Evolution-Aware Detection of Order-Dependent Flaky Tests

Chengpeng Li
The University of Texas at Austin
Austin, TX, USA
chengpengli@utexas.edu

August Shi
The University of Texas at Austin
Austin, TX, USA
august@utexas.edu

ABSTRACT

Regression testing is an important part of the software development process but suffers from the presence of flaky tests. Flaky tests are tests that can nondeterministically pass or fail regardless of code changes. Order-dependent flaky tests are a prominent kind of flaky tests whose outcome depends on the test order in which they are run. Prior work has focused on detecting order-dependent flaky tests through rerunning all tests in different test orders on a single version of code. As code is constantly changing, rerunning all tests in different test orders after every change is costly.

In this work, we propose IncIDFlakies, a technique to detect order-dependent flaky tests by analyzing code changes to detect newly-introduced order-dependent flaky tests due to those changes. Building upon existing work in iDFlakies that reruns tests in different test orders, IncIDFlakies analyzes and selects to run only the tests that (1) are affected by the change, and (2) can potentially result in a test-order dependency among each other due to potential shared state. Running IncIDFlakies on 67 order-dependent flaky tests across changes in code in their respective projects, including the changes where they became flaky, we find that IncIDFlakies can select to run on average 65.4% of all the tests, resulting in running 68.4% of the time that baseline iDFlakies would use when running the same number of test orders with the full test suite. Furthermore, we find that IncIDFlakies can still ensure that the test orders it runs can potentially detect the order-dependent flaky tests.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

flaky test detection, order-dependent flaky test, evolution-aware analysis

ACM Reference Format:
Chengpeng Li and August Shi. 2022. Evolution-Aware Detection of Order-Dependent Flaky Tests. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534404

1 INTRODUCTION

Regression testing is an important part of software development but is plagued by the presence of flaky tests. Flaky tests are tests that nondeterministically pass or fail when run on the same version of code [32]. A flaky test failure can mislead developers into believing there is a fault introduced in the change, and they would waste time debugging the non-existent fault when the real problem is that the flaky test would have nondeterministically failed regardless of that change. Prior work has found several reasons for flaky tests, with common ones including asynchronous wait, concurrency, or test-order dependencies [12, 32].

An important kind of flaky tests are order-dependent flaky tests. *Order-dependent flaky tests* are flaky tests whose outcome depends on the test order in which they are run. Luo et al. found order-dependent flaky tests to be among the top three most common kinds of flaky tests in their empirical study [32]. Prior work has focused on developing techniques that detect order-dependent flaky tests through various means of rerunning the entire test suite in different test orders [24, 47] or analyzing the dependencies between all tests in the test suite [9, 16]. However, these techniques are expensive and analyze the tests through (re)running all tests on just a single version of software. For example, iDFlakies [24] reruns all tests in different, random test orders, detecting an order-dependent flaky test when it runs both a test order in which it passes and another test order in which it fails. iDFlakies tries all these different test orders on one version of software. A developer who aims to use iDFlakies to detect order-dependent flaky tests as soon as they are introduced would have to run iDFlakies after every change they make. Each iDFlakies run is independent of each other, and they do not use any information concerning changes between versions. Over time, using iDFlakies becomes expensive given the need to rerun all tests multiple times across frequent changes.

We propose IncIDFlakies, a technique to detect newly-introduced order-dependent flaky tests efficiently after a code change. IncIDFlakies builds upon iDFlakies, detecting order-dependent flaky tests by rerunning tests in random test orders. However, IncIDFlakies also takes into consideration the code changes since the last time IncIDFlakies was run, running test orders consisting of only the subset of tests that may become order-dependent flaky tests after the change. In general, an order-dependent flaky test can only fail when run in relation with some other tests. The order-dependent flaky test may fail when another test runs before it and "pollutes" their shared state, or the order-dependent flaky test may actually only pass when run after some other test that sets up the shared state properly for it [40]. A detection technique that runs tests to detect order-dependent flaky tests would need to run multiple tests in relation to each other. A change may induce a test to become an order-dependent flaky test if the change makes this test depend on some shared state that some other test may modify. Conversely, a

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534404

change may be such that a test starts modifying some shared state, therefore leading to some other test to become an order-dependent flaky test. In both scenarios, IncIDFlakies has to run at least these related tests to detect newly-introduced order-dependent flaky tests.

