky tests have the potential to confound their framework and thus potentially detract from its intended educational value.

Conclusion for RQ2.3: What are the other consequences of flakiness? Tests with flaky coverage have been shown to be deleterious to the effectiveness of mutation testing, with one experiment finding that 5% of mutants ended up with an unknown killed status, resulting in uncertainty around the final mutation score [157]. One source has demonstrated that fault localization may be sensitive to the inconsistent outcomes of flaky tests, finding the applicability of three popular techniques to be negatively correlated with the magnitude of simulated flakiness [172]. Automatic test generation was shown to have the potential of producing flaky tests, in particular the Java tool Randoop, where one study found 21% of the overall tests it produced across five subjects to be flaky [156]. The potential efficiency benefits of the batch testing technique was also shown to be negatively affected by test flakiness, as evaluated with three proprietary subjects [144]. As another technique dependent on the outcomes of tests, one study evidenced the negative impacts of flaky tests on test suite-based automatic program repair, specifically the automatic assessment of the generated patches [179]. Studies have also found that flaky tests can be identified within students' submissions of software engineering assignments, potentially impacting grading [158]. Their authors' suggested that they can become an obstacle to learning [164] and also place undue stress on students [153].

5 DETECTION

Armed with knowledge of their underlying causes, as covered in Section 3, and motivated by their negative impacts, as described in Section 4, we now consult sources presenting insights and strategies for detecting flaky tests. The automatic identification of flakiness in a test suite has the potential to be useful for developers by not only highlighting tests that may need to be rewritten but also by enabling the use of various mitigation strategies (see Section 6.1.1). Along a similar vein as Section 3, we separately examine techniques targeting order-dependent tests. Given their unique costs, they have been afforded special attention with a line of research specifically examining their detection. We answer **RQ3** by following the same pattern as we did in previous sections, posing and answering two sub-research questions, with Table 8 providing a summary.

- **RQ3.1: What techniques and insights are available for detecting flaky tests?** Answered in Section 5.1, this question aims to capture the findings and approaches regarding the detection of flaky tests. Our answer provides the reader with a tour of the relevant literature and demonstrates the diversity of techniques available, ranging from those that require executions of a test suite to those that are entirely static in their analyses. Beyond presenting specific automatic techniques, general approaches for flaky test detection are also examined, as well as offering advice in regards to how best to apply automated tools.
- **RQ3.2: What techniques and insights are available for detecting order-dependent tests?** Answered in Section 5.2, the aim of this question is to demonstrate the range and extent of efforts to specifically identify test order dependencies, motivated by the specific impacts and causes of this particular category of flaky tests. In doing so, we provide a comparative survey of such techniques and demonstrate an emerging line of work.

5.1 Detecting Flaky Tests

Studies have explored the many different ways of identifying flaky tests within test suites. One paper offered insights on how best to manifest flaky tests derived from inspecting previous

repairs of flaky tests within the commit histories of a multi-language sample of projects [133]. With regards to automatic techniques, a common baseline approach is to repeatedly execute the tests in an attempt to identify inconsistencies in their outcomes [69, 121, 124, 139]. More targeted approaches have considered test coverage [69] or the characteristics of specific categories of flakiness [78, 158, 162]. Several references have presented approaches that do not explicitly require test reruns, such as those based on pattern matching [175] or machine learning [62, 70, 114, 151]. Given the potentially prohibitive costs of repeatedly applying automatic flaky test detection tools, one reference experimentally examined when it was best to use them to achieve a good ratio of detection to run time cost [125].

5.1.1 Insights from Previous Repairs. By studying historical commits that repaired flaky tests in open-source projects of the Apache Software Foundation, Luo et al. [133] offered several insights into how best to manifest test flakiness. They found that 96% of the flaky tests they examined were independent of the execution platform, meaning that they could be flaky on different operating systems or hardware, even if they were dependent on a component in the execution environment such as the file system. From this finding, they suggested that techniques for detecting flaky tests ought to consider environmental dependence with a higher priority than platform dependence. Of the flaky tests they categorized as being of the *asynchronous wait* category, they found that 34% used a simple time delay or a `sleep` or `waitFor` method [18, 19], to enforce a certain ordering in the test execution. They explained that these particular cases of flakiness may be manifested by changing, specifically decreasing, this time delay. They suggested a second manifestation approach for tests in this category: adding an additional time delay somewhere in the code. This technique, they explained, could be applicable to the 85% of sampled flaky tests in this category that neither depended upon external resources (since they are harder to control) and involved only a single ordering (one thread or process waiting on one other thread or process). On this theme, Endo et al. [81] proposed a technique for manifesting race conditions in JavaScript applications, which are typically highly asynchronous, by selectively introducing delays between events. Following an empirical study, they found their technique was also capable of manifesting flaky tests of the *asynchronous wait* category, identifying two such instances among 159 test cases. Of the flaky tests Luo et al. categorized as being of the *concurrency* category, they found that almost all involved, or could be simplified to, two interacting threads and that 97% were related to concurrent access to in-memory objects, as opposed to being related to external resources such as the file system. This, the authors posited, implies that existing techniques [80] for increasing context switch probability could manifest this kind of test flakiness. With regards to test order dependencies, the authors found that 47% of the flaky tests that they categorized under this cause were facilitated by external resources and thus recording and comparing internal object states may be insufficient to detect these instances, instead requiring modeling of the external environment or reruns with different test run orders.

5.1.2 Repeating Tests. The most straightforward approach to detecting flaky tests is to repeatedly execute the tests of a test suite. This is done with the rationale that, if re-executed enough times, the inconsistent outcomes of any flaky tests will be manifested. This approach has been universally adopted, becoming a part of the testing infrastructure within large software companies such as Google [139] and Microsoft [121]. Furthermore, plugins and extensions are available for popular testing frameworks, such as Surefire [34] for Maven [23] and flaky [9] for PyTest [28], for repeatedly executing failing tests to identify flakiness. Due to its ubiquity and simplicity, the repeated execution of tests may be considered a baseline approach from which to evaluate more sophisticated methods for detecting flakiness [69].

Bell et al. [69] assessed how many flaky tests could be identified in open-source Java projects by repeating the failing tests until they either passed, thus indicating a flaky test via an inconsistent outcome, or until a limit was reached. Reasoning that repeatedly executing a test case using the same strategy may result in it failing for the same reason each time and thus not exposing a flaky outcome, they used an incremental approach combining three rerunning strategies. They first repeated a failing test up to five times within the same Java Virtual Machine (JVM) process, followed by up to five times within separate processes and then again up to five times, cleaning the execution environment and rebooting the machine between repeats. The authors applied this approach to 28,068 test methods across 26 projects and found 18% of the test cases in their dataset to be flaky. Of these flaky tests, 23% were manifested by re-execution within the same process, a further 60% required repeats within their own JVM instances and the remainder could only be identified by removing generated files and directories (by using Maven's `clean` command [23]) and then rebooting the machine.

The decision of how many times to repeat a test is a difficult one, since a low number may risk missing the more elusive flaky tests and a high number can impose significant runtime costs. For example, at Google, failing tests may be repeated up to ten times to identify if they are flaky [139]. Whereas Microsoft's Flakes system repeats failing tests only once by default [121]. Lam et al. [124] set out to identify what would be a good number of repeats by examining the lengths of sequences of consecutive failures when repeatedly executing flaky tests, which they referred to as *burst lengths*. Initially, they re-executed 4,000 times, in various orders, the test suites of 26 modules of open-source Java projects, consisting of 7,129 test methods, to identify the flaky tests. They identified 107 flaky tests this way and went on to examine the cumulative distribution function of their *maximal* failure "burst lengths." The authors found that around 88% of the flaky tests in their study would fail up to five times consecutively before passing and revealing flakiness. This led them to suggest that no more than five repeats should be necessary to manifest the vast majority of flaky tests.

Kowalczyk et al. [116] presented a mathematical model for ranking flaky tests by their severity, based on their outcomes following repeated executions. Their model includes the concept of a *version*, a series of test runs where the source code, program data, configuration, and any other artifact that may be expected to affect test outcomes, remains unchanged. Within a single version, the authors described two flakiness measures. The first is the *entropy* of the test, a value between 0 and 1 inclusive where 0 indicates a test that always passes or always fails and 1 indicates a test that passes half the time and fails the other half. For a test with a probability of passing in a given version p, this is calculated as $-p \log_2 p - (1 - p) \log_2 (1 - p)$. Their second measure, the *flip rate*, captures the rate at which the test transitions between passing and failing, or vice versa, and is calculated simply as the ratio of the number of such transitions over the maximum number possible, which would simply be one less than the number of repeats. To aggregate these scores for a test across versions, the authors propose an *unweighted model*—simply the arithmetic mean of the score across some number of previous versions—and a *weighted model*—an exponentially weighted, moving average that gives more weight to more recent versions. Leveraging data from two services at Apple, the authors also explain how to use this generalizable model for test flakiness to both identify flaky tests and rank them according to their severity.

5.1.3 Differential Coverage. Bell et al. [69] presented an approach for detecting flaky tests in Java projects without having to repeatedly execute them. Their technique is based on the notion that if a test previously passed and now fails, and does not cover any code that was recently changed, then the failure must be spurious and thus flaky. To that end, they developed

a hybrid statement and class-level instrumentation technique that considers differential coverage only, meaning the statements that have recently changed according to a version control system, to identify if a given test method does indeed cover changed code or not. They implemented their technique as a tool named DeFlaker and evaluated it using 5,966 commits across 26 open-source Java projects. Initially, they compared their approach to a three-tiered strategy of rerunning failing tests until they passed, thus identifying flakiness, or until an upper limit was reached. They found that DeFlaker was able to identify 96% of the flaky tests identified by rerunning tests in this way, demonstrating its comparable effectiveness with this baseline approach. A further experiment demonstrated that, of 96 flaky tests confirmed as genuine based upon historical fixes from version control data, DeFlaker was able to verify up to 88%. Since DeFlaker is a coverage-based tool with no requirement to repeatedly execute tests, it has the advantage of being significantly more efficient. Examining the cost of DeFlaker's instrumentation, the authors measured the time overhead compared to several popular coverage measurement tools. They found that DeFlaker incurred a mean increase of approximately 5% the un-instrumented run time of the test suite, compared to a range of 33% to 79% for the other tools, concluding that DeFlaker was lightweight enough for continuous use during testing.

5.1.4 Non-deterministic Specifications. Shi et al. [158] presented a technique for identifying test flakiness stemming from the assumption of a deterministic implementation of a non-deterministic specification, or *ADINS*s as they abbreviated them, within Java projects. Such specifications are "underdetermined" and leave certain aspects of behavior undefined and thus up to the implementation, potentially allowing for multiple correct outputs for a single input [96]. They went on to describe three categories of ADINSs. The first was *random* and relates to the case where code assumes that the implementation of a method deterministically returns a particular scalar value when it's not specified to do so. They give the `hashCode` method of the abstract `Object` class [48] as an example of a potential subject of such an ADINS, since it is not specified to return any particular integer value and thus may be highly implementation dependent. The second category was *permute* and describes ADINSs regarding a particular iteration order of a collection type object when it is left unspecified. For example, a test that assumes a `HashMap` [44] will iterate over its elements in a particular order would be guilty of this type of ADINS and could thus be flaky across platforms, where the implementation of `HashMap` may vary. The final category was *extend* and describes the situation where a specification gives the length of a collection object returned by some method as a lower bound and the assumption is made by the developer that the object will *always* be of that length. They gave the `getZoneStrings` of the `DateFormatSymbols` class [43] as an example, since it is specified to return an array of length at least five, which it does in Java 7, but returns one of length seven in Java 8. If a developer writes tests that expect this method to return an array of length exactly five, then they may fail when transitioning from Java 7 to Java 8, becoming a flaky test of the *platform dependency* category in Table 4.

To manifest instances of ADINSs, they developed a tool named NonDex (also the subject of its own paper [97]) that consisted of a re-implementation of a range of methods and classes in the Java standard library, which randomized the non-deterministic elements of their respective specifications. For example, the implementation of `HashMap` was changed such that the iteration order could be randomized each time to deliberately violate and manifest any permute ADINSs. Applying their tool, they were able to identify up to 60 flaky tests across 21 open-source Java projects. Furthermore, they executed their approach upon student submissions of an assignment for a software engineering course that required students to both implement and write tests for a library management application. They identified up to 110 tests with ADINSs that were therefore flaky under their randomized implementations.

Developing the same theme, Mudduluru et al. [142] devised and implemented a type system for verifying determinism in sequential programs. Their Java-targeting implementation consisted of a type checker and several type annotations, including `@NonDet`, `@OrderNonDet`, and `@Det`. Respectively, these indicate a non-deterministic expression, an expression evaluating to a collection type object with a non-deterministic order (e.g., an instance of `HashMap`), and an entirely deterministic expression. Their type checker verifies these manually-specified determinism constraints and was demonstrated to expose 86 determinism bugs across 13 open-source Java projects. While not specifically targeting flaky tests, the authors demonstrated that their type checker was able to identify determinism bugs that NonDex was not, thus suggesting that it may be useful for identifying flaky tests, albeit indirectly.

5.1.5 Noisy Execution Environment. Silva et al. [162] presented a technique, called Shaker, for automatically detecting flaky tests, particularly of the *asynchronous wait* and *concurrency* categories. Their technique uses a stress-loading tool to introduce CPU and memory stress while executing a test suite, with the rationale being that this stress will impact thread timings and interleaving, potentially manifesting more flaky tests than if the test suites were just repeatedly executed as normal. The authors took 11 Android projects and repeatedly executed their test suites 50 times to arrive at an initial dataset of known flaky tests. Finding 75 flaky tests with this method, they split these into a "training" set, containing 35, and a "testing" set, containing 40. They used the training set to search for the set of parameters to the stress-loading tool that manifested the most flaky tests. After identifying the best parameter set, the authors applied Shaker to the remaining 40 flaky tests to identify how many it could identify. They compared this with rerunning the respective test suites without stress, as they did to find the initial set of previously known flaky tests. Their results indicated that Shaker was able to detect flaky tests at a faster rate than the standard rerunning approach. In particular, 26 of the 40 flaky tests failed after just a single run with the Shaker tool.

5.1.6 Machine Learning Applications. Dutta et al. [78] presented their FLASH technique for identifying flaky tests specific to machine learning and probabilistic projects. The reasoning underpinning their approach is that machine learning algorithms are inherently non-deterministic and operate probabilistically, hence their outputs ought to be thought of as probability distributions as opposed to deterministic values. Their technique consists of mining a test suite for approximate assertions, such as those that check that the output is within a certain range or approximately equal to some expected value with some specified tolerance. It then instruments the associated test cases and repeatedly executes them to arrive at a sample of actual values evaluated within each mined approximate assertion. To determine if the sample of values is large enough, it uses the *Geweke diagnostic*, which is satisfied once the mean of the first 10% of samples is not significantly different from the final 50% within some specified threshold. Once FLASH has collected enough samples for each assertion, it fits an *empirical distribution* over each of them, used to calculate the probability of the assertion failing. Under their technique, a test is considered flaky if it has an inconsistent outcome after repeated executions, as per the traditional definition, or if it appears to always pass but contains assertions with a probability of failing above a specified threshold. They evaluated FLASH with 20 open-source machine learning projects written in Python and identified 11 previously unidentified flaky tests, ten of which were confirmed to be flaky by the projects' developers. The authors further validated their technique by demonstrating that it could detect an additional 11 flaky tests that had been previously identified by developers as evidenced within their subjects' version control histories.

5.1.7 Pattern Matching. Waterloo et al. [175] performed static analysis on the test code of 12 open-source Java projects. They derived a set of syntactical code patterns associated with

Table 9. A Comparison of Four Different Studies That Applied Machine Learning to the Detection of Flaky Tests

Study	Model	Features
King et al. [114]	Bayesian network [87]	Static and dynamic
Bertolino et al. [70]	*k*-nearest neighbor [112]	Static only
Pinto et al. [151]	Random forest [161]	Static only
Alshammari et al. [62]	Random forest [161]	Static and dynamic

The Model column gives the main type of machine learning model used in the study. The Features columns gives the type of features that were used to train the model. Static features are those that can be derived without running the test, e.g., number of source lines. Dynamic features require at least one test run, e.g., line coverage.

common bugs in tests, many of which they believed to be indicative of potential test flakiness. Their analysis aimed to identify instances of tests that matched these patterns, which were split into three families. The first was *inter-test* and contained code patterns regarding the relationships between test cases, in other words, instances of violations of the test independence assumption, such as tests that invoke one another and share static fields or data streams. They cited Zhang et al. [185] to support the value of this family of patterns as an indicator of potential test order dependencies. The second was *external* and referred to tests with dependencies on external resources, specifically, test cases with hard-coded time delays for waiting on asynchronous results, those with unchecked dependencies upon the system state (e.g., reading/modifying environment variables) and tests that assume some network resource will be available during their execution. They specifically highlighted the hard-coded time delay pattern as being previously identified as a common cause of test flakiness, citing Luo et al. [133], who also identified network dependency as a possible avenue for flakiness. The third family was *intra-test* and pertained to issues with assertion statements within test cases, such as those that use a serialized version of an entire object, which may contain irrelevant information and thus result in fragile tests that "over check."

Their results indicated a very low incidence of *inter-test* patterns, which was at odds with what they expected given the prevalence of order-dependent tests as previously identified [185], suggesting that either their patterns were not indicative of such tests or that static analysis alone may be insufficient to identify them. As for the *external* family, the authors found many instances of potentially problematic patterns across their subject set. With regards to hard-coded time delays in particular, indicative of potential *asynchronous wait* flaky tests, their technique identified 807 matches in their subject set. Further manual analysis of 31 such matches showed that they were all true positives, that is, genuine cases of hard-coded time delays but not necessarily flaky tests, leading them to reaffirm the value of their static analysis approach for this particular pattern. Given its association with test flakiness [133], this finding suggests that static analysis may be useful for identifying possible flaky tests, which is particularly valuable given its low cost (which is on the order of minutes when run on a commodity laptop) compared to repeatedly re-executing tests to identify flakiness. Naturally, given its static nature, this approach is unable to verify if its matches indicate genuine, manifest flaky tests, and could thus have poor precision even if it exhibits high recall.