To select the necessary tests based on the changes, we leverage regression test selection (RTS) techniques to select tests that are affected by the change, meaning their behavior could differ after the change [18, 28, 46]. However, running just these affected tests is insufficient. It is possible that the change makes a test modify some state shared with other tests, or the change makes the test now depend on some shared state that other tests modify. These other tests' behavior may not differ after the change, so an RTS technique would (rightfully) not select them. A technique that aims to detect newly-introduced order-dependent flaky tests, however, would need to run those tests as to ensure the order-dependent flaky test can be detected. We use existing RTS techniques Ekstazi [17, 18] and STARTS [28, 29] to analyze changes, and we also augment the analysis to include additional tests that reach shared state used by the affected tests.

We evaluate IncIDFlakies on a dataset of 67 order-dependent flaky tests where prior work has already identified the code version/commit where the test became flaky, termed the flakiness-introducing commit [27]. These order-dependent flaky tests are spread across 14 open-source Java projects, and we evaluate on a total of 27 flakiness-introducing commits. For each flakiness-introducing commit, we additionally select up to five commits right before the flakiness-introducing commit, and we use the changes between the commits to evaluate how efficient IncIDFlakies would be at selecting and running the selected tests in different test orders upon analyzing those changes. We find that IncIDFlakies can on average run 65.4% of the tests while taking 68.4% of the time it would take iDFlakies to try the same number of test orders. Furthermore, we find that IncIDFlakies can successfully select the necessary tests to ensure it can run test orders that detect the newly-introduced order-dependent flaky test at the flakiness-introducing commit.

This paper makes the following contributions:

- **Evolution-aware flaky-test detection.** We develop the first technique that analyzes changes and code evolution to more efficiently detect order-dependent flaky tests. Our technique builds upon RTS techniques to select all the necessary tests to ensure order-dependent flaky tests can be detected through running in different test orders after a change.
- **Implementation.** We implement IncIDFlakies as an additional detector for the existing tool iDFlakies.
- **Evaluation.** Our evaluation of IncIDFlakies shows it can select the tests to detect all the order-dependent flaky tests from our dataset at the flakiness-introducing commits. Our evaluation shows that IncIDFlakies can provide substantial time savings, running 65.4% of tests and taking 68.4% of the time needed if running iDFlakies on the full test suite.

2 BACKGROUND

2.1 Order-Dependent Flaky Tests

An *order-dependent flaky test* is a flaky test whose outcome depends on the test order in which it is run [24, 32, 47]. An order-dependent flaky test must have both a passing test order (in which it passes) and a failing test order (in which it fails). Furthermore, the order-dependent flaky test must deterministically pass in the passing test order and fail in the failing test order, i.e., it is not a nondeterministic order-dependent flaky test [26].

Shi et al. [40] previously categorized order-dependent flaky tests into two types, victims and brittles. A *victim* fails when some other test, termed a *polluter*, runs before it in the test order. Conceptually, the polluter "pollutes" some state the two tests share, so the victim runs in an unexpected starting state and therefore fails. The victim passes when run on its own or before the polluter. Conversely, a *brittle* fails when run by itself, but it passes when another test, termed a *state-setter*, is run before it. Conceptually, the state-setter sets up the shared state properly for the brittle to start running in.

Figure 1 illustrates an example order-dependent flaky test taken from ktuukkan/marine-api, a real-world project we use in our evaluation. In this project, `testConstructor` (Line 23) is an order-dependent flaky test. `testConstructor` passes when it runs by itself. However, when `testRegisterParserWithAlternativeBeginChar` (Line 8) runs before, then `testConstructor` fails. In other words, `testConstructor` is a victim and `testRegisterParserWithAlternativeBeginChar` is its corresponding polluter. More specifically, the polluter unregisters `VDMParser.class` (Line 10) from the `parsers` map (Line 32) of the shared `SentenceFactory` instance obtained from calling `SentenceFactory.getInstance()` (Line 3). When the victim runs, it invokes `createParser()` (Line 20) to create a parser for `VDMParser.class`. However, if the polluter had run before and removed this parser from the `parsers` map, then executing `createParser()` would throw an `IllegalArgumentException` (Line 47), and so the victim would fail.