5.1.8 Applying Machine Learning to Detection. Using statically identifiable characteristics and the historical execution data of tests, King et al. [114] showed how to use a Bayesian network to classify a test as flaky. A Bayesian network [87] is a directed acyclic graph in which each node represents a probability distribution conditioned on its antecedents. In this case, test flakiness may

be considered a "disease" whose probability is conditional on the observation of a variety of test metrics or "symptoms," as they described it. They selected a multitude of such metrics, covering characteristics such as complexity, implementation coupling, non-determinism, performance and general stability. Examples of concrete metrics used include the number of assertions in a test, the average execution time, and the rate at which a test alternates between passing and failing based on historical records. The authors evaluated their approach within the context of a software company called Ultimate Software, where they gathered training examples from historical instances of flaky tests being identified, quarantined and eventually fixed. After training and evaluating their model on one of Ultimate Software's own products, they reported an overall prediction accuracy of 66%.

Bertolino et al. [70] presented FLAST,[1] a machine learning model for classifying tests as flaky or not based purely on static features. To represent a test case in their model, they used the *bag of words* technique on the tokens within its source code, such as identifiers and keywords. Bag of words is a common representation in the field of natural language processing, where a sample of text is represented as a typically very sparse vector where each element corresponds to the frequency of a particular token [186]. Their model is a *k-nearest-neighbor* classifier [112], which classifies representations of tests cases based on the labels (flaky or non-flaky) of their closest k "neighbor" training examples according to some metric over the vector space (which is *cosine distance* in this case) and a classification threshold that determines the proportion of a test's neighborhood that would have to be flaky to classify it as such. As training examples, they used the test cases from the subject projects of previous studies that had automatically identified flaky tests [69, 122]. This meant that they did not have to label the test cases as flaky or non-flaky themselves. For each of these projects, they individually evaluated the effectiveness of their model. To that end, they split their set of training examples for each project into 10 equal *folds*. For each fold, they trained their model on the other 9 folds and then evaluated its effectiveness on the remaining fold. Specifically, they calculated the *precision* and the *recall* attained by their model and then calculated the mean of these metrics across the 10 folds. Precision is the ratio of true positives (the number of tests correctly labeled as flaky) over all positives (the number of tests labelled as flaky, correctly or incorrectly). Recall is the ratio of true positives over true positives *and* false negatives (the number of flaky tests, labelled correctly or incorrectly). For each project, the authors performed evaluations of their model under four configurations. These were based on the combination of two values for k and the classification threshold. For k, the authors tried the values of 7 and 3. For the classification threshold, the authors tried 0.95 and 0.5. In the 0.95 case, their k-nearest-neighbor classifier would have to find the vast majority of a test's neighborhood as flaky to classify it as such, resulting in much more conservative predictions with respect to the positive case. The configuration that offered the best trade-off between precision and recall was $k = 3$ with a threshold of 0.5, giving a mean precision of 0.67 and a mean recall of 0.55, resulting in an *F1 score* of 0.60. F1 score is the harmonic mean of precision and recall and is intended to offer a fair assessment of a model's accuracy. As a static approach it was very fast, with an average training time of 0.71 s and an average prediction time of 1.03 s across their set of projects.

Pinto et al. [151] performed a related study, investigating the notion of flaky tests having a "vocabulary" of identifiers and keywords that occur disproportionately in flaky tests and may be indicative of them. To investigate this, they trained five common machine learning classifiers to predict flakiness using "vocabulary-based" features derived from the bodies of test cases. Specifically, their features encoded occurrences of whole identifiers and parts thereof (by splitting each word in a camel case identifier), as well as other metrics such as the number of

[1]While their paper was not published until 2021, their tool has been available in prototype form since 2019.

lines of code in the test case. To train these classifiers, they used flaky tests previously identified by DeFlaker [69] as positive examples. As negative examples, they repeatedly executed the test suites of the DeFlaker subject set and selected the same number of tests with consistent outcomes as flaky tests, ensuring they had a balanced dataset. The five types of machine learning classifiers they trained were *random forest*, *decision tree*, *naive Bayes*, *support vector machine*, and *nearest neighbor*. To evaluate these, they split their dataset into 80% for training and 20% for evaluation. Following this, they calculated F1 scores for each classifier. Unlike Bertolino et al., they did not calculate individual scores for each project and then average these scores. Rather, they trained and evaluated each classifier just once, using all the training examples from each project all together. Therefore, it is unknown from their results how the performance of these classifiers varies between projects. Furthermore, projects with more flaky tests, and thus more training examples, would have more impact on the final F1 score than those with fewer flaky tests. Their evaluation reported the F1 scores for each type of classifier, with random forest [161] being the best classifier with a score of 0.95. They also identified the most valuable features for classification in terms of their information gain. Information gain is measured in bits and, in this context, indicates how much information the knowledge of each feature gives toward knowing if a test is flaky or not. The top three features they identified were the occurrences of the tokens `job`, `table` and `id`, suggesting that the presence of these within a test case may be indicative of flakiness.

Studies have attempted to reproduce the findings of Pinto et al. several times in different contexts. Ahmad et al. [61] reproduced their methodology with a set of Python projects, and reported lower precision, recall, and F1 scores for three of the five machine learning classifiers used by Pinto et al.. Haben et al. [99] sought to investigate the effectiveness of Pinto et al.'s vocabulary-based feature approach when using a time-sensitive training and evaluation methodology, that is, to train a model with "present" flaky tests and evaluate it on "future" flaky tests, with respect to some time point or commit. They also evaluated the effectiveness of vocabulary-based features in Python, like Ahmad et al., and went on to consider if identifiers and keywords from the code under test were of any use to flaky test prediction. For their investigation of the time-sensitive methodology, they took six of the 24 projects used in the initial evaluation by Pinto et al., each with at least 30 flaky tests. For each of these six projects, they identified the commit where 80% of the known flaky tests were present, forming the training set, and 20% were yet to be introduced, forming the evaluation set. They then trained and evaluated a random forest classifier individually for each project. Their results showed that, for four of the six projects, the time-sensitive methodology produced poorer results than the "classical" methodology used by Pinto et al.—simply splitting the dataset into 80% for training and 20% for evaluation. Haben et al. argued that the time-sensitive methodology more accurately reflects the expected use case of a flakiness model, since presumably developers would use it to assess test cases as they introduced them. Following this, they adhered to Pinto et al.'s "classical" methodology for nine Python projects. They recorded generally good performance across their Python subject set, with a mean F1 score of 0.80 across each project. From this, they concluded that the vocabulary-based feature approach is generalizable across programming languages. Finally, they investigated whether including features from the code under test when training a vocabulary-based model would improve its performance. Since a vital detail of this approach is that it enables *static* prediction of flaky tests, Haben et al. were hesitant to use actual coverage data to identify the code under test associated with each test case. Instead, they used an information retrieval technique to statically estimate which functions of the code under test each test case was likely to cover, from which they extracted identifier counts. Using both the Java and Python subject sets, they found that including features from the code under test did not improve the performance of the model.

Alshammari et al. [62] proposed, implemented and evaluated FlakeFlagger, a machine learning approach for predicting tests that are likely to be flaky. An initial motivating study demonstrated that, across the test suites of 24 open-source Java projects, flaky tests were still being detected after up to 10,000 reruns. This demonstrated the impracticality of straight-forward rerunning as an approach for detecting flaky tests, highlighting the need for alternative methods. Following a literature review, the authors identified 16 test features that they believed to be potentially good indicators of flaky tests. These consisted of eight boolean features regarding the presence or absence of various test smells [65, 91], such as whether or not the test accesses external resources. The remaining features were numeric, such as the number of lines making up the test case, the number of assertions, and the total line coverage of production code by the test. Their approach for collecting these features for a given test was a hybrid of static and dynamic analysis, since not all features (e.g., line coverage) are attainable from static analysis alone. The authors evaluated a variety of machine learning models, finding random forest to be the most effective. As their training and evaluation procedure, Alshammari et al. used stratified-cross-validation with a 90–10 training–testing split [183]. When training a model, they used the SMOTE oversampling technique [74] to ensure a balanced data set with an equal number of flaky and non-flaky training instances. When evaluating a model, however, they did not apply SMOTE to reflect the real-life environment in which their model would be applied, where non-flaky tests far outnumber flaky tests. The authors went on to compare their approach to that of Pinto et al. described previously. Alshammari et al. noted that Pinto et al. *evaluated* their model using a balanced sampling approach, which may have led to an overestimation of their model's effectiveness. To that end, they evaluated both FlakeFlagger and Pinto et al.'s vocabulary-based feature approach under their training and evaluation methodology, as well as a hybrid approach combining the feature set of FlakeFlagger with vocabulary-based features. Unlike Pinto et al., Alshammari et al. presented their F1 scores for each individual project, and presented an overall average of these, ensuring each project had the same degree of influence on the final score. Their results showed that FlakeFlagger alone had an average F1 score of 0.66, the vocabulary-based approach had an average F1 score of 0.19, and the combination of the two feature sets resulted in an average F1 score of 0.86.

5.1.9 Applying Automatic Tools. Lam et al. [125] set out to investigate the most effective strategy for applying flaky test detection tools. As objects of study, they considered two tools, `iDFlakies` [122] and `NonDex` [97, 158], and as subjects, they reused the *comprehensive* set of the `iDFlakies` study (see Section 5.2.7). The combination of these two tools enabled them to identify flaky tests that were order-dependent via `iDFlakies` (see Section 5.2.7) and implementation-dependent via `NonDex` (see Section 5.1.4), thus covering a wide range of flaky test categories. Initially, they executed these tools on the commits that were sampled as part of the `iDFlakies` subject set to identify flaky tests. For each flaky test, they then ran the tools on the initial commit that introduced it into the test suite. If they found that it was not flaky on this test introducing commit, then they searched for the first commit where the test was flaky, reasoning that this commit would have introduced the flakiness. To that end, they performed a binary search starting with the test-introducing commit and the `iDFlakies` commit, eventually converging on the flakiness-introducing commit. They identified 684 flaky tests across the `iDFlakies` commits of their subject set, of which they were able to successfully compile and execute the test-introducing commits of 245. Of these, 75% were flaky from their test-introducing commit, the remaining 25% were associated with a flakiness-introducing commit later in their lifetime. Furthermore, they found the median number of commits between the test-introducing commits and the flakiness-introducing commits of these 25% to be 144, representing 154 days of development time. This led them to suggest that running flaky test detectors immediately after tests are introduced and then

periodically every 150 commits or so would achieve a good detection-to-cost ratio, as opposed to running them after every commit, which would be prohibitively expensive for many projects.

Conclusion for RQ3.1: What techniques and insights are available for detecting flaky tests in general? The most straight forward techniques for automatically detecting flaky tests are based on repeatedly executing them [69, 124]. Since this can be very time consuming, more efficient and more targeted approaches have been proposed. One technique makes use of the difference in coverage between consecutive versions of a piece of software to identify flaky tests as those whose outcome changes despite not coving any modified code. An evaluation of this approach found that is was able to identify 96% of the flaky tests identified by repeated test suite runs [69]. One paper evaluated a tool that randomizes the implementations of various Java classes that have non-deterministic specifications with the aim of manifesting implementation-dependent flaky tests [158]. Across 21 open-source Java projects, this technique identified 60 flaky tests. Another study presented an approach targeting flaky tests of the *asynchronous wait* and *concurrency* categories by introducing CPU and memory stress to impact thread timings and interleaving during test suite reruns [162]. An empirical evaluation found that this approach could detect such flaky tests at a faster rate than standard rerunning alone. One paper presented an approach for detecting flaky tests of the *randomness* category, specifically within machine learning projects [78], and demonstrated its effectiveness by detecting 11 previously unknown flaky tests. Techniques from the field of machine learning have been applied to flaky test detection, with several studies considering the presence of particular identifiers in test code, and other general test characteristics, as potential predictors of flakiness [62, 70, 151]. With regards to prediction based on identifiers, also known as the vocabulary-based model [99], three separate studies presented different degrees of effectiveness on the same subject set [62, 99, 151], highlighting the impact of different evaluation methodologies.

5.2 Detecting Order-dependent Tests

Given their particular costs to methods for test suite acceleration, a thread of work has emerged specifically concerned with detecting order-dependent tests. Some authors have proposed methods that, while not detecting order-dependent tests directly, may still be of use to developers for debugging them [98, 106]. For detecting order-dependent tests directly, one early study [143] explained how isolating test cases can identify bugs that are masked by implicit dependencies between other test cases. Since then, more sophisticated and multi-faceted approaches have emerged, often motivated by the desire to mitigate against the unsoundness that order-dependent tests impose upon techniques for test suite prioritization [185] and parallelization [68, 71]. Many of these approaches directly build upon one another, iteratively making improvements or adding additional features [68, 71, 88, 185]. Three of these studies perform evaluations with the developer-written test suites of the same four Java projects [68, 88, 185], the results and analysis costs of which are summarized in Table 10. Other work has presented techniques based on repeating test suites in different orders in a systematic way as a method for identifying order-dependent tests [122, 176].

5.2.1 Brittle Assertions. Huo et al. [106] developed a way to detect *brittle assertions*, assertion statements that are affected by values derived from inputs uncontrolled by the test, and *unused inputs*, inputs that are controlled by the test but that do not affect any assertions. Brittle assertions in particular may create an opportunity for order-dependent tests to arise, since they make the outcome of a test depend upon inputs that it does not control, but that could potentially be set by another test. To detect brittle assertions, they presented a technique based on input tainting. This involves associating "taint marks" to uncontrolled inputs that are propagated across data and

control dependencies to identify if they eventually end up affecting an assertion statement. The technique, targeting Java programs, selects the static and non-final fields of a test's containing class as the initial uncontrolled inputs. For example, consider a test that assumes a particular field is at its default value and makes use of it in an assertion statement. This is an uncontrolled input since the test does not set the default value itself and another source, such as a previously executed test, could have modified it. In an attempt to eliminate false positives, for each input identified as the cause of a brittle assertion, their technique re-executes the respective test and mutates the value of the uncontrolled input. In the case where this does not impact the test outcome, the result is considered a false positive. The authors implemented their approach as the OraclePolish tool, evaluating it with a subject set of over 13,609 tests, within which 164 brittle assertions were detected and verified as true positives. Their approach incurred a run-time cost of between five to 30 times that of a regular test suite run.

5.2.2 Test Pollution. Gyori et al. [98] proposed a technique that specifically identifies tests that leave side-effects, as opposed to the tests that are impacted by them. Such tests may induce order dependency in subsequently executed tests, and thus their detection may assist developers in debugging them. Their approach models the internal memory heap as a multi-rooted graph, with objects, classes and primitive values as nodes and fields as edges. The roots of this graph represent global variables accessible across test runs, that is, the static fields of all the loaded classes in the current execution. Their approach compares the state of the heap graph before the setup phase and after the teardown phase of a test method's execution to identify any changes, or as they termed them, "state pollution." They implemented their technique as a tool named PolDet, which integrates with the popular Java testing framework JUnit [55], complete with various measures for ignoring side-effects upon irrelevant global state and thus avoiding false positives, by ignoring mock classes for example. They leveraged a modified JUnit test runner to invoke their state graph building logic, which is implemented using reflection upon all loaded classes at each "capture point," i.e., before a test's setup method is invoked and after its teardown method. Furthermore, they equipped PolDet with functionality for capturing the state of the file system across test runs to identify file system pollution, another potential avenue for test order dependencies.

The authors evaluated their tool with the test suites of 26 open-source projects, in total comprising over 6,105 test methods. They found that, when it was configured to ignore irrelevant state, PolDet identified 324 heap polluting tests and a further 8 that polluted the file system. To determine if a positive result for heap pollution was true or false, the authors manually inspected each one, labelling them as a true positive if they could write another test whose outcome would depend on whether it was run before or after the reported polluting test (thus, by definition, creating an order-dependent test), or a false positive if they could not. They identified 60% of the positive results as true this way, suggesting that their tool may have some issues with its precision. In terms of efficiency, the authors found that PolDet had an overhead of 4.5 times the usual test run duration, but with significant variance across different projects.

5.2.3 Dependence Aware. Zhang et al. [185] proposed four algorithms for manifesting order-dependent flaky tests in Java test suites by comparing their outcomes when executed as normal to when executed in a different order. The first, *reversal*, simply reverses the test run order. The second, *randomized*, shuffles the test run order. The third, *exhaustive*, executes every k-permutation of the test suite in isolation, that is, in a separate Java Virtual Machine. The fourth, *dependence-aware*, aims to improve the efficiency of the *exhaustive* technique by filtering permutations that are unlikely to reveal a test order dependency, which it does by analyzing the access patterns of *shared resources*, such as global variables and files, across test runs. In

Table 10. The Number of Developer-written Order-dependent Tests as Identified by DTDetector [185], ElectricTest [68], and PraDeT [88] with Their Analysis Costs in Seconds Given in Parentheses

			Order-Dependent Tests (Analysis Cost in Seconds)								
			DTDetector [185]								
			Randomized			Exhaustive		Dep. Aware			
Subject	**Tests**	**Rev.**	n = 10	n = 100	n = 1000	k = 1	k = 2	k = 1	k = 2	ElectricTest [68]	PraDeT [88]
Joda-Time	3875	2 (11)	1 (57)	1 (528)	6 (5538)	2 (1265)	2 (86400)*	2 (291)	2 (86400)*	121 (2122)	8 (46)
XML Security	108	0 (11)	1 (65)	4 (594)	4 (5977)	4 (106)	4 (11927)	4 (93)	4 (3322)	103 (57)	4 (146)
Crystal	75	18 (2)	18 (14)	18 (131)	18 (1304)	17 (166)	18 (7323)	17 (95)	18 (4155)	39 (22)	2 (106)
Synoptic	118	1 (1)	1 (7)	1 (67)	1 (760)	0 (25)	1 (3372)	0 (24)	1 (1797)	117 (34)	4 (14914)
Total	4176	21 (26)	21 (143)	24 (1320)	29 (13579)	23 (1562)	24 (109022)	23 (503)	25 (95674)	380 (2235)	18 (15212)

The figures for analysis cost are not directly comparable between studies, since they used machines of different specifications. For DTDetector, figures reported are those of Zhang et al. [185]. For ElectricTest, the number of order-dependent tests reported are the numbers of tests identified as reading a shared resource previously written to by another test. Since this does not necessarily imply that they are order-dependent, some may be false positives. An asterisk indicates that the execution timed out after 24 h.

the case where $k = 1$, the dependence-aware algorithm performs a test run in the default order to establish their baseline outcomes. Any tests that do not read or write any shared resources during this run are considered unlikely to be involved in a test order dependency and are filtered out. The remaining tests are executed in isolation and if their outcomes differ from the baseline then they are reported as order-dependent. In the case where $k \geq 2$, each test is first executed in isolation to establish the baseline, again monitoring reads and writes to shared resources. When generating permutations, if it is the case that each test does not read any shared resources that are written to by any previous tests, as measured during the baseline isolation run, then the permutation is discarded. To record reads and writes to global variables, their approach uses bytecode instrumentation to monitor accesses of static fields, conservatively considering any read to also be a potential write. To identify test order dependencies facilitated by files, DTDetector installs a custom `SecurityManager` [50] that monitors the files read from and written to by each test.