Note that the test code that tries to create the parser is actually part of an instance field declaration in the test class `AbstractAISMessageListenerTest`, meaning it is run when an instance of the test class is initialized, which, in JUnit, happens before every test method gets run. As such, `testConstructor` is not the only victim in this example but rather every test method in the test class `AbstractAISMessageListenerTest` is a victim as well.

2.2 Regression Test Selection (RTS)

When developers run tests after they make changes as part of their regression testing process, they can speed up the process by using regression test selection (RTS) to select and run a subset of the full test suite, namely the tests whose behavior could be affected by those changes [46]. In general, an RTS technique analyzes the dependency relationship between tests and code under test, creating a mapping from tests to code elements, such as statements, methods, or classes, that the tests depend on [46]. After determining which code elements have changed, an RTS technique would select to run the tests that depend on those elements; these tests are the *affected tests*. Prior work has found that tracking dependencies at the class level, i.e., mapping test classes to other classes that they depend on, is the most efficient at reducing the cost of regression testing [18, 28]. These findings have led to the creation of two open-source RTS tools for Java projects, Ekstazi [17] and STARTS [29].

Ekstazi. Ekstazi [17, 18] is a dynamic RTS technique. Ekstazi creates its class-level dependency mapping by instrumenting the code under test and tracking which classes each test class covers after

```
// existing test class that contains polluter
public class SentenceFactoryTest {
  private final SentenceFactory instance =
    SentenceFactory.getInstance();

  @Test
  public void
      testRegisterParserWithAlternativeBeginChar() {
    ...
    instance.unregisterParser(VDMParser.class);
  }
}

// newly-introduced test class with order-dependent test
public class AbstractAISMessageListenerTest {
  private final SentenceFactory sf =
    SentenceFactory.getInstance();

  private final AISSentence AIS_01 = (AISSentence)
    sf.createParser("VDM");

  @Test
  public void testConstructor() {
    ...
  }
}

// class of shared static field
public final class SentenceFactory {
  // map containing parser classes
  private static
    Map<String, Class<? extends SentenceParser>> parsers;

  public void unregisterParser
      (Class<? extends SentenceParser> parser) {
    for (String key : parsers.keySet()) {
      if (parsers.get(key) == parser) {
        parsers.remove(key);
        break;
      }
    }
  }

  public Sentence createParser(String nmea) {
    String sid = SentenceId.parseStr(nmea);
    if (!parsers.containsKey(type)) {
      throw new IllegalArgumentException(...);
    }
    return createParserImpl(sid, nmea);
  }
}
```

Figure 1: Example order-dependent flaky test from ktuukkan/marine-api

executing the tests. To track which classes have changed between versions, Ekstazi also records a checksum representation of the contents of each Java class file (a class in Java is represented as a class file on disk). After recompiling the code after a change, Ekstazi checks whether any new compiled class files now have a different checksum from what was recorded before, meaning the class has changed. Then, Ekstazi would select to run the test classes that depend upon the changed classes based on the dependency mapping collected from running tests on the previous version; these tests would be the affected tests. Ekstazi would also update the class file checksums and the dependency mapping with coverage information collected from executing the affected tests on this new version in preparation for the next change.

STARTS. STARTS [28, 29] is a static RTS technique. Similar to Ekstazi, STARTS also creates a class-level dependency mapping, from test classes to other classes each test class depends upon. However, STARTS creates this mapping by first constructing an intertype relation graph [35] of the code. In this graph, nodes are classes, and an edge exists between one node to another if a class either uses or inherits from that other class. This graph is constructed purely statically by analyzing the compiled class files in a project. STARTS then constructs a dependency mapping from test class to other classes by finding which other classes are reachable within this graph from each test class. To determine which classes have changed, STARTS uses the same checksum logic as employed by Ekstazi. Also like Ekstazi, STARTS determines affected tests as the test classes that are mapped to any changed classes in the dependency mapping. The key difference is that STARTS does not need to run any tests beforehand to create this dependency mapping.

3 INCIDFLAKIES

We present IncIDFlakies to detect order-dependent flaky tests after they are introduced in a change. The use scenario for IncIDFlakies is that developers would run the detection technique after their changes and have performed basic regression testing to check the correctness of their changes. The developer is already willing to use a flaky-test detection such as iDFlakies after changes, so IncIDFlakies aims to reduce that detection cost. In this scenario, developers would then already be using RTS techniques to speed up regression testing by running only the affected tests based on the changes. The developer would only use IncIDFlakies to detect newly-introduced order-dependent flaky tests after finishing running affected tests, and as such, IncIDFlakies can then reuse the results of the already run RTS technique, namely the affected tests and the dependency mapping information (Section 2.2). IncIDFlakies also takes as input the full test suite, as it needs to analyze which additional tests from the full test suite are necessary to also select as to ensure detecting newly-introduced order-dependent flaky tests. IncIDFlakies would then run its selected tests in different test orders to detect order-dependent flaky tests.

We next describe how IncIDFlakies selects the tests to run as well as how IncIDFlakies then runs the tests to detect any newly-introduced order-dependent flaky tests.

3.1 Selection

IncIDFlakies starts with the affected tests provided by the RTS technique that ran previously. These affected tests are those whose behavior could differ based on the changes, which means they could now be modifying some state shared with other tests in the test suite (i.e., they become polluters or state-setters), or they could now depend on some shared state that is modified by some other tests in the test suite (i.e., they become victims or brittles). As such, these tests must be run to detect whether any test orders involving them result in an order-dependent flaky test failure.

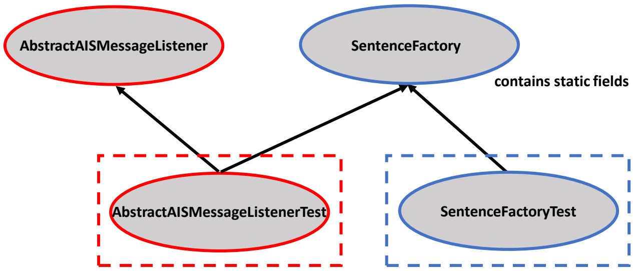

Figure 2: Graph representation of dependencies between classes from ktuukkan/marine-api

However, just running the selected tests based on the RTS tool is not enough to ensure detection of newly-introduced order-dependent flaky tests. Consider once again the example in Figure 1. The example shows code and tests at commit `3d8dff0f`. Before this commit, the polluter already existed in the test suite. While it was always unregistering the `VDMParser.class` from the shared `SentenceFactory` instance, no other tests actually relied on that instance. As such, no other test would be found to be an order-dependent flaky test in relation to the polluter. However, this commit introduced a whole new test class `AbstractAISMessageListenerTest`, and its initialization depends on this shared `SentenceFactory` instance. As such, tests in this test class are victims from the very beginning, upon introduction into the test suite. Furthermore, this commit only introduced the new test class and modified the class it tests. Figure 2 illustrates how `AbstractAISMessageListenerTest` and `AbstractAISMessageListener` are the only modified classes (nodes with red outline), and only `AbstractAISMessageListenerTest` is an affected test class (red-dashed rectangle). No other test classes would be affected by this change (no dependency edge to the modified classes). An RTS technique would (rightfully) only select tests in `AbstractAISMessageListenerTest` as affected tests, because no other test should behave differently after this change. Even though the newly-introduced tests are indeed order-dependent flaky tests due to their relation with another existing test (in `SentenceFactoryTest`), it would not be detected as such if not run alongside that other test.

To address this issue, we introduce analysis to select additional tests that are related to the affected tests and should then be run together to ensure any newly-introduced order-dependent flaky test can be detected. Figure 3 shows the process. We start with the affected tests provided by the RTS technique. For each affected test, we use the mapping to find all its dependency classes (Line 13). We analyze these dependency classes for potential sources of shared state, namely static fields. While shared state can come in variety of forms such as static fields, files, database connections, etc., we focus on just static fields, as prior work found them to be a common source of state shared between tests in Java [8, 20, 31, 32, 47].