Zhang et al. evaluated each of these approaches on the developer-written and automatically generated test suites of four open-source projects implemented in the Java programming language. They found, in both cases, that the *reversal* technique manifested the fewest order-dependent tests but was also the cheapest in terms of run-time by a considerable margin. The *randomized* approach with 1,000 repeats manifested the most, but had a relativity significant cost, proving to be three orders of magnitude greater than *reversal*. The *exhaustive* and *dependence-aware* techniques with $k = 1$ had run times comparable to *randomized* with 100 repeats but manifested order-dependent tests. With $k = 2$, both techniques timed-out after 24 h and failed to perform better than the *randomized* approach. These results indicate that executing a test suite in reverse may be an acceptable baseline approach, since it could detect 72% of all identified order-dependent tests and ought to take no longer than a standard test suite run.

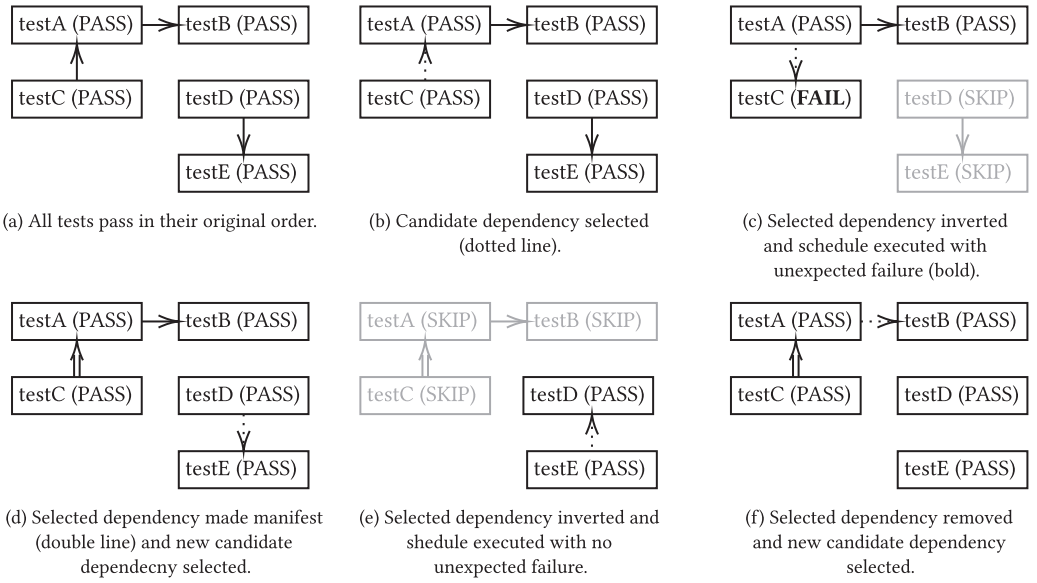
5.2.4 Object Tagging. Bell et al. [68] aimed to develop a technique more efficient than DTDetector [185] for identifying order-dependent tests. Their approach differentiates between two types of dependency relationship. The first, *read-after-write*, refers to the case where `testB` reads a shared resource last written to by `testA`, such that `testB` must be executed after `testA` to preserve the dependency. The second, *write-after-read*, describes the scenario where `testC` also writes to the same resource as `testA`, meaning that `testC` should not be executed between `testA` and `testB`. They implemented their technique as a tool, called ElectricTest, that can identify instances of the two relationships during an instrumented test suite run. The authors explained that recording accesses to static fields would be insufficient to accurately detect all test order

dependencies facilitated by in-memory shared resources. This is because a test could *read* a static field to get a pointer to some object and then *write* to that object's instance fields, with the whole operation being reported as just a read. While DTDetector gets around this problem by conservatively considering all static field reads to also be potential writes, ElectricTest takes a more fine-grained approach. To detect *read-after-write* relationships, it forces a garbage collection pass after each test run and tags any reachable objects as having been written to by the test that was just executed, provided that they've not already been tagged as such by another test. In subsequent test executions, ElectricTest is notified when a tagged object is read from, in which case the reading test is marked as being dependent on the writing test. The tool follows a similar approach for identifying cases of *write-after-read*. This tagging functionality is provided by the *JVM Tooling Interface* [21]. As well as in-memory resources, ElectricTest uses the Java Virtual Machine's built-in `IOTrace` features to detect test order dependencies over external resources such as files.

Evaluating their tool against DTDetector [185], using the same four developer-written test suites, they found that ElectricTest could detect all the same order-dependent tests as DTDetector and many more, indicating that the recall of ElectricTest was at least as good as that of DTDetector. Since ElectricTest requires only a single test suite run, it was found to be up to 310 times faster than the *dependence-aware* mode of DTDetector. The fact that ElectricTest does not verify the dependencies it detects by executing the concerned tests means that its precision may be poorer, since, as Bell et al. noted, the order-dependent tests it identifies may not be *manifest*, in other words, while they may read resources written to by previous tests, their outcomes may not be impacted. One could consider these cases as false positives, since they do not hinder test suite acceleration techniques (see Section 4.2), one of the primary motivations for detecting order-dependent tests. This is a particularly pertinent point given the cost of accommodating order-dependent tests (see Section 6.1.1), which means that too many false positives could become a significant burden. A further evaluation of ElectricTest with the test suites of ten open-source projects not previously used, with an average of 4,069 test methods between them, indicated a mean relative slowdown of 20 times a regular test suite run and an average of 1,720 test methods that wrote to a shared resource that was later read from and 2,609 that read a previously written resource.

5.2.5 Dependency Validation. Building on the work of ElectricTest [68], Gambi et al. [88] presented PraDeT. One limitation of ElectricTest, as previously explained, is that it may identify order-dependent tests that are not *manifest*, meaning that breaking their dependencies by reordering the test suite does not result in a different test outcome. Initially, PraDeT operates in a similar way to ElectricTest, monitoring access patterns of in-memory objects between test executions to identify instances of possible test order dependencies. One improvement of PraDeT over ElectricTest at this stage, as the authors explained, is in its handling of `String` objects and enumerations, respecting their pooled and immutable implementation, which results in fewer false positives.

Modelling a test suite as a graph, with tests as nodes and possible dependencies as directed edges, PraDeT verifies the dependencies it identifies by selecting edges, inverting them as to break the dependency, and generating a corresponding test run schedule using a topological sort on the graph. For efficiency, the schedule does not contain tests that are irrelevant to the selected dependency, that is to say, those that do not belong to the weakly connected component. When executing the schedule, in the case where the dependent test corresponding to the inverted edge does not produce a different outcome, as compared to a regular test suite run, the dependency is considered non-manifest and is removed. The tool follows a *source-first* strategy, selecting edges corresponding to the later executed tests first. Compared to a random approach, this allows initially



(a) All tests pass in their original order.

(b) Candidate dependency selected (dotted line).

(c) Selected dependency inverted and schedule executed with unexpected failure (bold).

(d) Selected dependency made manifest (double line) and new candidate dependecny selected.

(e) Selected dependency inverted and shedule executed with no unexpected failure.

(f) Selected dependency removed and new candidate dependency selected.

Fig. 7. An illustration of several iterations of the dependency verification stage used by PraDeT [88] and TEDD [71]. Test nodes are represented by rectangles and dependency edges by arrows. The tests of the disconnected components with respect to the inverted dependencies in parts (c) and (e) are skipped, since they are deemed irrelevant for verifying the dependency.

impossible schedules, as in the case where inverting a dependency leads to a cycle, to be reliably deferred to a later stage, such as when the cycle is broken by another candidate dependency being removed. Figure 7 gives a visual summary of this verification approach used by tools like PraDeT.

They performed two evaluations of their tool, comparing PraDeT to DTDetector using its *reverse* mode and its *exhaustive* mode with $k = 1$ and $k = 2$. The first evaluation used the developer-written test suites of the same four subjects as used in its initial evaluation of DTDetector by Zhang et al. [185]. Their findings suggested that PraDeT identified more order-dependent tests than any of these modes, although this is at odds with what Zhang et al. reported (and as presented in Table 10), which would indicate that PraDeT actually detected the fewest. The second evaluation used a wider subject set of 15 projects. In this setup, the *exhaustive* mode with $k = 2$ detected the most order-dependent tests by a significant margin, followed by the *reverse* mode, and then by PraDeT. In terms of analysis cost, PraDeT was more than five times faster than the *exhaustive* mode with $k = 2$, though it was about ten times slower than with $k = 1$. Unsurprisingly, the *reverse* mode was the fastest, since it only requires a single test suite run with no isolation overhead.

5.2.6 Web Applications. Biagiola et al. [71] presented an approach, implemented as a Java program named TEDD, for building dependency graphs of tests within the test suites of web applications. In this context, a dependency graph consists of nodes representing individual Selenium test cases and directed edges representing test order dependencies. Previous approaches based upon read/write operations on Java objects [68, 88, 185] are unsuitable in this domain, since web applications are more prone to test order dependencies from persistent data stored on the server-side and implicit shared data via the Document Object Model on the client-side. Their multi-faceted technique consisted of four stages.

In the first stage, the technique extracts the initial dependency graph by iterating through each test case in the test suite's original order and identifying the set of named inputs submitted by each test, such as the values inserted into input fields of a web page via `sendKeys` [31]. Then, for every following test, the set of inputs *used* when evaluating oracles are also identified. When these two sets have an intersection, and at least one test submits an input that another uses, a candidate dependency is added to the dependency graph between the two tests in question. They referred to this technique as *sub-use string analysis graph extraction*. Alternatively, the tool can connect every pairwise combination of tests according to the original test run order, a baseline method they referred to as *original order graph extraction*.

In the second stage, natural language processing techniques applied to the names of test cases are used to filter likely false dependencies. A technique known as *part-of-speech-tagging* [117] is used to identify the verbs and nouns of each name. Semantic analysis on each verb decides the CRUD operation (i.e., *create*, *read*, *update*, or *delete*) to which it is closest. This is considered with respect to the nouns in the name to identify if they suggest dependence. For instance, the names `updateFile` and `readFile` would pass this filtering stage, since they both concern the same noun, `File`, and the verbs suggest a read-after-write dependence. Their approach offers three configurations for this filtering stage: consider only the verb in the name, consider the verb and the direct object it applies to only, or the most comprehensive, consider the verb and all the nouns in the name of the test case.

In the third graph-building stage, the TEDD tool validates the candidate dependencies that survived the filtering stage using a similar inversion approach to that of PraDeT [88]. Given that the filtering stage may not be conservative, TEDD attempts to recover any dependencies that may be missing from the graph. To that end, after executing a schedule with a candidate dependency inverted and witnessing a failure in the corresponding dependent test, when the test was otherwise passing in the original order, TEDD then generates and executes another schedule with the dependency *not* inverted. This is to identify if the failure was due to the inverted dependency or a potentially missing one. In the case were one or more tests unexpectedly fail in the non-inverted schedule, it is assumed that one or more dependencies are missing and need to be recovered. To do so, TEDD connects the first failing test to each of its preceding tests in the original test run order, that were not executed in the non-inverted schedule, as candidate dependencies. In turn, these newly added edges are inverted and tested as the validation stage continues until all candidate dependencies from the previous stages have either been verified or removed.

In the fourth and final stage, missing dependencies are recovered for disconnected components, these are nodes (tests) with no outgoing edges (no dependencies) or those that are fully isolated with no incoming edges (no dependents) either. To that end, all such tests are executed in isolation in separate Docker containers [7]. For each failing test, TEDD connects it as a candidate dependent to every preceding test in the original order. For each passing test that is not an isolated node, having a non-zero in-degree, TEDD executes every schedule containing the test according to the dependency graph's current state. If a test in any of these schedules fails, then TEDD connects it with every preceding test as before. To verify these newly added candidate dependencies, it re-executes the third graph building stage.

The authors went on to evaluate TEDD using six open-source subjects. They considered every combination of both the original order and the sub-use string analysis graph extraction methods along with four initial filtering modes, no filtering plus the three granularities of natural language processing techniques, for a total of eight (2×4) overall configurations. In terms of effectiveness, they found that every configuration found roughly the same number of order-dependent tests, between 32 and 34. In terms of performance, they found that the combination of sub-use string analysis with filtering considering every noun to be the fastest. Analyzing filtering methods

specifically, in the case of original order extraction, the verb only, direct object only and all nouns filtering methods achieved run time savings of 27%, 67%, and 70%, respectively, over no filtering. For sub-use string analysis, these were −69% (i.e., in this case the filtering increased the total run time), 17% and 28%, respectively. These findings indicate that the most comprehensive natural language filtering technique of examining the verb and all the nouns in the names of tests was the most effective.

5.2.7 Repeating Failing Orders. Lam et al. [122] presented iDFlakies, a framework and tool for detecting flaky tests in Java projects. This framework can classify a flaky test as being the result of a test order dependency or not. Their approach involves repeatedly rerunning the test suite in a modified order, the type and granularity of which can be specified by the user, that is, reversed or randomized at the class-level, method-level or both. Initially, the test suite is repeatedly executed to determine which tests pass in their original order. Following this stage, the test suite is repeatedly executed in a modified order. Upon a new test failure, the test suite is re-executed up to and including the failing test both in the modified order that witnessed the failure and in the original test suite order where the test was previously passing consistently. When the test fails in the modified failing order but passes in the original order, it is classified as an order-dependent (OD) flaky test, otherwise, it is classified as a flaky test that is not order-dependent (NOD).

They took projects from previous work [69] and augmented them with 150 popular projects from GitHub, to arrive at a total of 2,921 modules across 183 projects, which they referred to as their *comprehensive* subject set. They took a further 500 projects from GitHub, disjoint from their *comprehensive* set, to form what they called their *extended* subject set. Using the union of these two subject sets, they evaluated their tool in a configuration that randomizes the test run order at both the class and method granularity (i.e., shuffles the order of test classes and then shuffles the order of test methods within these classes). They identified a total of 213 OD flaky tests and 209 NOD flaky tests across 111 modules of 82 projects. This would suggest that just over 50% of flaky tests were order-dependent, which is at odds with the findings of Table 5. This discrepancy could be due to the existence of many more order-dependent tests than developers are aware of, which makes sense if they are not using automatic tools such as *iDFlakies*. Using only the *comprehensive* set, they then compared the effectiveness of the different configurations of *iDFlakies* at detecting flaky tests. They found that randomizing at both the class and method level, as before, was the clear winner, manifesting 162 OD tests and 74 NOD tests. For comparison, the runner up, randomizing at the class level only, identified 40 OD tests and 53 NOD tests.

Later findings by Lam et al. [124] suggest that classifying tests into OD and NOD may be too coarse. Using a subset of the modules used to evaluate iDFlakies, they calculated the failure rates (i.e., the ratio of failures to total runs) of 96 flaky tests identified by repeatedly executing test suites in different test class orders. A chi-squared test of independence identified that 70 such flaky tests had failure rates that were different across orders to a statistically significant degree. They concluded that such tests failed more often in some orders and less often in others and were therefore likely to be **non-deterministically order-dependent (NDOD)**, in other words, somewhere between OD and NOD.

5.2.8 Tuscan Squares. Wei et al. [176] proposed a technique for generating test run orders that are more likely to manifest order-dependent tests than simply sampling orders at random (such as the randomized mode of iDFlakies [185]). Given that the majority of test-order dependencies depend on only a single other test [159, 185], their technique generates the provably smallest number of test run orders in some instances, such that every pair of tests is covered, that is, executed consecutively both ways. For example, for four test cases, each of the 12 permutations of length 2 can be covered by the following four test run orders: $\{\langle t_1, t_4, t_2, t_3\rangle, \langle t_2, t_1, t_3, t_4\rangle, \langle t_3, t_2, t_4, t_1\rangle,$

$\langle t_4, t_3, t_1, t_2 \rangle\}$. To determine these orders, their technique computes the *Tuscan square* [93] from the field of combinatorics. A Tuscan square for the natural number m is equivalent to a decomposition of the complete graph of m vertices into m Hamiltonian paths. This object contains m rows, each of which is a permutation of the integers $[1...m]$. For every pair of distinct numbers in that same range, there exists a row in the Tuscan square where they occur consecutively. In this context, each number represents a test case and each row represents a test run order.

Having observed that Java test suite runners do not interleave test cases within classes, the authors extended the concept of test pairs to that of *intra-class* and *inter-class* pairs. For a set of test run orders to attain full intra-class pair coverage, for every test class, each pair of test cases must be executed consecutively. For a Java class with n tests, there are $n(n-1)$ intra-class pairs. To attain full inter-class coverage, for every pair of classes, each test case from the first class must be executed consecutively with each test case from the second class. For a test suite with k test classes, there are $2\sum_{1 \leq i < j \leq k} n_i n_j$ inter-class pairs. To avoid generating test run orders that test runners would not execute (due to the interleaving of test classes), and thus potentially detecting false-positive order-dependent tests, the authors devised a randomized algorithm, based on computing Tuscan squares, to ensure all intra-class and inter-class pairs are covered with substantially less cost. They evaluated their algorithm on 121 modules from the same set of Java projects used to evaluate iDFlakies [122], finding that it only required 51.8% of the test case runs that executing every pair of tests exhaustively in isolation would require, a trivial method of ensuring that all test pairs are covered.