For each dependency class, we obtain the static fields defined in the class, if any (Line 21). If the dependency class contains static fields, that means the affected test could modify or use any state reachable from the static field that could also be shared with other tests. As such, we need to include other tests that also can reach this dependency class. The affected test's behavior could be changed such that it modifies this shared state these other tests depend on, or the affected test could now be depending on this shared state that some other test is already modifying (e.g., the example in Figure 1). We construct a reverse dependency mapping going from classes to tests that depend on each class (Line 9). We use that mapping to find all tests that reach this dependency class with static fields, to be included as the additional tests needed to run (Line 27).

```
1  # Input: tests : all tests at the current sha
2  #        affected_tests : affected tests from RTS
3  #        deps : test to dependencies mapping from RTS
4  # Output: tests to run in different test orders
5  def IncIDFlakies(tests, affected_tests, deps):
6    additional_tests = set()
7    processed_deps = set()
8    # get the dependencies from class to tests
9    reverse_deps = get_reverse_dependencies(deps)
10   for test in affected_tests:
11     if not test in deps.keys():
12       continue
13     dependencies = deps[test]
14     for dp in dependencies:
15       # process each dependency once
16       if dp in processed_deps:
17         continue
18       if in_third_party_library(dp):
19         continue
20       processed_deps.add(dp)
21       static_fields = get_static_fields(dp)
22       for field in static_fields:
23         # exclude final fields that are immutable
24         if is_final(field) and is_immutable(field):
25           continue
26         # include all tests that reach this class
27         additional_tests |= reverse_deps[dp]
28         break
29
30   return affected_tests | additional_tests
```

Figure 3: Process for selecting additional tests to run

Consider Figure 2 again. IncIDFlakies would start with `AbstractAISMessageListenerTest` and find all its dependency classes, eventually finding that `SentenceFactory` has a static field that is not final and immutable. The process then finds all the test classes that depend on `SentenceFactory`, namely test classes `AbstractAISMessageListenerTest` and `SentenceFactoryTest`, including all of them to be run. As such, `SentenceFactoryTest` is an additional test class to be selected (blue dashed rectangle). Running both test classes together could result in detecting the newly-introduced order-dependent flaky tests in `AbstractAISMessageListenerTest`.

As a heuristic to avoid selecting too many additional tests, we ignore dependency classes that are in third-party libraries beyond the code in the project itself (Line 18). We want to focus on shared state within the code that the developers have control over. In particular, we do not want to track dependencies within the Java standard library, as many classes could depend on some few Java classes (as found in prior work on RTS [28]). While ignoring potential dependencies on third-party library classes can result in missing to select relevant tests, we believe there is only a small chance that tests may affect each other's behavior through state shared from static fields in these external classes.

As another optimization, we also check whether static fields contained in a dependency class are final and are of an immutable type, such as `java.lang.String`, `java.util.regex.Pattern`, or a primitive type. We ignore any such static fields (Line 24). A final field value cannot be changed after its initial assignment, and an immutable object's state cannot change, so tests cannot modify shared state through these fields. If a dependency class only contains such static fields, we would skip processing that class altogether.

3.2 Running Selected Tests

Once IncIDFlakies has selected the tests to run based on the changes, it will next run them in different test orders for detecting any newly-introduced order-dependent flaky tests. We build this part of IncIDFlakies on top of iDFlakies [1, 24], an existing tool that detects order-dependent flaky tests in a single version of software by running all the tests in different, random test orders. While effective at detecting order-dependent flaky tests, iDFlakies would be costly to deploy as part of a continuous integration process given that it would need to run many different test orders after every change. IncIDFlakies aims to improve upon iDFlakies by running fewer tests after every change.

Since we are building upon iDFlakies, we inherit many of the same configuration options as well. The most relevant configuration option is the number of rounds to run. In iDFlakies, a *round* represents a test order that iDFlakies would run the tests in. If any tests have a different outcome when run in this round than the outcomes recorded from an initial so-called original test order, iDFlakies would mark the test as flaky. It would also rerun that same test order to check whether it maintains the same outcome, and if so, that detected flaky test would also be categorized as order-dependent. The more rounds that iDFlakies is configured with then the higher the chance of detecting any order-dependent flaky tests (since iDFlakies would have tried more test orders). However, more rounds means also a higher running cost.

In the case of IncIDFlakies, the expected use scenario is that developers would choose a number of rounds to run IncIDFlakies with, and IncIDFlakies would only run the selected tests that many rounds, reporting any detected order-dependent flaky tests afterwards. While some extra analysis is needed to determine which additional tests to select, the expectation is that running fewer tests per round would overall decrease the running cost for detecting order-dependent flaky tests after every change. Furthermore, with fewer tests, there would be fewer permutations of test orders overall, so IncIDFlakies would likely cover more relative orderings between the relevant tests in the limited number of rounds.