Conclusion for RQ3.2: What methods have been developed to detect order-dependent tests? One early study presented a technique for detecting brittle assertions, which the authors reasoned may indirectly detect order-dependent tests [106]. After implementing their technique as a Java tool named OraclePolish, they identified 164 brittle assertions across a subject set of 13,609 test cases. A later study presented PolDet, a tool for detecting tests that modify shared program or environmental state, thus potentially creating test-order dependencies. One issue with their approach was that it was difficult to identify if the tests it detected would genuinely go on to induce order dependency in other tests. With the aim of detecting order-dependent tests directly, another study presented DTDetector, a multi-faceted approach consisting of four modes of varying complexity for detecting order-dependent tests [185]. Following an evaluation upon four open-source Java projects, the authors found that the more complex modes incurred a very high run-time cost and did not perform significantly better than the simpler ones. Following this, two tools emerged that improved upon some of DTDetectors short-comings, these were ElecticTest and PraDeT [68, 88]. Unfortunately, the former of these was prone to false positives and the latter was shown to incur a potentially prohibitive run-time cost. Focusing specifically on web applications, a later study presented TEDD, a multi-stage technique that included the novel application of natural language processing techniques on the names of test cases. Following an evaluation on six open-source projects, the tool was able to identify between 32 and 34 flaky tests depending on its configuration. Based on executing test suites in randomized orders, another study presented iDFlakies, a framework for detecting flaky tests and classifying them as order-dependent or not [122]. Following a large empirical evaluation upon 111 modules of open-source Java projects, the authors identified 422 flaky tests, just over half of which were order-dependent. Finally, a later study presented a more systematic approach to detecting order-dependent tests via rerunning the test suite in different orders, based on the mathematical concept of Tuscan squares [176]. Additionally, the technique was mindful of test run orders that typical test runners would not execute.

Table 11. Summary of the Findings and Implications Answering RQ4: What Insights and Techniques Can Be Applied to Mitigate or Repair Flaky Tests?

	Finding	Implications	Source
</>	**Isolating the execution of tests** in their own processes is an expensive mitigation for order-dependent tests with an average time overhead of 618%. By only **reinitializing the relevant program state** between tests runs, as with VmVm, the same effect can be achieved with an average overhead of 34%.	While full process isolation for each test run ought to eliminate all test order dependencies facilitated by shared in-memory resources, and is straightforward to implement, developers should avoid using it due to its **prohibitive costs**.	[67]
🎓 </>	When order-dependent tests are present and known, scheduling approaches for sound **test suite parallelization** achieved an average speedup of 7 times a non-parallelized test suite run. This is compared to an average speedup of 19 times if order-dependent tests did not have to be considered.	It is possible to achieve sound parallelization in test suites with known order-dependent tests, placing further value upon tools for detecting them (see Section 5.2) beyond highlighting their presence. Yet, test suite parallelization is **most effective when no order dependent tests are present**, so repairing the underlying test order dependencies ultimately may be necessary.	[68]
</>	When **order-dependent tests** are present and not known, certain configurations for test suite parallelization are more accommodating than others. An evaluation found the most sound to witness an average of 2% of tests failing with a speedup of 1.9 times a non-parallelized test suite run. The least sound saw an average of 24% of tests failing and a speedup of 12.7 times.	When order-dependent tests are not known in advance, **reliability may become a trade off for speed** when applying test suite parallelization. Given the poor speedup of the most sound configuration, it may be preferable to apply tools to detect order-dependent tests so they can be specifically mitigated, or better, repaired.	[73]
🎓 </>	An algorithm for re-satisfying known test order dependencies broken by the application of **test suite prioritization, selection or parallelization** reduced the number of failed order-dependent tests in developer-written test suites by between 79% and 100%.	Beyond test suite parallelization, detecting order-dependent tests is **beneficial to the sound application** of other test suite acceleration techniques such as prioritization and selection.	[123]
</>	One study demonstrated that a technique for sorting test failures by their stability in the context of **GUI regression testing** was able to rank the failures witnessing genuine, known bugs over those that were more flaky.	Given the **specific factors** that lead to flakiness in GUI testing (see Section 3.3), developers may find it useful to apply tools that specifically mitigate against them.	[90]
🎓	Repeating and isolating test executions in the mutant killing run of **mutation testing** was shown by one experiment to reduce the number of mutants with an unknown killed status by 79%.	Those using mutation testing to assess the quality of their test suites should consider how **repeating and isolating** tests may improve this method's reliability.	[159]
🎓	Several modifications to EvoSuite, a **test suite generation** tool, reduced the number of generated flaky test methods from an average of 2.43 per class to 0.02.	Given the capability of **automatic test generation** tools to quickly generate many tests, it is important to develop methods for decreasing the potentially significant amount of test flakiness often introduced into automatically created test suites.	[64]
</>	Between 71% to 88% of historical repairs of flaky tests were **exclusively applied to test code**.	By fixing a flaky test, a developer may **identify weakness in the code under test**, as evidenced by the fact that some repairs of flaky tests pertained to source code outside of the test suite.	[79, 121, 133]

(Continued)

Table 11. Continued

	Finding	Implications	Source
</>	Between 57% and 86% of previous fixes of *asynchronous wait* flaky tests and up to 46% of *concurrency* flaky tests involved the addition or modification of an **explicit waiting mechanism** such as `waitFor`.	Developers should prefer waiting mechanisms such as `waitFor` over **fixed time delays** such as via the `sleep` method, which require a developer to **estimate timings** that may be inconsistent across machines.	[79, 133]
🎓	**The nature, the origin, and the context leading to the failure**, of a flaky test were rated by developers as the most important information needed to repair it.	Automatic approaches for assisting developers in repairing flaky tests should **focus on retrieving these pieces of information** as they would likely be the most useful.	[79]
🎓	An approach for identifying the code locations that likely contained the root cause of a flaky test, based upon **comparing passing and failing execution** traces, was considered useful for repairing them in 68% of cases.	Researchers should consider how techniques for **automatically identifying the causes of flaky tests** could be useful for developers.	[188]
🎓	Techniques have emerged for the **automatic repair of order-dependent flaky tests and implement-dependent flaky tests**, with generated fixes submitted to the repositories of open-source projects and accepted by their developers.	Initial techniques for the automatic repair of flaky tests show promising results but further work is required to address **other categories of flakiness**.	[159, 184]

Findings relevant to researchers are marked with 🎓. Findings relevant to developers are marked with </>.

6 MITIGATION AND REPAIR

Having examined techniques for detecting flaky tests, we now turn to approaches for their mitigation and repair. The mitigation strategies we examine attempt to limit the negative impacts of flaky tests without explicitly removing or repairing them. Once again, given their specific costs to test suite acceleration, as explained in Section 4.2, we consult several studies on the mitigation of order-dependent tests. Beyond that, we examine sources on the mitigation of flaky tests with regards to some of the more specific testing-related activities previously identified as being negatively influenced by test flakiness. In the context of repairing flakiness, we present and analyse studies offering advice on repairing specific categories of flaky tests, listed in Table 4, derived from previous fixes. We then go on to examine techniques that may assist developers in fixing their flaky tests and those capable of repairing them automatically [159, 185]. To answer **RQ4**, we address two sub-research questions, with our findings summarized in Table 11.

- **RQ4.1: What methods and insights are available for mitigating against flaky tests?** Answered in Section 6.1, this question addresses the techniques that have been developed to minimize the negative impacts of flaky tests without explicitly removing or repairing them. Our answer considers adjustments to the methodologies of many of the testing activities identified as being negatively impacted by flaky tests.
- **RQ4.2: What methods and insights are available for repairing flaky tests?** Answered in Section 6.2, by addressing this question we provide insights into how developers may eliminate the flakiness in their test suites. Our answer presents common strategies employed by developers and discusses which pieces of information are the most important for repair. We also examine a technique for the automatic repair of order-dependent tests.

6.1 Mitigation

As well as detecting flaky tests, many sources have proposed and evaluated techniques for the mitigation of their negative impacts. Many of these studies specifically examine order-dependent tests with respect to minimizing their costs to test suite acceleration techniques, given the well documented issues that they cause. One such source presented a technique for *unit test virtualization*, a faster alternate to isolating each test case execution in its own process, a technique shown to be particularly costly in terms of execution time [67]. Other studies have proposed revisions and alternatives to various approaches for test suite prioritization, selection and parallelization in the face of order-dependent tests [68, 73, 123]. As well as test order dependencies, other techniques have been examined for reducing the costs of flaky tests upon user interface testing [89, 90], mutation testing [157] and automatic test suite generation [64].

6.1.1 Order-Dependent Tests. Bell et al. [67] presented a lightweight approach for mitigating against order-dependent tests. Initially, they examined the prevalence and the overhead of executing test cases in isolation from one another, by executing each one within their own process. This strategy is used to prevent any state-based side effects of each test case run from impacting later tests, thereby eliminating test order dependencies facilitated by shared in-memory resources. By parsing the build scripts of over 591 open-source Java projects, they found that 240 of them executed their tests with isolation, suggesting that the practice was commonplace. Their results also indicated that there was a positive correlation between the probability that a project used isolated test runs and both its number of tests and its lines of code. This suggests that the developers of more complex projects were more likely to have experienced order-dependent tests. They performed an evaluation with 20 open-source Java projects, where they executed each of their test suites with and without isolation to calculate the time cost of this strategy. They found that, on average, test runs using isolation had a time overhead of 618%, indicating that it is a very expensive mitigation.[2] Having established the significant cost of per-test process isolation, they proposed a lightweight alternative, which they implemented as a tool named VmVm for Java projects. Their high-level approach is to identify which static fields of classes could act a vector for state-based side effects. The classes containing such fields are then dynamically reinitialized, as opposed to reinitializing the entire program state between each test run. Initially, static fields are labelled as "safe" if they hold constant values not derived from other non-constant sources, as identified via static analysis of the source code. Under normal circumstances, the Java Virtual Machine initializes a class (if not yet done so) when it's instantiated, when one of its static methods or fields is accessed or when explicitly requested to do so via reflection. Through bytecode instrumentation, VmVm ensures that all classes containing unsafe static fields are reinitialized under those same conditions repeatedly, thus eliminating any possible side effects that they may propagate between test case executions. An empirical evaluation found that their approach was significantly more efficient than full process isolation, with an average time overhead of only 34%. Furthermore, their results indicated that, under VmVm, test outcomes were the same as when executed with isolation, indicating that their more efficient approach was also just as effective for its intended purpose.

As part of the evaluation of their order-dependent test detection tool, ElectricTest, Bell et al. [68] proposed two approaches for test suite parallelization that respect test order dependencies. Given a test dependency graph, generated by their tool or a similar one such as PraDeT [88], they

[2]In recent years, Nie et al. [146], including Bell, identified and submitted a patch for a bug in Maven [23], considerably reducing the overhead of isolation.

described a *naive* scheduler and a more optimized *greedy* one that both group tests into units that can be safely executed in parallel. The naive scheduler simply traverses the graph and groups tests into chains representing dependencies, where each test appears in only one group containing all the dependencies for every test within it. The greedy scheduler takes into account the execution time of each test as well as their dependency relationships to opportunistically achieve better parallelization while still respecting order dependencies, by allowing some tests to appear in multiple groups. For example, given ten CPUs and ten tests, which all take ten minutes to run, and are all dependent on a single test that takes 30 s to run, the naive scheduler would place each of these tests into a single execution group where the 30 s test would run first, achieving no parallelization. In contrast, the greedy scheduler would create ten groups, each containing the 30 s test followed by one of the ten minute tests. While this duplicates the execution of the faster test ten times, the speedup attained from the parallelization of the ten slower tests, which is now possible given their satisfied dependency, far outweighs the cost. Using ElectricTest to generate the dependency graphs, they evaluated their two schedulers with ten open-source Java projects using a 32 CPU machine. They found that their naive scheduler attained an average speedup of 5 times a non-parallelized test suite run, whereas their greedy scheduler achieved an average speedup of 7 times. For comparison, they measured the speedup they would have achieved had they not had to respect any test order dependencies and found this to be 19 times on average. These findings demonstrate that order-dependent tests can be mitigated to achieve sound test suite parallelization (i.e., not leading to inconsistent outcomes) but at the cost of considerable effectiveness.

Candido et al. [73] offered several insights into applying test suite parallelization to Java test suites with order-dependent tests not necessarily known in advance. They found that forking the Java Virtual Machine process to achieve parallelization and allocating each instance a portion of the test classes to execute in serial was the most robust approach with respect to manifesting the fewest test failures, in tests that were otherwise passing when executed in serial. Under this strategy they observed approximately 2% of tests failing on average within the test suites of 15 open-source projects. This method was also the least effective in terms of speeding up the test run, with Candido et al. reporting an average speedup of only 1.9 times a serial test suite run. This was as opposed to more flexible configurations that parallelized using multiple threads within a single process and executed test methods concurrently, which were found to be considerably unsound. One such strategy resulted in an average of up to 24% of failed tests in the most extreme case. However, this was also the most effective configuration, achieving a much more significant speedup of 12.7 times a non-parallelized test suite run on average. These findings indicate that the soundness and effectiveness of test suite parallelization may be at odds with one another when order-dependent tests are present. This led them to suggest that developers should address the order-dependent tests in their test suites, using a tool such as ElectricTest [68], allowing them to utilize the more powerful parallelization strategies that were less accommodating of test order dependencies.

Lam et al. [123] proposed an algorithm for enhancing the soundness of test suite prioritization, selection and parallelization with respect to test order dependencies, again with the requirement that they are known and specified. As input, their algorithm takes the original test suite, a *modified* test suite, for example, a subset of the original test suite as produced by test selection, and a set of test order dependencies. The set of dependencies consists of a list of pairs of tests where the latter is impacted by the prior execution of the former. These are split into *positive dependencies*, where the first must be run before the second for the second to pass, and *negative dependencies*, where the second would fail if executed after the first. Using previous terminology, the former case represents a state-setter and a brittle and the latter case represents a polluter and a victim [159]. Examples of tools that could produce such a set include many of those examined in Section 5.2.

As output, the algorithm produces an *enhanced* test suite—a test suite as close to the modified test suite as possible while respecting the specified test order dependencies. Initially, the algorithm computes the set of tests that the tests of the modified test suite transitively depend upon from the specified positive dependencies. Iterating through the modified test suite, each test is inserted into to an initially empty enhanced test suite. This is done with the pre- and post-condition that the enhanced suite has all of its dependencies satisfied and the additional post-condition that the test is added to the end of the enhanced suite. To that end, the algorithm uses the previously computed set of dependencies to determine the sequence of tests that are needed to be executed before the test to be inserted into the enhanced suite. This sequence is sorted such that the order of the emergent enhanced suite will be as close to the optimized order of the input modified suite as possible. The dependent tests of this sequence are then iteratively inserted into the enhanced suite in a recursive manner, such that they also have their dependencies resolved (if any). To evaluate their algorithm, they performed prioritization, selection and parallelization upon a range of open-source Java subjects from previous work [122], using DTDetector [185] to identify the dependency pairs. They measured the reduction in failures of order-dependent tests after enhancing both the modified developer-written and automatically generated test suites of their subjects. In the case of prioritization, they observed an 100% reduction in failures for the developer-written test suites (i.e., all tests passed) and a 57% reduction for the automatically generated test suites. For selection, the result was 79% and 100%, respectively, and for parallelization, it was 100% and 66%.

6.1.2 User Interface Testing. Gao et al. [89] set out to investigate which external factors are the most important to control in an attempt to reduce flakiness in system user interface tests, performing an experiment upon five GUI-based Java projects. Specifically, they measured the impact of varying the operating system, the initial starting configuration, the time delay used by the testing framework (used to wait for a user interface to be fully rendered and stable before evaluating oracles) and the Java vendor and version, upon non-deterministic behavior during test suite runs with respect to three application layers. For each layer, they derived metrics to measure the magnitude of the non-determinism across repeated test runs under various configurations of the investigated external factors. The lowest level layer was the *code layer* and refers to the source code of the software under test, where they measured the line coverage. To measure the magnitude of the variation in this layer, they calculated an entropy-based metric to measure the extent to which the line coverage was inconsistent over multiple test suite executions. The next layer was the *behavioral layer*, which pertains to invariants regarding functional data, such as runtime values of variables and function return values, which are mined over a run of a test suite. Once again, they derived an entropy-based metric to measure how inconsistently invariants could be observed when repeating tests. The highest-level layer was the *user interaction layer*, representing the interface state visible to the end user. They used a metric based upon the number of false positives observed from a GUI state oracle generator as a measure of the stability in this layer. From the findings of their experiments, they arrived at three overall guidelines for researchers and developers for promoting the reproducibility of test suite executions. The first was to ensure that the exact configuration of the application and the state of the execution platform when running a test suite is explicitly reported and shared. Referring to how some examples of non-determinism appeared unaffected by controlling the various external factors examined, the second was to run tests multiple times to accommodate these cases. The third was to use application domain information in an attempt to control test suite variability, giving an example of controlling the system time as an environmental factor when testing a time-based application.

In a related study, Gao et al. [90] presented an approach for ranking flaky test failures in the context of GUI regression testing based on their stability. The authors described the process of GUI regression testing as executing a regression test suite upon two versions of a GUI application, which invokes particular events to get the GUI into some particular state, where various properties (e.g., the coordinates and dimensions of elements such as text boxes) are measured and compared to identify discrepancies indicative of bugs. Given the flakiness of such tests, developers may face difficulty when identifying which discrepancies represent genuine faults as opposed to being artifacts of external factors such as window placement or resolution. Under the reasoning that the most unstable properties are the least useful for finding bugs, they described an approach for ranking test failures based upon the entropy of the properties they compare. To calculate the entropy, the test suite must be repeatedly executed to arrive at a decent sample of values for each measured property. They evaluated their approach on three Java GUI applications with known bugs and found that their ranking technique indeed percolated the genuine, known bugs to within the top 30 of all the lists of discrepancies.

6.1.3 Mutation Testing. Shi et al. [157] investigated ways to mitigate tests with inconsistent coverage when performing mutation testing. In particular, they used 30 open-source Java projects to evaluate the effectiveness of various modifications to the Java mutation testing tool called PIT [27]. Specifically, they investigated the impact of repeating and isolating tests during the coverage collection and mutant killing runs of PIT. The purpose of the coverage collection run is to identify the set of program statements each test covers so that PIT can generate mutants that they have a chance of killing. Given the finding that 22% of statements were inconsistently covered in their motivating study, they reasoned that repeating tests improves the reliability of this process, since a single run may not accurately represent which lines a test may cover. As for the motivation behind isolating tests at this stage, they referred to the potential for test order dependencies to impact coverage. They found an insignificant increase in the number of generated mutants when repeating and isolating tests for the considerable increase in the amount of time taken to do so, thus concluding that it may not be worthwhile. They went on to investigate the impact of repeating and isolating tests during the mutant killing run, citing the same motivations as before. In this run, tests are executed to identify which mutants they are able to identify by failing and thus kill. The authors found that repeating and isolating tests at this stage reduced the number of mutants with an unknown killed status (i.e., mutants that a test ought to cover based on the coverage collection run, but at the mutant killing stage, does not) by 79%, demonstrating the benefits of this approach.

6.1.4 Automatic Test Generation. Arcuri et al. [64] extended the search-based test suite generation tool called EvoSuite [85] to improve the coverage and "stability," a term they used to mean the absence of flakiness, of test suites generated for Java classes with environmental dependencies. Their multifaceted approach involved instrumenting the bytecode of generated tests to reset the state of static fields following executions and extending EvoSuite's genetic algorithm to generate test methods capable of writing to the standard input stream to accommodate classes that read user input. They also implemented a virtual file system, by mocking Java classes that perform input and output, that is reset after the execution of each test case. Finally, they mocked methods and classes that provide sources of environmental input or other non-determinism, such as the `Calendar` [41] and `Random` [49] classes, both of which make use of system time. An experiment using 30 classes of the SF100 corpus [85] known to lead to flaky generated tests indicated that the combination of these mitigations reduced the number of flaky generated tests from an average of 2.43 per class to 0.02.

Conclusion for RQ4.1: What methods and insights are available for mitigating against flaky tests? An early study proposed and evaluated a lightweight alternative to mitigating test order dependencies via process isolation, based on dynamically re-initializing modified values following test runs, finding it to introduce an average time overhead of only 34% compared to 618% [67]. In the context of test suite parallelization, one study proposed and evaluated schedulers for making the most of available CPUs when executing test suites with known and specified order-dependent tests, reporting an average speedup of up to 7 times compared to a regular test suite run without breaking any test order dependencies [68]. Another source examined how best to configure concurrent execution when faced with order-dependent tests, with the main finding being that ensuring consistent test outcomes appears to be a trade off with achieving the fastest test runs [73]. One experiment demonstrated the applicability of an algorithm for satisfying broken dependencies following rescheduling by prioritization, selection or parallelization, reducing the number of failed order-dependent tests in developer-written test suites by between 79% and 100% [123]. In the area of user interface testing, one study suggested three guidelines after studying the impact of various factors pertaining to the test execution environment and platform on the flakiness of tests. They recommended that the exact application configuration and state of the execution platform be explicitly reported, that it may be useful to re-execute tests several times to account for unknown or uncontrollable non-determinism, and that information regarding the application domain ought to be taken into account when attempting to control flakiness [89]. A study into mitigating flaky test coverage in mutation testing found that repeating and isolating test runs during the mutant killing phase reduced the number of non-deterministically killed mutants by 79% [157]. Finally, extensions to the automatic test generation tool called EvoSuite [85] were found to significantly reduce the number of flaky tests it generated, from an average of 2.43 per test class to 0.02 [64].

6.2 Repair

With regards to repairing flaky tests, numerous sources have presented insights and strategies to assist developers in removing the flakiness from their test suites, as opposed to simply mitigating it. Some have mined insights from examining previously repaired flaky tests [79, 133, 155]. With the aim of helping developers improve the reliability of their test suites in the shortest amount of time, one study presented a technique for prioritizing flaky tests for repair [174]. Another deployed a survey with the aim of identifying the most important piece of information needed to repair flaky tests, as rated by software developers [79]. A line of research concerned with automatically identifying the root causes of flaky tests, to assist developers in repairing them, has emerged in recent years [120, 169, 188]. Finally, we examine techniques for the automatic repair of order- and implementation-dependent flaky tests and those that are flaky due to external data [83, 159, 184].

6.2.1 Insights from Previous Repairs. Studies have provided insights into repairing various kinds of flaky tests by considering previously applied fixes in general software projects. Luo et al. [133] examined commits that repaired flaky tests within projects from the Apache Software Foundation. Eck et al. [79] surveyed Mozilla developers about flaky tests they'd previously fixed. The prevalence of the different types of repairs as reported by these two studies are summarized in Table 12. In terms of where these were were applied, figures range from between 71% and 88% exclusively test code [79, 121, 133]. The remainder were applied to either the code under test, both the test code and the code under test or elsewhere, such as within configuration files. This finding indicates that the origin of test flakiness may not exclusively be test code, and in the cases where fixes were applied to non-test code, suggests that a flaky test has, possibly indirectly, led a developer to discover and repair a bug in the code under test. This provides an additional

Table 12. Prevalence of Different Types of Repairs for Three Prominent Categories of Flaky Tests According to Luo et al. [133] and Eck et al. [79]

Category	Repair	Luo et al. [133]	Eck et al. [79]
Asynchronous Wait	Add/modify `waitFor/await`	57%	86%
	Add/modify `sleep`	27%	—
	Reorder code	3%	13%
	Other	13%	1%
Concurrency	Add/modify `waitFor/await`	—	46%
	Add lock	31%	21%
	Modify concurrency guard	9%	26%
	Make code deterministic	25%	5%
	Modify assertions	9%	—
	Other	26%	2%
Test order dependency	Setup/teardown state	74%	—
	Remove dependency	16%	100%
	Merge tests	10%	—
	Other	0%	0%

Dashes indicate that the study did not consider that type of repair.

argument against ignoring flaky tests, as previously discussed in Section 4.1. With regards to repairing flaky tests of the *asynchronous wait* category, the most common fix involved introducing a call to Java's `waitFor` [18], or Python's `await` [5], or modifying an existing such call, accounting for between 57% and 86% of previous cases. The `waitFor` method blocks the calling thread until a specific condition is satisfied, or until a time limit is reached. This would be used to explicitly wait until an asynchronous call has fully completed before evaluating assertions, thus eliminating any timing-based flakiness. Similarly, Luo et al. identified fixes regarding the addition or modification of a fixed time delay, such as via a call to Java's `sleep` [19], to make up 27% of historical repairs. Compared to `waitFor`, `sleep` is a less reliable solution, since the specified time delay can only ever be an estimate of the upper limit of the time taken for the asynchronous call to complete in any given test run. To that end, the authors identified that 60% of such repairs increased a time delay, suggesting that the developers believed their estimate to be too low or perhaps too unreliable across different machines. A study by Malm et al. [135] found `sleep`-type delays to generally be more common than `waitFor`, via automatic analysis of seven projects written in the C language. Another solution was to simply reorder the sequence of events such that some useful computation is performed after making an asynchronous call instead of quiescently waiting for it to complete, accounting for between 3% and 13% of cases. Such a change does not actually address the root problem, but may be preferable to simply blocking the calling thread.

For fixing flaky tests of the *concurrency* category, the most common repair, according Eck et al., again pertained to the introduction or modification of a call to `waitFor` or `await`, with a prevalence of 46%. Adding locks to ensure mutual exclusion between threads accounted for between 21% and 31% of fixes for flaky tests in this category. Between 9% and 26% of historical repairs involved modifying concurrency guard conditions, such as the conditions for controlling which threads may enter which regions of code at any one time. Making code more deterministic by eliminating concurrency and enforcing sequential execution made up between 5% and 25% of fixes. An additional type of repair as identified by Luo et al. consisted of modifying test assertions, and nothing else, to account for all possible valid behaviors in the face of the non-determinism permitted by the concurrent program, describing 9% of previous repairs.

Ensuring proper setup and teardown procedures between test runs was the most common fix for order-dependent tests according Luo et al., accounting for 74% of cases. Another fix was to explicitly remove the test order dependency by, for instance, making a copy of the associated

Table 13. Useful Information for Repairing Flaky Tests as Rated on a Likert Scale by Developers with Regards to Importance and Difficulty in Obtaining, as Found by Eck et al. [79]

Information	Importance	Difficulty
Context leading to failure	2.69	1.65
Nature of the flakiness	2.40	1.71
Origin of the flakiness	2.22	1.17
Involved code elements	2.21	1.13
Changes to perform the fix	2.08	1.59
Context leading to passing	1.95	1.20
Commit introducing the flakiness	1.89	0.75
History of the test's flakiness	1.79	0.83

shared resource (e.g., an object or directory), making up between 16% and 100% of repairs. Another repair was to merge tests into a single, independent test by, for example, simply copying the code of one test into another, which described 10% of fixes for order-dependent tests according to Luo et al..

As part of an empirical evaluation into flakiness specific to user interface tests, Romano et al. [155] examined 235 developer-repaired flaky user interface tests across a sample of web and Android applications. They categorized the type of repair applied into five categories. The first category, *delay*, accounted for 27% of cases and was subcategorized into repairs involving the addition or increase of a fixed time delay or those relating to an explicit waiting call, as described previously. The second category, *dependency*, related to repairs regarding an external dependency, such as fixing an incorrectly called API function or changing the version of a library. The authors placed 8% of flaky tests into this category. The final three categories were *refactor test*, *disable features* (specifically disabling animations in the user interface, which was found to be a significant cause of flakiness), and *remove test*. These represented 32%, 2% and 31% of repairs, respectively, the latter category not being a true repair, but simply eliminating the flaky test from the test suite.

6.2.2 Prioritizing Flaky Tests for Repair. Vysali et al. [174] presented GreedyFlake, a technique for prioritizing the fixing of flaky tests based on the number of *flakily covered* program statements that would become *robustly covered* if the flaky test was repaired. They defined a flakily covered statement as one that is only covered by flaky tests and a robustly covered statement as one that is covered by one or more non-flaky tests. Their motivation for this approach was based on the notion that program elements covered exclusively by flaky tests are unlikely to be as well-tested as those covered by tests with more deterministic behavior. They evaluated this technique with the test suites of three large open-source Python projects by comparing how quickly flakily covered program statements would become robustly covered when prioritizing flaky test repairs based on other schemes. Specifically, they compared their approach to random prioritization (i.e., merely shuffling the list of flaky tests to be repaired), prioritization based on total statement coverage, prioritization based on newly covered statements (in a regression testing context) and prioritization based on total number of flakily covered statements. Their evaluation demonstrated that GreedyFlake outperformed all of the other methods for every subject with regards to identifying the best fixing order for reducing the greatest number of flakily covered program statements in the fewest number of flaky test fixes.

6.2.3 Important Information for Repair. From a multi-vocal literature review, Eck et al. [79] identified eight pieces of useful information for fixing flaky tests. They asked developers via an online survey to rate on a Likert scale, for each piece of information, how important they considered it and how difficult they thought it was to obtain. The results of this survey are presented in Table 13. The most import of these, as considered by developers, was the context leading to the failure,

which the developers also identified as the second hardest to obtain. As one developer described: "The most difficult operation is reproducing a flaky test, as sometimes only 1/20 fails." Another respondent explained how the verification of the failing behavior can be even more difficult due to the slowness of tests and the execution environment, among other factors, citing their slow UI tests as an example. The second most important was the nature of the flakiness, or its category, which the developers considered the most difficult to obtain. On this topic, one developer referred to identifying the root cause of a flaky test as "a big challenge," due to the wide array of possible contributing factors such as concurrency issues or cache related problems. The remaining pieces of information in descending order of perceived importance were: the origin of the flakiness (i.e., whether it's rooted in test code of the code under test), the involved code elements, the changes needed to perform the fix, the context leading to the flaky test passing, the commit introducing the flakiness and the history of the test's flakiness (i.e., previous causes and fixes).

6.2.4 Identifying Root Causes to Assist Repair. Having identified the nature and origin of flaky tests to be the most important pieces of information needed for repair [79], several approaches have been developed to assist developers by revealing information about their root causes. One such approach, RootFinder, was presented by Lam et al. [120] and compares attributes of the execution of flaky tests in their passing and failing cases. Their technique leverages the `Torch` framework for instrumenting API method calls. Specifically, the framework replaces method calls with a generated version, which calls the original method, but also provides three callbacks, one just before the call, one just after and one upon a raised exception. As part of RootFinder, they implemented callbacks to measure properties including, but not limited to, the identifiers of the calling process, thread and parent process, the caller API, the timestamp of the call, the return value and any exception that was raised. During repeated test suite runs, the information recorded by these callbacks is stored in separate logs for passing and failing test executions. Using these logs, RootFinder evaluates various predicates, the outcomes of which are kept in *predicate logs*, again with one for passing tests and one for failing. Along with the values of these predicates, the code location of the instrumented method call, the calling thread id and an index that is incremented each time the method is called at the same location is also recorded to provide additional context, referred to as *epochs*. Examples of predicates that RootFinder evaluates include whether the return value at some epoch is the same as all chronologically previous epochs (which is useful for identifying non-determinism), whether a specified amount of time is observed between a particular sequence of calls (which is useful for identifying thread interleavings), and whether the method call took longer than a specified upper limit. RootFinder uses these predicate logs to perform further analysis to identify those that are indicative of flaky tests. Those that are always true in passing cases and always false in failing cases (or vice versa) indicate that a method consistently behaves differently in passing and failing runs, which they posited, may help to explain why a test is flaky by showing how the passing and failing executions differ. They went on to provide examples of methods to instrument and predicates to evaluate that may be useful in identifying examples of several of the categories of flaky tests identified by Luo et al. [133]. For instance, for the *asynchronous wait* category, the predicate should indicate that the asynchronous call always took longer in the failing runs such that some timing assumption did not hold (e.g., a fixed time delay was insufficient to wait for an asynchronous method and thus results in the failing test cases).

Ziftci et al. [188] presented another way to compare passing and failing executions. For a given flaky test, their approach performs some number of instrumented runs, collecting execution traces. For a given failing run of the flaky test, their technique identifies the passing execution with the longest common prefix and extracts the point at which the execution diverges into the failing case. They posited that by inspecting the code location of the divergence between passing and failing

cases, developers might understand why their test is flaky. They implemented their approach as a tool in Google's continuous integration platform, making use of its internal flakiness scoring mechanism to select tests that are flaky, but whose failing cases are not too rare, citing limited resources for repeated executions. To assess the applicability of their tool, they performed several case studies. One such study involved two developers fixing 83 flaky tests, that had previously been repaired by other developers, with the assistance of the tool. The developers reported that, in between 36% and 43% of cases, the divergent lines reported by the tool contained the exact code location that required repair. For between 25% and 32% of cases, the fix was applied to a different region of code, but the report was still helpful and relevant. In 15% of cases, the report was inconclusive, hard to understand or not useful.

Terragni et al. [169] proposed a new technique for identifying the root causes of flaky tests, motivated by two main limitations of RootFinder [120]. First, the `Torch`-based instrumentation incurs some amount of runtime overhead, which may be problematic for several reasons, such as potentially impacting the manifestation of time-sensitive flaky tests and increasing the overall analysis times. Second, a general limitation of any strategy based upon rerunning tests, is that it *passively* explores the non-deterministic space of test executions. In other words, repeatedly executing a test under the exact same conditions means that the more elusive flaky tests that are manifested only in rarely occurring scenarios, such as a particular execution order of multiple threads, are unlikely to be identified without a significant number of repeats. This led the authors to propose their technique, which they described as *actively* exploring this space. Specifically, their approach executes some test under analysis repeatedly in separate containers (i.e., via Docker [7]), each designed to manifest a particular category of flakiness. For example, one container attempts to manifest flaky tests of the *concurrency* category by varying the number of available CPU cores and randomly spawning threads that execute dummy operations to occupy the available resources. Another container attempts to identify order-dependent tests by arbitrarily executing other tests before the test under analysis, randomly taken from its containing test suite. As well as these, the test is executed in a baseline container, which makes no explicit attempt to manifest any particular kind of flakiness. After some upper limit of executions, the failure rate (i.e., the ratio of test failures to total runs) is calculated for each container and the category associated with the container that has the largest difference in failure rate compared to the baseline container is presented as the most likely cause.

In the domain of web applications, Morán et al. [141] proposed a technique named FlakyLoc for identifying the root causes of flaky tests. Similar to the work of Presler-Marshall et al. [153], their technique consisted of repeatedly executing a Selenium [32] test suite under multiple configurations of a range of factors believed to impact the likelihood of manifesting test flakiness. These factors were: the operating system, the screen resolution, the web driver (i.e., web browser), the number of CPU cores, the network bandwidth, and the amount of available memory. In the case that a test fails under some configurations and not others, FlakyLoc identifies the likely cause by calculating suspiciousness rankings for each factor, as would be done for source code lines in the context of fault localization [72, 75, 172, 178].

Formalizing a test case as a series of discrete "steps," Groce et al. [94] presented a modification to the delta debugging approach [182], a type of binary search, for minimizing the set of such steps required to exhibit non-determinism. Conceivably, this could help a developer to identify the specific cause of a flaky test. Their approach operates with two types of non-determinism. The first, that they term *horizontal non-determinism*, refers to a divergence in execution traces between two independent runs of a test case. The second, *vertical non-determinism*, describes the case where a supposedly idempotent operation, such as a file system operation that fails due to access permissions, returns inconsistent results following repeated executions in the same trace.