4 EXPERIMENTAL SETUP

4.1 Subjects

To evaluate IncIDFlakies, we use the order-dependent flaky tests contained with IDoFT [2], a public dataset of flaky tests collected across open-source Java projects on GitHub. These flaky tests were detected using automated flaky-test detection tools such as iDFlakies [1] or NonDex [4]. In addition, Lam et al. [27] previously conducted a longitudinal study on flaky tests within this dataset, where they determined the commit in the flaky test's project's history where the flakiness was introduced, termed its *flakiness-introducing commit*. They marked these flakiness-introducing commits for a subset of tests in IDoFT. Given that IncIDFlakies specifically targets order-dependent flaky tests, we start with the 104 tests in the dataset that are both marked as order-dependent and have a corresponding flakiness-introducing commit. However, 9 of these tests fall into the category of what IDoFT considers non-order-dependent or nondeterministic order-dependent [26], which means the tests do not deterministically pass or fail in a specific test order; we ignore such tests from our evaluation.

Table 1: Filtering tests from IDoFT

# tests	Category
104	initial order-dependent flaky tests
9	denoted as non-order-dependent flaky test
20	could not compile
8	cannot reproduce flakiness
67	**final # tests**

To determine whether IncIDFlakies can run the proper tests to detect order-dependent flaky tests at the flakiness-introducing commit, we need to first find possible passing and failing test orders for each order-dependent flaky test. First, we found we could no longer compile or run tests at the flakiness-introducing commit for one of the projects (Activiti/Activiti), and so we excluded their 20 tests. Next, we ran iDFlakies on each of the flakiness-introducing commits for 100 rounds as to detect both such test orders for each of the remaining order-dependent flaky tests. Unfortunately, we could not detect 8 order-dependent flaky tests at their flakiness-introducing commits even after running iDFlakies for this large number of rounds. From our inspection, we found that three of them likely have their flakiness-introducing commit marked wrong in the dataset, one cannot be run using the current iDFlakies tooling given their use of a specialized test runner, and the remaining four are likely non-order-dependent flaky tests. Overall, our evaluation dataset consists of 67 order-dependent flaky tests from 14 projects, consisting of 21 modules. Table 1 shows a breakdown of our filtering process to obtain those 67 order-dependent flaky tests.

These 67 order-dependent flaky tests are also spread across 27 flakiness-introducing commits (order-dependent flaky tests of the same project/module may share the same flakiness-introducing commit). We confirmed that these commits are indeed flakiness-introducing commits for each test by also running iDFlakies on the commit right before as to check the test is not detected as flaky on the previous commit. There was just one flakiness-introducing commit from the IDoFT dataset where we found the corresponding test was still flaky on the previous commit, so we went back further in that test's history to find an earlier commit that constitutes the flakiness-introducing commit (which matched with the flakiness-introducing commit of another order-dependent flaky test in the dataset). Table 2 provides the details of these projects/modules and the flakiness-introducing commits. In the table, each row represents the flakiness-introducing commit for a project and module, where

Table 2: Projects/modules, commit SHAs, and # order-dependent flaky tests used in evaluation

ID	SLUG	Module	SHA	# OD Tests
V1	apache/incubator-dubbo	dubbo-cluster	fcd1af81	2
V2	apache/incubator-dubbo	dubbo-common	3c6201cf	1
V3	apache/incubator-dubbo	dubbo-config/dubbo-config-api	46c6ccb3	1
V4	apache/incubator-dubbo	dubbo-config/dubbo-config-spring	5f5fecd6	2
V5	apache/incubator-dubbo	dubbo-filter/dubbo-filter-cache	9f7306f9	1
V6	apache/incubator-dubbo	dubbo-rpc/dubbo-rpc-api	b055991b	2
V7	apache/incubator-dubbo	dubbo-serialization/dubbo-serialization-fst	67843758	1
V8	ctco/cukes	cukes-http	216a5da6	1
V9	dropwizard/dropwizard	dropwizard-logging	416bdaaf	1
V10	elasticjob/elastic-job-lite	elastic-job-lite/elastic-job-lite-core	7e2505bc	1
V11	elasticjob/elastic-job-lite	elastic-job-lite/elastic-job-lite-core	a000cd2a	1
V12	elasticjob/elastic-job-lite	elastic-job-lite/elastic-job-lite-core	c808c8e7	2
V13	fhoeben/hsac-fitnesse-fixtures	.	faff990c	1
V14	hexagonframework/spring-data-ebean	.	2d3515b0	1
V15	kevinsawicki/http-request	lib	cb9e021f	1
V16	ktuukkan/marine-api	.	3d8dff0f	9
V17	ktuukkan/marine-api	.	b4afeed8	3
V18	openpojo/openpojo	.	2ab65a57	1
V19	openpojo/openpojo	.	4d69e2cf	1
V20	openpojo/openpojo	.	92186a6f	1
V21	openpojo/openpojo	.	a22bb1c5	2
V22	querydsl/querydsl	querydsl-hibernate-search	2c195ffa	1
V23	tbsalling/aismessages	.	12864f81	2
V24	wikidata/wikidata-toolkit	wdtk-dumpfiles	f7cb4087	3
V25	wikidata/wikidata-toolkit	wdtk-util	539f5223	2
V26	wildfly/wildfly	naming	c22e2311	22
V27	zalando/riptide	riptide-spring-boot-starter	08442067	1
Overall				**67**

the project GitHub SLUG is presented under column "SLUG", the module name is presented under column "Module", and the commit SHA is presented under column "SHA". Each flakiness-introducing commit is given an ID for reuse in later tables. We also present in the table the number of order-dependent flaky tests out of the total 67 associated with each flakiness-introducing commit.

Collecting additional commits. To give a better sense of how many tests IncIDFlakies would select to run and how much time it would take, we collect additional commits before each flakiness-introducing commit where we would run IncIDFlakies. The use scenario is that a developer would run a flaky-test detection technique after all changes without knowing whether there would be a flaky test introduced or not. As such, we should evaluate the potential time savings IncIDFlakies provides over baseline iDFlakies even on commits where there would not be flaky tests detected.

Starting from each flakiness-introducing commit, we go back in the Git history one commit at a time, and we include that commit in our evaluation if there is a change in Java source code from either the module that contains the order-dependent flaky test or any modules it depends on. We restrict to such commits because the RTS techniques we use, Ekstazi and STARTS, focus on Java code changes. As such, they would not do any selection if there are non-code changes. We collect up to five commits prior to each flakiness-introducing commit that satisfy our criteria (we could only collect four commits for V8 that satisfy our criteria).

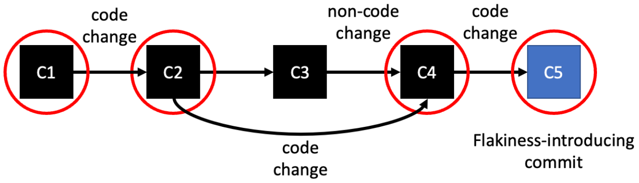

Figure 4: The process of collecting additional commits

Figure 4 illustrates how we collect these additional commits per flakiness-introducing commit. In the figure, commit C5 is the flakiness-introducing commit, the latest commit considered, and commits C1–C4 are immediately before C5 in the Git history. There are code changes between C4 and C5, so we use C4 in the evaluation (circled in the figure). However, there are non-code changes between C3 and C4, so we do not include C3. Since there are code changes between C2 and C4, we then also include C2 for evaluation.

4.2 Running IncIDFlakies

For each commit we use in our evaluation, we use each RTS technique to analyze the changes between that commit and the subsequent commit we collected, obtaining both the set of affected tests and the dependency mapping from tests to classes provided by the RTS techniques. This part simulates our expected usage for IncIDFlakies where a developer would have already used RTS as part of their regression testing process, and IncIDFlakies as a flaky-test detection tool can reuse that information. We then run IncIDFlakies using the affected tests and dependency mapping provided by the RTS technique on this subsequent commit. We denote IncIDFlakies that uses Ekstazi as *IncIDFlakies*$_E$, and IncIDFlakies that uses STARTS is denoted as *IncIDFlakies*$_S$.

On each commit where we run IncIDFlakies, we configure it to use 10 rounds, as suggested by previous work on iDFlakies [24]. On each of these commits, we also run iDFlakies, which does not analyze any changes and simply runs with the entire test suite; we also use the same 10 rounds configuration so it can be a fair baseline for comparison against IncIDFlakies. We measure what percentage of all the tests are selected to run by IncIDFlakies and how much time it takes for IncIDFlakies to run compared against iDFlakies for the same number of rounds. In both cases, a smaller percentage is better, as it would indicate IncIDFlakies provides savings in tests to run and time to run them. Note that the time we measure for running IncIDFlakies includes both the analysis time for determining the additional tests to also select and also the time to run all the selected tests for 10 rounds.