Table 14. A Comparison of Three different Studies That Proposed Techniques That May Repair Particular Types of Flaky Tests

Study	Targets	Underlaying Technique
Shi et al. [159]	Order-dependent tests	Delta debugging [182]
Fazzini et al. [83]	External dependencies	Component-based program synthesis [113]
Zhang et al. [184]	Implementation-dependent tests	Template-based repair [131]

Implementing their technique in Python, an evaluation with four subjects found that it reduced the number of steps in non-deterministic test cases by between 85% and 99%.

6.2.5 Automatic Approaches to Generate or Improve Repairs. Shi et al. [159] developed a tool for automatically repairing order-dependent tests, named iFixFlakies, using the statements of other tests in the test suite, which they refer to as *helpers*. They identified two categories of order-dependent test, *victims*, which pass when run in isolation but may fail when executed after other tests, and *brittles*, which fail in isolation and thus depend on previous tests to be executed beforehand to pass. They refer to a subsequence of tests that appear to induce a failure in a victim as a *polluter* and those that result in a brittle passing as a *state-setter*. Furthermore, they refer to a sequence of tests executed between a polluter and a victim that appears to enable the victim to pass as a *cleaner*, in other words, it appears to "clean" the state pollution. Together with state-setters, these constitute the two types of helpers that they reasoned may contain the functionality needed to repair the order dependence in the corresponding victim or brittle. Given an order-dependent test and an example of a passing and a failing test run order, their tool employs delta debugging [182] to identify a minimal sequence of tests that act as a polluter in the case of a victim, or a state-setter in the case of a brittle. In the case of a victim, iFixFlakies will apply a delta debugging approach again to find a minimal cleaner for the identified polluter. Applying delta debugging once more, it identifies the minimal set of statements from the identified helper tests that, when arranged as a generated method called at the beginning of the order-dependent test (much like a set up method), induces it to pass in the previously failing order. To evaluate their framework, they recycled subjects from previous work [122], arriving at a subject of 13 Java modules. Their respective test suites contained a total of 6,744 tests, 110 of which were identified as order-dependent. With these order-dependent tests, iFixFlakies was able to generate an average of 3.2 unique patches, with a mean size of 1.5 statements, that removed their order dependency. On average, it took only 186 s to generate a patch. They reported no cases where iFixFlakies was unable to generate a patch for an order-dependent test. To further demonstrate the validity of their generated patches, they submitted them as pull requests to their respective projects. At the time of writing, 21 had been accepted by their developers.

Having categorized flaky tests discovered by Microsoft's distributed build system, Lam et al. [121] identified the majority (78%) as being of the *asynchronous wait* category. This motivated them to specifically examine how such flaky tests had been previously repaired. By examining pull requests, they determined that the most common fix was related to changing the time delay between making the asynchronous call and evaluating assertions, accounting for 31% of cases. This further motivated them to propose the *Flakiness and Time Balancer* or *FaTB* technique for automatically improving such fixes by reducing their execution time. Their approach is essentially a binary search for the shortest waiting time that does not result in flakiness. Starting with the waiting time before the flaky test was fixed and the new waiting time that fixed the flaky test, the *FaTB* technique repeatedly bisects this interval and re-executes the test 100 times. For example, if a flaky test previously waited for 500 ms before being adjusted to wait for 1000 ms, *FaTB* would try 750 ms. If this reintroduced flakiness after 100 repeats, then it would attempt 875 ms.

Otherwise, if no flakiness was observed, then it would try the middle point between no waiting and the previously successful time of 750 ms (i.e., 375 ms). This process then continues up to some upper limit of iterations. To evaluate this technique, they randomly sampled five applicable flaky tests from their dataset. In each case, their technique was able to find a waiting time that did not manifest flakiness and was shorter than the initial developer's fix. However, for four of their sampled tests they never observed any flakiness at all, indicating that a larger set of tests may be needed to properly evaluate the technique.

Fazzini et al. [83] proposed a technique for the automatic generation of test mocks [163] in mobile applications. Test mocks replace real interactions between a mobile application and its environment (e.g., camera, microphone, GPS), a potential source of flakiness [170]. As such, their framework may not only improve the maintainability and performance of a test suite, but may also repair flaky tests. Their proposed technique, named MOKA, attempts to generate test mocks using a multi-stage strategy. Initially, MOKA leverages component-based program synthesis [113] to attempt to generate a test mock using a database of other mobile applications and their corresponding test suites. Should this fail, MOKA falls back to a "record and play" approach, which is to create a mock based on previous data recorded from the environmental interaction. On the same theme, Zhu et al. [187] proposed a machine learning-based technique, named MockSniffer, for suggesting to developers whether a particular external dependency ought to be mocked or not within a test case. Following an empirical study involving four large Java projects, the authors devised ten rules to match cases where developers had decided to use mock objects when testing. These rules stemmed from several observations, for example, the tendency of developers to mock classes related to networked services or concurrency. Encoding a mocking decision as a tuple consisting of the test case, the class under test, the dependency (a class used in the test case but not in the class under test) and a label indicating whether the dependency was mocked or not, the authors devised 16 features to train their machine learning model, based on their earlier observations. Using the same four projects as their training set, the authors built static analyzers for each feature and evaluated several models, finding Gradient Boosting [86] to be the best. To evaluate their approach in the absence of similar techniques, they compared MockSniffer to three baselines: one based on existing heuristics from previous studies, another using the mock list used by EvoSuite [82, 85], and a third based on the authors' observations from their empirical study. Using an evaluation set of six projects distinct from their four training projects, the authors found MockSniffer to generally outperform the three baselines.

Extending on the work of NonDex (see Section 5.1.4), Zhang et al. [184] proposed and evaluated a technique for repairing flaky tests arising from assumptions about non-deterministic or "underdetermined" specifications. Their approach, implemented as a tool named DexFix, uses an enhanced version of NonDex to identify tests in need of repair, their failing assertion and the root cause of the flakiness. Their technique then applies one of four template-based repair strategies based on the nature of the non-determinism. The first is to replace the AssertJ assertion method `containsExactly` [38], for comparing collections, to an alternative method that only checks contents and not the order. This addresses flaky tests that expect a particular order for unordered collection type objects, such as sets or hash maps, in an assertion statement. The second is to replace the statement `new HashMap` [44] or `new HashSet` [45] with `new LinkedHashMap` [46] or `new LinkedHashSet` [47], respectively. These latter classes subclass their respective former, unordered classes but guarantee a deterministic iteration order. This strategy addresses flaky tests that expect a particular iteration order for objects of these types. The third strategy is to sort the output of the `getDeclaredFields` method [42], which is specified to return an array of the fields of a class but in no particular order. This addresses flaky tests that expect some kind of order, previously identified as a common cause of implementation dependent flakiness in Java projects [97, 158]. The final strategy

addresses assertions that compare JSON strings [52] by replacing JUnit's `Assert.assertEquals` [54] method, which performs an exact string comparison, with `JSONAssert.assertEquals` from the JSONassert project [53]. The latter method does not consider the order of JSON *objects*, a set of key/value pairs, and so is far less strict and brittle than a simple string comparison. Using NonDex, Zhang et al. identified 275 flaky tests across 37 open-source Java projects from GitHub. Using DexFix, they were able to automatically repair 119. Of these generated patches, they submitted 102 as pull requests to the respective projects' repositories. At the time of writing, 74 had been accepted, 23 were still pending and 5 had been rejected. One reason for rejection, the authors noted, was the developer's hesitance to introduce a dependency on JSONassert.

Conclusion for RQ4.2: What methods and insights are available for repairing flaky tests? By examining historical commits in Apache Software Foundation projects, and surveying Mozilla developers, two studies agreed that the most common type of fix for flaky tests of the *asynchronous wait* category involved a `waitFor` method or its equivalent [79, 133]. For those of the *concurrency* category, common repairs also involved `waitFor`-like constructs, as well as ensuring mutual exclusion via adding locks to concurrent code [79, 133]. For order-dependent tests, the most common fixes involved a test's setup and teardown methods or explicit attempts to eliminate the dependency, by for instance creating a duplicate instance of some shared resource [79, 133]. A survey asking software developers to rate the most important pieces of information needed for fixing a flaky test, and the difficulty in obtaining them, identified the context leading to failure as the most important and the nature of the flaky test as the most difficult to obtain [79]. To that end, one study presented the RootFinder tool that compares attributes from the execution of tests in passing and failing cases to identify any differences that could indicate the root cause of flakiness [120]. Another technique employs multiple containers that each attempt to manifest some specific category of flakiness, with the category associated with the container with the greatest difference in failure rate, as compared to a baseline, considered the most likely cause [169]. For the automatic repair of order-dependent flaky tests, one source presented an approach using program statements from elsewhere in the test suite, generating fixes for 110 order-dependent tests, with 21 accepted by developers through pull requests [159]. For mobile applications, another study presented a framework for automatically generating test mocks, which could potentially repair flaky tests of several categories, including network, concurrency, and randomness. Another study presented DexFix [184], a template-based repair technique for automatically repairing implementation-dependent flaky tests, such as those detected by NonDex [97, 158]. Of 275 flaky tests across 37 open-source Java projects, DexFix could repair 119, the generated patches of 74 of which had been submitted as pull requests to their respective repositories and merged by their developers.

7 ANALYSIS

This section takes a high-level view of the studies examined in this survey. We identify the most prominent papers and authors and examine the various individual threads of research in the area, offering insights to those readers who want to familiarize themselves with the field. Furthermore, we provide a roundup of all the well-known tools, as examined in this survey, for detecting, mitigating, and repairing flaky tests and consider the various sets of projects used as subjects in their evaluations. Finally, we suggest future directions for research on flaky tests.

7.1 Prominent Papers and Authors

Table 15 shows the top ten most cited papers examined in this survey. This data is provided by Google Scholar [11] and is accurate as of the 7th of May, 2021. The most cited paper is *An Empirical*

Table 15. The Top Ten Most Cited Studies

Author	Paper	Cited by
Luo et al.	An Empirical Analysis of Flaky Tests [133]	251
Shamshiri et al.	Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges [156]	160
Martinez et al.	Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset [136]	160
Zhang et al.	Empirically Revisiting the Test Independence Assumption [185]	126
Memon et al.	Taming Google-scale Continuous Testing [138]	105
Hilton et al.	Trade-offs in Continuous Integration: Assurance, Security, and Flexibility [104]	105
Bell et al.	DeFlaker: Automatically Detecting Flaky Tests [69]	77
Bell et al.	Unit Test Virtualization with VMVM [67]	76
Vahabzadeh et al.	An Empirical Study of Bugs in Test Code [171]	73
Gyori et al.	Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency [98]	66

Table 16. The Top Five Most Prolific Authors

Author	Papers
Darko Marinov [69, 97, 98, 103, 104, 122, 124, 125, 133, 157–160, 176, 184]	15
August Shi [78, 97, 98, 122, 123, 150, 157–160, 184]	11
Wing Lam [120–125, 159, 176, 185]	9
Jonathan Bell [62, 67–69, 88, 103, 125, 146, 157]	9
Michael Hilton [62, 69, 77, 103, 104]	5

Analysis of Flaky Tests published in 2014 by Luo et al. [133]. Furthermore, it is cited by the majority of our sample of papers published after 2014, with two describing it as "seminal" [70, 151]. For these reasons, we consider it essential reading for people who want an introduction to the field of flaky tests. The fourth most cited paper *Empirically Revisiting the Test Independence Assumption*, was the first to compare different strategies for detecting order-dependent tests, and to present an automated tool for doing so [185]. This prefaced a line of further studies and a range of related tools [68, 71, 98, 122]. Several of the top ten papers are not specifically about flaky tests, but discuss a negative impact of flaky tests, such as *Taming Google-Scale Continuous Testing* [138]. We would recommend anyone beginning research into flaky tests to read these papers for a solid foundation in the research area.

The top five most frequent authors of our set of studies are listed in Table 16. At the top is Darko Marinov [6]. Marinov has been active in the area of flaky tests since its early beginnings, being a coauthor of *An Empirical Analysis of Flaky Tests* [133]. Along with his students, August Shi [4] and Wing Lam [37], second and third in the list, respectively, he has been involved in the development of several automatic tools for detecting flaky tests, including iDFlakies [122], PolDet [98] and NonDex [97, 158]. Along with other collaborators, they were the first to develop and evaluate a fully automatic approach for repairing order-dependent tests [159] and implementation-dependent tests [184]. At number four on the list, but joint third with Lam, is Jonathon Bell [20], who has also been involved in the creation of numerous flaky test detection techniques such as ElectricTest [68], PraDeT [88] and DeFlaker [69]. Bell frequently coauthors with Michael Hilton [24], at number five on the list, who has been particularly active on test flakiness in the context of continuous integration [77, 104]. We would direct those individuals who want to read more about flaky test research to the publications of these five authors.

7.2 Tools and Subject Sets

Table 17 provides a complete list of all the named tools for detecting, mitigating and repairing flaky tests examined in this survey. To evaluate the performance of these tools, a number of sets of projects have been created as subjects and in several cases have been reused in multiple studies. Zhang et al. [185] used four open-source Java projects, containing a combined total of 4,176

Table 17. List of Named Tools Examined in This Study, with Descriptions, Citations, and References to Relevant Sections in this Article

Tool	Description	Source	Section
DTDetector	Detects order-dependent tests by either reversing the test suite order, shuffling it, executing every k-permutation in isolation or only executing permutations that are likely to expose a test order dependency, based upon conservative analysis of static field access between tests and monitoring file usage.	[185]	§ 5.2.3
OrcalePolish	May indirectly identify order-dependent tests, or tests that have the potential to become order-dependent, by detecting brittle assertions using taint analysis.	[106]	§ 5.2.1
VmVm	Implements a lightweight alternative to full process isolation for mitigating possible test order dependencies by only reinitializing classes containing static fields that may facilitate state-based side effects between test runs.	[67]	§ 6.1.1
ElectricTest	Refines the dependency analysis of DTDetector by considering aliasing, using an approach that leverages garbage collection and object tagging. Unlike DTDetector, ElectricTest does not verify the order-dependent tests it identifies, i.e., by executing them, meaning that it may give many false positives.	[68]	§ 5.2.4
PolDet	Models heap memory as a multi-rooted graph, with objects, classes and primitive values as nodes, fields as edges and static fields of all loaded classes as roots. Computes and compares heap graph before the setup phase and after the teardown phase of each test run to identify any observable changes in program state or "pollution", to arrive a list of polluting tests, i.e., tests that may cause (but may not themselves be impacted by) test order dependencies.	[98]	§ 5.2.2
NonDex	Manifests flaky tests caused by the assumption of a deterministic implementation of a non-deterministic specification by reimplementing a range of methods and classes in the Java standard library to randomize the non-deterministic elements of their respective specifications.	[97, 158]	§ 5.1.4
DeFlaker	Given a new version of the software under test, i.e., after a commit, detects flaky tests as those whose outcome changes despite not covering any modified code.	[69]	§ 5.1.3
PraDeT	Follows a similar methodology as ElectricTest, but verifies its findings by modelling a test suite as a graph, with tests as nodes and possible dependencies as edges, and proceeds to invert each edge and schedule a corresponding test run to identify if the respective dependency is manifest, i.e., if it affects the outcome of the dependent test.	[88]	§ 5.2.5
RootFinder	Collects and processes information recorded from passing and failing executions of repeated test runs via instrumented method calls. Aims to assist developers in understanding the causes of flaky tests by, for example, showing the differences in the execution environment between passing and failing cases.	[120]	§ 6.2.4
TEDD	Detects order-dependent tests in web applications by considering dependencies facilitated by persistent data stored on the server-side and implicit shared data via the Document Object Model on the client-side, rather than by internal Java objects.	[71]	§ 5.2.6
iDFlakies	Classifies flaky tests as being order-dependent or not based upon comparing the outcomes of repeated executions of failing tests in a modified test run order to the original test run order.	[122]	§ 5.2.7
iFixFlakies	Generates fixes for an order-dependent test from the statements of other tests in the test suite that have been identified as "helpers", i.e., that enable the dependent test to pass when executed previously.	[159]	§ 6.2.5
FLASH	Targeting Machine Learning applications, mines test suites for approximate assertions and repeatedly executes the containing tests to arrive at a sample of actual values evaluated within them. From these samples, FLASH estimates the probability that an assertion will fail and declares a containing test flaky if it is above a threshold.	[78]	§ 5.1.6
FLAST	A purely static approach to flaky test detection, represents tests using a bag of words model over the tokens of their source code and trains a k-nearest-neighbor classifier to label them as flaky or not.	[70]	§ 5.1.8
MOKA	Uses component-based program synthesis to automatically generate test mocks for mobile apps, thereby potentially repairing flaky tests caused by dependence upon external data.	[83]	§ 6.2.5
Shaker	Introduces CPU and memory stress during repeated test suite executions in an attempt to increase the probability of manifesting flaky tests of the *asynchronous wait* and *concurrency* categories.	[162]	§ 5.1.5
DexFix	Uses template-based automatic program repair to fix implementation-dependent flaky tests, such as those detected by NonDex.	[184]	§ 6.2.5
FlakeFlagger	A technique for detecting flaky tests without requiring test suite reruns, using a machine learning model. Requires a combination of dynamic test data, such as line coverage, and static data, such as features of the test's source code.	[62]	§ 5.1.8

developer-written tests, to compare the performance of various configurations of DTDetector, their order-dependent test detection tool. These four subjects were then reused as part two further evaluations of later tools, namely, ElectricTest [68] and PraDeT [88], and, along with the results of these three evaluations, are presented in Table 10. Bell et al. [69] presented a set of 5,966 commits

Table 18. Connections between Related Areas and the Field of Flakiness, with Citations and References to Relevant Sections in This Article

Area	Reference	Section
Continuous integration	[77, 103, 104, 118, 119, 121, 129, 138, 160]	§ 3.1.4, § 4.2.1, § 6.2.4
Test acceleration	[68, 71, 73, 88, 123, 185]	§ 3.2, § 4.2, § 5.2, § 6.1.1
User interface testing	[89, 90, 153, 155, 181]	§ 3.3.1, § 4.2.2, § 4.3.6, § 6.1.2
Software engineering education	[153, 158, 164]	§ 4.3.6
Automatic test generation	[64, 149, 156]	§ 4.3.3, § 6.1.4
Mutation testing	[102, 157]	§ 4.3.1, § 6.1.3
Automatic program repair	[136, 179]	§ 4.3.5
Machine learning testing	[78, 145]	§ 3.3.2, § 5.1.6
Fault localization	[172]	§ 4.3.2

across 26 open-source Java projects comprising 28,068 test methods. They used these to evaluate the performance of DeFlaker, their flaky test detection tool. The DeFlaker subject set was also used by Pinto et al. to train various classifiers using natural language processing techniques in an attempt to statically identify flaky tests [151]. Projects from this subject set, along with various other sources, such as popular projects on GitHub, were combined by Lam et al. [122] to create two subject sets for their evaluation of their own detection tool called iDFlakies. The first of these two sets, called the *comprehensive* set, contained 183 open-source Java projects. The second of these, called the *extended* set, consisted of 500 further projects disjoint from the *comprehensive* set. The projects of the *comprehensive* set were then reused in later studies involving Lam [121, 123, 125, 159]. Flaky tests identified from these subject sets were later called the *Illinois Dataset of Flaky Tests* [62, 176].

7.3 Related Research Areas

In many instances, the flaky test literature is tangential with several other research areas. An index of these are given in Table 18, with relevant citations from our set of collected papers, and references to the sections in this article that mention them. Emerging as the area with the strongest intersection is continuous integration research. Our associated citations pertain to the evaluation of the prevalence and impacts of test flakiness with respect to continuous integration systems. Since such systems involve a vast number of test executions, it is perhaps unsurprising that there is a strong overlap between this area of study and flaky test research. Test acceleration research, producing techniques that often change the test execution schedule, such as test suite selection [129, 134, 160], prioritization [101, 150, 181], reduction [63, 130, 173] and parallelization [68, 73, 100], is another area with a significant relationship with flaky tests.

7.3.1 Continuous Integration. Flaky tests have been studied several times in the context of continuous integration systems. An early source described an account of how transitioning to a continuous integration system led to developers observing the true scale of the flakiness in their test suites, since tests were being executed more often [119]. Since then, continuous integration systems have been a frequent object of study due to the vast amount of test execution data they make available. Labuschagne et al. [118] studied the build history of open-source projects using Travis and found that 13% of a sample of transitioning tests were flaky. Hilton et al. [104] deployed a survey asking to developers to estimate the number of continuous integration builds that failed due to true test failures and due to flaky test failures, finding no significant difference between the two distributions. Memon et al. [138] analysed Google's internal continuous integration system and found flakiness was to blame for 41% of "test targets" that had previously passed at least once and failed at least once. Microsoft's distributed build system has also been an object of study several times, with one study finding that had it not automatically identified and filtered the 0.02% of

sampled test executions that were flaky, they would have gone on to cause what would have been 5.7% of all failed builds, demonstrating the impact that a relatively small number of flaky tests can impose [121]. Finally, Durieux et al. [77] found that 47% of previously failing builds that had been manually restarted went on to pass, suggesting that they may have initially failed due to flaky tests.

7.3.2 Test Acceleration. A specific category of flaky tests, order-dependent tests, have been identified as a potential burden to the soundness of much research in the area of test acceleration. This is because they may produce inconsistent outcomes when their containing test suites are reordered or otherwise modified. Zhang et al. [185] examined how many test cases gave inconsistent outcomes when applying five different test prioritization schemes. Bell et al. [68] proposed a scheduler for achieving sound parallelization when test suites contain order-dependent tests, though this was found to be at the cost of a considerable degree of efficiency. Candido et al. [73] examined the degree to which order-dependent tests were manifested when applying several different test suite parallelization techniques. Lam et al. [123] proposed an algorithm for ensuring order-dependent tests had their dependencies satisfied after applying test acceleration, while attempting to minimize the loss of speedup in the test suite execution process.

7.4 Future Research Directions

Recent research has proposed a technique for the automatic repair of order-dependent flaky tests [159]. One limitation of this approach is the requirement that the statements that would fix the dependency must be present elsewhere in the test suite. A possible avenue to explore is the application of techniques from automatic program repair, a field of research concerning the automatic generation of patches for bugs [92, 128, 140], to the task of repairing flaky tests [148], which could be considered a type of test bug [171]. Such an approach could make use of a series of manually authored, pre-defined templates for applying fixes, though naturally this would be limited in scope. As examined in Section 6.2.5, such a tool has already been implemented for the repair of two common cases of implementation-dependent flaky tests [184]. Conceivably, a more general approach could automatically derive repair templates from historical examples of flaky test repairs, via GitHub for example. This would remove the requirement for the fix to come from elsewhere in the test suite and could widen the range and variety of flaky tests that could be automatically repaired. Furthermore, the current state of research has only proposed automatic repair techniques for just two specific types of flakiness. There would potentially be great benefit in extending this work to cover the *asynchronous wait* and *concurrency* categories of flakiness, which have frequently been identified as the leading causes, as mentioned in Section 3.1.

On a different note, Harman and O'Hearn [102] proposed the aphorism "Assume all Tests Are Flaky" (ATAF), thereby taking the view that all tests have the potential to be non-deterministic and that testing methodologies ought to accommodate for this, rather than to assume that tests are generally deterministic and that flaky tests are an exception. Overall, the general thesis of ATAF is to move away from efforts to control and mitigate flakiness, in the hope of making testing fully deterministic, and to move toward a situation where flaky tests are considered an unavoidable aspect of testing that are fully considered and accommodated. To that end, the authors proposed five open research questions regarding the assessment, prediction, amelioration, reduction, and general accommodation of test flakiness.

7.4.1 Assessment. The first research question considers the possibility of an approximate measurement of the flakiness of a test case that goes beyond simply classifying it as flaky or not. So far, studies have measured flakiness using entropy [90, 116] and recently one has defined its own flakiness-ratio (FR) calculated for a test case τ as $FR(\tau) = 2 * min(\tau_P, \tau_F) \div (\tau_P + \tau_F)$, where τ_P and

τ_F are the numbers of passing and failing runs of τ, respectively [172]. Another study considered the failure rates of flaky tests when executed under different settings, drawing various conclusions about the probability of a flaky test manifesting itself and how many repeated test runs are likely to be necessary. Their finding that certain test run orders can lead to different failure rates for the same test, as opposed to just always failing or always passing, suggests that the binary classification of a flaky test as order-dependent or not may be insufficient, providing further motivation for a more continuous assessment of flakiness [124].

7.4.2 Prediction. Their second research question concerns the development of predictive models for flaky tests, such that they may be identified without having to execute them repeatedly. An attempt at fitting a Bayesian Network model to predict flakiness based on a diversity of test features, such as code complexity and historical failure rates, demonstrated mixed results [114]. Machalica et al. [134] trained a model to predict if tests would fail, given a set of code changes, that was used as part of a test selection system. It is conceivable that a similar approach might be useful for predicting if a test might fail flakily. An emerging thread of research has been concerned with the application of machine learning models and natural language processing techniques for the prediction of test flakiness based on purely static features of the body of the test case [70, 151]. Recently, Alshammari et al. [62] combined these static features with dynamic features of a test case, such as its line coverage, demonstrating that the combination of both feature sets considerably improves the model's predictive effectiveness. However, their results demonstrated that there is still some way to go before the development a predictive model that would be reliable enough for practical use by developers.

7.4.3 Amelioration. Their third question asks if we can find ways to transform test goals so that they yield stronger signals to developers when tests are flaky. For example, in the context of regression testing, it may be desirable to apply test acceleration techniques such as selection or prioritization to reduce the amount of time taken to receive useful feedback from the runs of a test suite. In the past, objectives have included the selection/prioritization of tests based on the coverage of modified program elements, runtime and energy consumption [180]. Under ATAF, a new objective could be the degree of test determinism, to achieve greater certainty sooner in the test run. To that end, a reliable measure of test flakiness is required, as per their previous research question regarding the assessment of flaky tests.

7.4.4 Reduction. Their fourth question regards how to reduce the flakiness of a test, or more abstractly, to reinterpret the signal of a flaky test in some way that makes flakiness less of an issue. The authors then go on to describe the possibility of an abstract interpretation approach [76] for testing and verification such that an increased degree of abstraction reduces the degree of variability and non-determinism in test outcomes. A possible realization of this, to potentially reduce the flakiness stemming from the algorithmic non-determinism commonly seen in machine learning projects [78, 145], could reinterpret the outputs of probabilistic functions as parameterized probability distributions instead of concrete values. An assertion that then checks if an output is within an acceptable range, a type of oracle approximation [145] linked to flakiness when too restrictive [79], would instead pass if the probability of observing a value in that range within the abstractly interpreted distribution was above some acceptable threshold. Such a reinterpretation could eliminate the flakiness when corner cases are outside of the assumed range of acceptable outputs.

7.4.5 Reformulation. Their fifth and final question asks how the various techniques impacted by flaky tests, such as those examined in Section 4, could be reformulated to cater for non-deterministic tests. To that end, one recent study proposed reformulations of various fault

localization metrics that are "backward compatible" with their original forms but are able to accommodate flaky test outcomes [172]. Beyond flaky outcomes, the coverage of tests has also been observed to be inconsistent [89, 103, 157]. This potentially motivates the need for the reformulation or generalization of techniques that depend on coverage information, such as fault localization again, and also mutation testing [157] and search-based automatic test generation [111]. In the latter case, the coverage of automatically generated tests is often used as part of a fitness function that guides a search algorithm such as hill climbing, simulated annealing or an evolutionary search [137]. Given that the coverage of tests, automatically generated or otherwise, may not always be as deterministic as perhaps initially thought, more abstract measures of test suite quality may be preferable.

8 CONCLUSION

In this survey, we examined 76 sources related to test flakiness and answered four research questions regarding their origins, consequences, detection, mitigation, and repair. Our first research question considered the causes of flaky tests and the factors associated with their occurrence. We identified a taxonomy of general causes and considered the specific factors associated with order-dependent tests in particular as well as how the causes of flakiness can differ across specific types of projects. We then set out to examine what areas of testing are impacted by flaky tests, allowing us to answer our second research question regarding their consequences and costs. We found that they impose negative effects on the reliability and efficiency of testing in general, as well as being a hindrance to a host of techniques in software engineering. Our third research question concerned the detection of flaky tests. We first examined approaches for detecting flaky tests, in general, before discussing a series of tools for the detection of order-dependent tests, in particular. Our final research question set out to investigate the ways in which the negative impacts of flaky tests can be mitigated, or going a step further, repaired entirely. To that end, we presented a range of techniques for alleviating some of the problems presented by flaky tests, as previously identified, and described a technique for the automatic repair of order-dependent tests. Overall, we provided researchers with a snapshot of the current state of research in the area of flaky tests and identified topics for which further work may be beneficial. We also gave those wishing to familiarize themselves with the field the necessary reading for a succinct knowledge of the insights and achievements to date.

ACKNOWLEDGMENTS

We thank the numerous cited authors who reviewed a draft of this survey and provided feedback.

REFERENCES

[1] ACM Digital Library. 2021. [Online]. Retrieved on Oct. 6, 2021 from https://dl.acm.org/.
[2] Agitar Technologies. 2021. Retrieved on Oct. 6, 2021 from http://www.agitar.com/.
[3] The Apache Software Foundation. [Online]. Retrieved on Oct. 6, 2021 https://www.apache.org/.
[4] S. August, Personal web page. Retrieved on Oct. 6, 2021 from https://sites.utexas.edu/august/.
[5] Coroutines and Tasks. Retrieved on Oct. 6, 2021 from https://docs.python.org/3/library/asyncio-task.html.
[6] D. Marinov, Personal web page. Retrieved on Oct. 6, 2021 from http://mir.cs.illinois.edu/marinov/.
[7] Docker. Retrievd on Oct. 6, 2021 from https://www.docker.com/.
[8] EvoSuite Automatic Test Suite Generation for Java. Retrieved on Oct. 6, 2021 from https://www.evosuite.org/.
[9] Box, Flaky. Retreived Oct. 6, 2021 from https://github.com/box/flaky.
[10] Google Chrome. Retrieved on Oct. 6, 2021 from https://www.google.com/chrome/.
[11] Google Scholar. Retrievd on Oct. 6, 2021 from https://scholar.google.com/.
[12] Home Assistant, Core. Retrieved on Oct. 6, 2021 from https://github.com/home-assistant/core.
[13] Balloob, Fix flaky media player test. 2020. Retrieved on Oct. 6, 2021 from https://github.com/home-assistant/core/commit/2b42d77f5899b9a87b7713c4582b71bce1fd40dd.
[14] HtmlUnit. 2021. Retrieved on Oct. 6, 2021 from https://htmlunit.sourceforge.io/.
[15] Facebook Research, Hydra. Retrieved on Oct. 6, 2021 from from https://github.com/facebookresearch/hydra.

[16] IEEE Xplore. Retrieved on Oct. 6, 2021 from https://ieeexplore.ieee.org/Xplore/home.jsp.
[17] ICSE International Conference on Software Engineering. 2021. Retrieved on Oct. 6, 2021 from http://www.icse-conferences.org/.
[18] Class Process. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/lang/Process.html.
[19] Class Thread. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html.
[20] J. Bell, Personal web page. Retrieved on Oct. 6, 2021 from https://www.jonbell.net/.
[21] Java Virtual Machine Tool Interface. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/.
[22] Canonical, Launchpad. Retrieved on Oct. 6, 2021 from https://launchpad.net/.
[23] Apache Maven Project. 2021. Retrieved on Oct. 6, 2021 from https://maven.apache.org/.
[24] M. Hilton, Personal web page. Retrieved on Oct. 6, 2021 from https://www.cs.cmu.edu/~mhilton/.
[25] Mozilla, Firefox. Retrieved on Oct. 6, 2021 from https://www.mozilla.org/en-GB/firefox/.
[26] PhantomJS Scriptable Headless Browser. Retrieved on Oct. 6, 2021 from https://phantomjs.org/.
[27] Pitest. Retrieved on Oct. 6, 2021 from https://pitest.org/.
[28] Pytest. Retrieved on Oct. 6, 2021 from https://docs.pytest.org/en/6.2.x/.
[29] Randoop Automatic Unit Test Generation for Java. Retrieved on Oct. 6, 2021 from https://randoop.github.io/randoop/.
[30] Scratch. Retrieved on Oct. 6, 2021 from https://scratch.mit.edu/.
[31] sendKeys. Retrieved on Oct. 6, 2021 from https://www.selenium.dev/documentation/webdriver/keyboard/.
[32] Selenium. Retrieved on Oct. 6, 2021 from https://www.selenium.dev/.
[33] Springer Link. Retrieved on Oct. 6, 2021 from https://link.springer.com/.
[34] Rerun Failing Tests. 2020. Retrieved on Oct. 6, 2021 from https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html.
[35] test_compose.py. 2019. Retrieved on Oct. 6, 2021 from https://github.com/facebookresearch/hydra/blob/804dab9e710496112260c42e3be57c5467b0c2dd/tests/test_compose.py#L54.
[36] Travis CI. Retrieved on Oct. 6, 2021 from https://travis-ci.org/.
[37] W. Lam, Personal web page. Retrieved on Oct. 6, 2021 from http://mir.cs.illinois.edu/winglam/.
[38] AssertJ Fluent Assertions Java Library. Retrieved on Oct. 6, 2021 from https://assertj.github.io/doc/.
[39] Mining Software Repositories. Retrieved on Oct. 6, 2021 from http://www.msrconf.org/.
[40] The International Conference on Software Testing, Verification and Validation. Retrieved on Oct. 6, 2021 from https://cs.gmu.edu/icst/.
[41] Class Calendar. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html.
[42] Class Class. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html.
[43] Class DateFormatSymbols. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/text/DateFormatSymbols.html.
[44] Class HashMap. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html.
[45] Class HashSet. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html.
[46] Class LinkedHashMap. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html.
[47] Class LinkedHashSet. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html.
[48] Class Object. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html.
[49] Class Random. Retrieved Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/util/Random.html.
[50] Class SecurityManager. Retrieved on Oct. 6, 2021 from https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html.
[51] ESEC / FSE. Retrieved on Oct. 6, 2021 from https://www.esec-fse.org/.
[52] Introducing JSON. Retrieved on Oct. 6, 2021 from https://www.json.org/json-en.html.
[53] JSONassert. Retrieved on Oct. 6, 2021 from http://jsonassert.skyscreamer.org/cookbook.html.
[54] Class Assert. Retrieved on Oct. 6, 2021 from https://junit.org/junit4/javadoc/latest/org/junit/Assert.html.
[55] Junit 5. Retrieved on Oct. 6, 2021 from https://junit.org/junit5/.
[56] R. Wagner. 2020. Remove test_spinup_time. Retrieved on Oct. 6, 2021 from https://github.com/OpenMined/PySyft/commit/fc729acc88c2c26ece4fb5e4a931a7a13fe0971a.
[57] OpenMined, PySyft. Retrieved on Oct. 6, 2021 from https://github.com/OpenMined/PySyft.
[58] The Python Package Index. Retrieved on Oct. 6, 2021 from https://pypi.org/.
[59] O. Parry. 2021. A Survey of Flaky Tests: Bibliography. Retrieved on Oct. 6, 2021 from https://github.com/flake-it/flaky-tests-bibliography.
[60] 2021. Scopus. Retrieved Oct. 6, 2021 from https://www.scopus.com/.

[61] A. Ahmad, O. Leifler, and K. Sandahl. 2020. An evaluation of machine learning methods for predicting flaky tests. In *Proceedings of the International Workshop on Quantitative Approaches to Software Quality (QuASoQ'20)*. 37–46.
[62] A. Alshammari, C. Morris, M. Hilton, and J. Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *Proceedings of the International Conference on Software Engineering (ICSE'21)*.
[63] A. Alsharif, G. M. Kapfhammer, and P. McMinn. 2020. STICCER: Fast and effective database test suite reduction through merging of similar test cases. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'20)*.
[64] A. Arcuri, G. Fraser, and J. Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *Proceedings of the International Conference on Automated Software Engineering (ASE'14)*. 79–89.
[65] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. 2015. Are test smells really harmful? an empirical study. *Empir. Softw. Eng.* 20, 4 (2015), 1052–1094.
[66] J. Bell. 2014. Detecting, isolating, and enforcing dependencies among and within test cases. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'14)*. 799–802.
[67] J. Bell and G. Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*. 550–561.
[68] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*. 770–781.
[69] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE'18)*. 433–444.
[70] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia. 2021. Know your neighbor: Fast static prediction of test flakiness. *IEEE Access* 9, 76119–76134. Issue 4.
[71] M. Biagiola, A. Stocco, A. Mesbah, F. Ricca, and P. Tonella. 2019. Web test dependency detection. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 154–164.
[72] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. 2013. Entropy-based test generation for improved fault localization. In *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*. 257–267.
[73] J. Candido, L. Melo, and M. D'Amorim. 2017. Test suite parallelization in open-source projects: A Study on its usage and impact. In *Proceedings of the International Conference on Automated Software Engineering (ASE'17)*. 153–158.
[74] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. SMOTE: Synthetic Minority over-sampling technique. *J. Artific. Intell. Res.* 16 (2002), 321–357.
[75] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, and M. J. Harrold. 2011. Localizing SQL Faults in database applications. In *Proceedings of the International Conference on Automated Software Engineering (ASE'11)*.
[76] Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *J. Logic Comput.* 2, 4 (1992), 511–547.
[77] T. Durieux, C. L. Goues, and R. Hilton, M. Abreu. 2020. Empirical study of restarted and flaky builds on Travis CI. In *Proceedings of the International Conference on Mining Software Repositories (MSR'20)*. 254–264.
[78] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and Misailovic S. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'20)*. 211–224.
[79] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 830–840.
[80] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. 2003. Framework for testing multi-threaded Java programs. *Concurr. Comput.: Exper.* 15 (2003), 485–499.
[81] A. T. Endo and A. Moller. 2020. NodeRacer: Event race detection for Node.js applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'20)*. 120–130.
[82] Z. Fan. 2019. A systematic evaluation of problematic tests generated by EvoSuite. In *Proceedings of the International Conference on Software Engineering (ICSE'19)*. 165–167.
[83] M. Fazzini, A. Gorla, and A. Orso. 2020. A framework for automated test mocking of mobile apps. In *Proceedings of the International Conference on Automated Software Engineering (ASE'20)*. 1204–1208.
[84] M. Fowler. 2011. Eradicating Non-Determinism in Tests. Retrieved from https://martinfowler.com/articles/nonDeterminism.html.
[85] G. Fraser and A. Arcuri. 2014. A large scale evaluation of automated unit test generation using EvoSuite. *Trans. Softw. Eng. Methodol.* 24 (2014), 8.
[86] J. H. Friedman. 2001. Greedy function approximation: A gradient boosting machine. *Ann. Stat.* 29, 5 (2001), 1189–1232.
[87] N. Friedman, D. Geiger, and M. Goldszmidt. 1997. Bayesian network classifiers. *Mach. Learn.* 29 (1997), 131–163.

[88] A. Gambi, J. Bell, and A. Zeller. 2018. Practical test dependency detection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'18)*. 1–11.

[89] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *Proceedings of the International Conference on Software Engineering (ICSE'15)*. 55–65.

[90] Z. Gao and A. Memon. 2015. Which of my failures are real? Using relevance ranking to raise true failures to the top. In *Proceedings of the International Conference on Automated Software Engineering Workshops (ASEW'15)*. 62–60.

[91] V. Garousi and B. Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *J. Syst. Softw.* 138 (2018), 52–81.

[92] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic software repair: A survey. *Trans. Softw. Eng.* 45, 1 (2019), 34–67.

[93] S. W. Golomb and H. Taylor. 1985. Tuscan squares—A new family of combinatorial designs. *Ars Combinatoria* 20 (1985), 115–132.

[94] A. Groce and J. Holmes. 2021. Practical automatic lightweight nondeterminism and flaky test detection and debugging for Python. In *Proceedings of the International Conference on Software Quality, Reliability, and Security (QRS'21)*. 188–195.

[95] S. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. 2021. An empirical study of flaky tests in Python. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'21)*.

[96] A. Gyori, B. Lambeth, S. Khurshid, and D. Marinov. 2017. Exploring underdetermined specifications using Java PathFinder. *Softw. Eng. Notes* 41 (2017), 1–5.

[97] A. Gyori, B. Lambeth, A. Shi, O. Legunsen, and D. Marinov. 2015. NonDex: A tool for detecting and debugging wrong assumptions on Java API Specification. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'15)*. 223–233.

[98] A. Gyori, A. Shi, F. Hariri, and D. Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA'15)*. 223–233.

[99] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon. 2021. A replication study on the usability of code vocabulary in predicting flaky tests. In *Proceedings of the International Conference on Mining Software Repositories (MSR'21)*.

[100] F. Haftmann, D. Kossmann, and E. Lo. 2005. Parallel execution of test runs for database application systems. In *Proceedings of the International Conference on Very Large Data Bases*.

[101] Shifa E. Zehra Haidry and Tim Miller. 2013. Using dependency structures for prioritization of functional test suites. *Trans. Softw. Eng.* 39 (2013), 258–275.

[102] M. Harman and P. O'Hearn. 2018. From start-ups to scale-ups: opportunities and open problems for static and dynamic program analysis. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'18)*. 1–23.

[103] M. Hilton, J. Bell, and D. Marinov. 2018. A large-scale study of test coverage evolution. In *Proceedings of the International Conference on Automated Software Engineering (ASE'18)*. 53–63.

[104] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig. 2017. Trade-Offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'17)*. 197–207.

[105] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*. 426–437.

[106] C. Huo and J. Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'14)*. 621–631.

[107] Y. Jia and M. Harman. 2011. An analysis and survey of the development of mutation testing. *Trans. Softw. Eng.* 37, 5 (2011), 649–678.

[108] R. Just, G. M. Kapfhammer, and F. Schweiggert. 2012. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'12)*. 11–20.

[109] G. M. Kapfhammer. 2004. Software testing. In *The Computer Science Handbook*. Chapman and Hall/CRC.

[110] G. M. Kapfhammer. 2010. Regression testing. In *The Encyclopedia of Software Engineering*. Wiley-Interscience.

[111] G. M. Kapfhammer, P. McMinn, and C. J. Wright. 2013. Search-based testing of relational schema integrity constraints across multiple database management systems. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'13)*.

[112] J. M. Keller, M. R. Gray, and J. A. Givens. 1985. A fuzzy k-Nearest-neighbor algorithm. *Trans. Syst. Man Cybernet.* 4 (1985), 580–585.

[113] S. Kensen, J. Steinhardt, and P. Liang. 2019. FrAngel: Component-based synthesis with control structures. *Proceedings of the ACM Conference on Programming Languages*.
[114] T. King, D. Santiago, J. Phillips, and P. Clarke. 2018. Towards a Bayesian network model for predicting flaky automated tests. In *Proceedings of the International Conference on Software Quality, Reliability, and Security Companion (QRS-C'18)*. 100–107.
[115] B Korel, L. H. Tahat, and B. Vaysburg. 2002. Model-based regression test reduction using dependence analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*. 214–223.
[116] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon. 2020. Modeling and ranking flaky tests at Apple. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'20)*. 110–119.
[117] J. Kupiec. 1992. Robust part-of-speech tagging using a hidden Markov model. *Comput. Speech Lang.* 6, 3 (1992), 225–242.
[118] A. Labuschagne, L. Inozemtseva, and R. Holmes. 2017. Measuring the cost of regression testing in practice: A Study of Java projects using continuous integration. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'17)*. 821–830.
[119] F. Lacoste. 2009. Killing the gatekeeper: Introducing a continuous integration system. In *Proceedings of the Agile Conference (AGILE'09)*. 387–392.
[120] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'19)*. 204–215.
[121] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE'20)*. 1471–1482.
[122] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. IDFlakies: A framework for detecting and partially classifying flaky tests. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'19)*. 312–322.
[123] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie. 2020. Dependent-Test-Aware regression testing techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'20)*. 298–311.
[124] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *Proceedings of the International Conference on Software Reliability Engineering (ISSRE'20)*. 403–413.
[125] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, 11 (2020), 202–231.
[126] W. Lam, S. Zhang, and M. D. Ernst. 2015. *When Tests Collide: Evaluating and Coping With the Impact of Test Dependence.* Technical Report UW-CSE-15-03-01. Department of Computer Science and Engineering, University of Washington.
[127] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 Bugs for $8 Each. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*. 3–13.
[128] C. Le Goues, M. Pradel, and A. Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://dl.acm.org/doi/10.1145/3318162.
[129] C. Leong, A. Singh, M. Papadakis, Y. Le Traon, and J. Micco. 2019. Assessing transition-based test selection algorithms at Google. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19)*. 101–110.
[130] C. T. Lin, K. W. Tang, and G. M. Kapfhammer. 2014. Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests. *Info. Softw. Technol.* 56, 10 (2014), 1322–1344.
[131] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. 2019. TBar: Revisiting Template-based automated program repair. In *Proceedings of the International Conference on Software Testing and Analysis (ISSTA'19)*. 31–42.
[132] Q. Luo, L. Eloussi, F. Hariri, and D. Marinov. 2014. Can we trust test outcomes? *Network* 5, 1 (2014).
[133] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE'14)*. 643–653.
[134] M. Machalica, A. Samylkin, M. Porth, and S. Chandra. 2019. Predictive test selection. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19)*. 91–100.
[135] J. Malm, A. Causevic, B. Lisper, and S. Eldh. 2020. Automated analysis of flakiness-mitigating delays. In *Proceedings of the International Conference on Automation of Software Test (AST'20)*. 81–84.
[136] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. 2017. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empir. Softw. Eng.* 22, 4 (2017), 1936–1964.
[137] P. McMinn. 2004. Search-based software test data generation: A survey. *Softw. Test. Verificat. Reliabil.* 14, 2 (2004).
[138] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. 2017. Taming Google-Scale continuous testing. In *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'17)*. 233–242.

[139] J. Micco. 2017. The State of Continuous Integration Testing at Google. Retrieved from http://aster.or.jp/conference/icst2017/program/jmicco-keynote.pdf.
[140] M. Monperrus. 2018. Automatic software repair: A bibliography. *Comput. Surveys* 51, 1 (2018), 1–24.
[141] J. Morán, C. Augusto, A. Bertolino, C. De La Riva, and J. Tuya. 2019. Debugging flaky tests on web applications. In *Proceedings of the International Conference on Web Information Systems and Technologies (WEBIST'19)*. 454–461.
[142] R. Mudduluru, J. Waataja, S. Millstein, and M. D. Ernst. 2021. Verifying determinism in sequential programs. In *Proceedings of the International Conference on Software Engineering (ICSE'21)*.
[143] K. Muşlu, B. Soran, and J. Wuttke. 2011. Finding bugs by isolating unit tests. In *Proceedings of the Symposium on Foundations of Software Engineering (FSE'11)*. 496–499.
[144] A. Najafi, P. C. Rigby, and W. Shang. 2019. Bisecting commits and modeling commit risk during testing. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 279–289.
[145] M. Nejadgholi and J. Yang. 2019. A study of oracle approximations in testing deep learning libraries. In *Proceedings of the International Conference on Automated Software Engineering (ASE'19)*. 785–796.
[146] P. Nie, A. Celik, M. Coley, A. Milicevic, J. Bell, and M. Gligoric. 2020. Debugging the performance of maven's test isolation: Experience report. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'20)*. 249–259.
[147] F. Palomba and A. Zaidman. 2019. The smell of fear: On the relation between test smells and flaky tests. *Empir. Softw. Eng.* 24, 11 (2019), 2907–2946.
[148] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2020. Flake it 'till you make it: using automated repair to induce and fix latent test flakiness. In *Proceedings of the International Workshop on Automated Program Repair (APR'20)*. 11–12.
[149] S. Paydar and A. Azamnouri. 2019. An experimental study on flakiness and fragility of Randoop regression test suites. In *Lecture Notes in Computer Science*. 111–126.
[150] Q. Peng, A. Shi, and L. Zhang. 2020. Empirically revisiting and enhancing IR-based test-case prioritization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'20)*. 324–336.
[151] G. Pinto, B. Miranda, S. Dissanayake, M. D. Amorim, C. Treude, A. Bertolino, and M. D'amorim. 2020. What is the vocabulary of flaky tests? In *Proceedings of the International Conference on Mining Software Repositories (MSR'20)*. 492–502.
[152] J. A. Prado Lima and S. R. Vergilio. 2020. Test case prioritization in continuous integration environments: A systematic mapping study. *Info. Softw. Technol.* 121 (2020), 106268. https://doi.org/10.1016/j.infsof.2020.106268
[153] K. Presler-Marshall, E. Horton, S. Heckman, and K. T. Stolee. 2019. Wait wait. No, tell me. Analyzing Selenium configuration effects on test flakiness. In *Proceedings of the International Workshop on Automation of Software Testing (AST'19)*. 7–13.
[154] M. T. Rahman and P. C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. 857–862.
[155] A. Romano, Z. Song, S. Grandhi, W. Yang, and W. Wang. 2021. An empirical analysis of UI-based flaky tests. In *Proceedings of the International Conference on Software Engineering (ICSE'21)*.
[156] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. 2015. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*. 201–211.
[157] A. Shi, J. Bell, and D. Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'19)*. 296–306.
[158] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'16)*. 80–90.
[159] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 545–555.
[160] A. Shi, P. Zhao, and D. Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'19)*.
[161] T. Shi and S. Horvath. 2006. Unsupervised learning with random forest predictors. *J. Comput. Graph. Stat.* 15, 1 (2006), 118–138.
[162] D. Silva, L. Teixeira, and M. D'Amorim. 2020. Shake It! Detecting flaky tests caused by concurrency with Shaker. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'20)*. 301–311.

[163] D. Spadini, M. Aniche, M. Bruntink, and A. Bacchelli. 2017. To mock or not to mock? An empirical study on mocking practices. In *Proceedings of the International Conference on Mining Software Repositories (MSR'17)*. 402–412.

[164] A. Stahlbauerm, M. Kreis, and G. Fraser. 2019. Testing scratch programs automatically. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 165–175.

[165] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark. 2020. Intermittently failing tests in the embedded systems domain. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'20)*. 337–348.

[166] P. Sudarshan. 2012. No More Flaky Tests on the Go Team, Retrieved from https://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.

[167] S. Tahvili, M. Ahlberg, E. Fornander, W. Afzal, M. Saadatmand, M. Bohlin, and M. Sarabi. 2018. Functional dependency detection for integration test cases. In *Proceedings of the International Conference on Software Quality, Reliability and Security Companion (QRS-C'18)*. 207–214.

[168] S. Tahvili, L. Hatvani, M. Felderer, W. Afzal, and M. Bohlin. 2019. Automated functional dependency detection between test cases using Doc2Vec and clustering. In *Proceedings of the International Conference On Artificial Intelligence Testing (AITest'19)*. 19–26.

[169] V. Terragni, P. Salza, and F. Ferrucci. 2020. A container-based infrastructure for fuzzy-driven root causing of flaky tests. In *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'20)*. 69–72.

[170] S. Thorve, C. Sreshtha, and N. Meng. 2018. An empirical study of flaky tests in Android apps. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'18)*. 534–538.

[171] A. Vahabzadeh, A. A. Fard, and A. Mesbah. 2015. An empirical study of bugs in test code. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'15)*. 101–110.

[172] B. Vancsics, T. Gergely, and A. Beszédes. 2020. Simulating the effect of test flakiness on fault localization effectiveness. In *Proceedings of the International Workshop on Validation, Analysis and Evolution of Software Tests (VST'20)*. 28–35.

[173] B. Vaysburg, L. H. Tahat, and B. Korel. 2002. Dependence analysis in reduction of requirement-based test suites. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02)*. 107–111.

[174] S. Vysali, S. McIntosh, and B. Adams. 2020. Quantifying, characterizing, and mitigating flakily covered program elements. *Trans. Softw. Eng.* (2020). https://ieeexplore.ieee.org/document/9143477.

[175] M. Waterloo, S. Person, and S. Elbaum. 2015. Test analysis: searching for faults in tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*. 149–154.

[176] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam. 2021. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'21)*. 270–287.

[177] C. Wohlin. 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering (EASE'14)*. 1–10.

[178] W. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A survey on software fault localization. *Trans. Softw. Eng.* 42, 8 (2016), 707–740.

[179] H. Ye, M. Martinez, and M. Monperrus. 2021. Automated patch assessment for program repair at scale. *Empir. Softw. Eng.* 26, 2 (2021), 1–15.

[180] S. Yoo and M. Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliabil.* 22, 2 (2012), 67–120.

[181] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian. 2019. TERMINATOR: Better Automated UI test case prioritization. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 883–894.

[182] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *Trans. Softw. Eng.* 28, 2 (2002), 183–200.

[183] X. Zeng and T. R. Martinez. 2000. Distribution-Balanced stratified cross-validation for accuracy estimation. *J. Exper. Theoret. Artific. Intell.* 12, 1 (2000), 1–12.

[184] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi. 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *Proceedings of the International Conference on Software Engineering (ICSE'21)*. 50–61.

[185] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. 385–396.

[186] Y. Zhang, R. Jin, and Z. Zhou. 2010. Understanding Bag-Of-Words Model: A statistical framework. *Int. J. Mach. Learn. Cybernet.* 1, 1–4 (2010), 43–52.

[187] H. Zhu, L. Wei, M. Wen, Y. Liu, S. C. Cheung, Q. Sheng, and C. Zhou. 2020. MockSniffer: Characterizing and recommending mocking decisions for unit tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE'20)*. 436–447.

[188] C. Ziftci and D. Cavalcanti. 2020. De-Flake your tests automatically locating root causes of flaky tests in code at Google. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'20)*. 736–745.

[189] B. Zolfaghari, R. M. Parizi, G. Srivastava, and Y. Hailemariam. 2020. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Softw. Exper.* 51, 5 (2020), 851–867.

Received December 2020; revised May 2021; accepted July 2021