Another configuration detail for our experiments is that we disable the step where iDFlakies reruns test orders where it discovers a difference in test outcome as to check whether the test is an actual order-dependent flaky test [1, 24]. Since iDFlakies and IncIDFlakies can be running different tests at the same commit, they may run different test orders that may detect different sets of flaky tests. As such, their runtimes can differ drastically if one detects more flaky tests and has to rerun more times. To allow for fairer comparison of runtime differences, we disable the rerun step in both iDFlakies and IncIDFlakies, ensuring they always run the exact same number of rounds without any additional test runs.

4.3 Potential for Detection

Given that the order-dependent flaky test flakiness only first manifests at the flakiness-introducing commit, we evaluate how effective IncIDFlakies is at detecting the order-dependent flaky tests only on the flakiness-introducing commit. We compare how many of the order-dependent flaky tests are detected by IncIDFlakies compared against those detected by baseline iDFlakies within the same 10 rounds, with the higher percentage the better.

However, given the nondeterministic nature of running different test orders, it is not guaranteed that every order-dependent flaky test would be detected in the 10 rounds we use for our evaluation. As such, we need a metric for determining whether it is possible for IncIDFlakies to *potentially* detect the order-dependent flaky test if it was to run long enough (iDFlakies would always be guaranteed to detect the test given that it is running the full test suite).

An order-dependent flaky test would only be detected if running in different test orders reveals a passing test order and a failing test order. In other words, if the order-dependent flaky test is a victim, then it needs to be run with at least one of its polluters; if the order-dependent flaky test is a brittle, then it needs to be run with at least one of its state-setters. As such, we say the technique can potentially detect the order-dependent flaky test at the flakiness-introducing commit if it selects both the order-dependent flaky test and at least one of its corresponding polluters/state-setters.

For each order-dependent flaky test, by definition, we will already have one passing test order and one failing test order at the flakiness-introducing commit. We run iFixFlakies [40] on the order-dependent flaky test using those test orders. iFixFlakies can minimize the test orders and be configured to find all the possible polluters and state-setters. We record the polluters and state-setters, and we use that information to determine whether a technique can actually detect the order-dependent flaky test at the flakiness-introducing commit.

5 EVALUATION

We address the following research questions:

- **RQ1:** How efficient is running IncIDFlakies compared against baseline iDFlakies?
- **RQ2:** How effective is IncIDFlakies at detecting the newly-introduced order-dependent flaky tests?

We address RQ1 to check whether it is practical to use IncIDFlakies compared against just running iDFlakies after every change. We address RQ2 to check whether using IncIDFlakies can detect a newly-introduced order-dependent flaky test by selecting both the test itself and the relevant other tests such that the order-dependent flaky test can be detected when run in different test orders.

5.1 RQ1: Efficiency of Running IncIDFlakies

Table 3 shows results concerning efficiency of IncIDFlakies compared against baseline iDFlakies when run across all commits used in our evaluation. In the table, we show under column "iDFlakies" the average number of tests across all commits (which is the number of tests that iDFlakies would run) along with the average time in seconds to run iDFlakies configured for 10 rounds across those commits; each row represents commits collected relative to the corresponding flakiness-introducing commit (Section 4.1). The final row shows the average number of tests and the time to run across all commits across all projects.

In the table, we show the average percentages of tests selected by IncIDFlakies$_E$ and IncIDFlakies$_S$ across all commits w.r.t. the baseline all tests that would be run by iDFlakies. We also show the percentage of time IncIDFlakies$_E$ and IncIDFlakies$_S$ would take to both analyze and select tests to run along with running those tests the same number of rounds as baseline iDFlakies. We see that IncIDFlakies allows for savings in both number of tests to run and time to run and detect order-dependent flaky tests. On average, IncIDFlakies$_E$ selects 65.4% of the tests while taking only 68.4% of the time for running iDFlakies. IncIDFlakies$_S$ has a similar reduction, selecting 70.0% of the tests while taking only 70.5% of the time for running iDFlakies.

We also compare the results of IncIDFlakies against running only the tests selected by the RTS techniques Ekstazi and STARTS. These tests are just the affected tests based on the changes and would

Table 3: IncIDFlakies tests selected and time to run

	iDFlakies		IncIDFlakies$_E$		IncIDFlakies$_S$		Ekstazi		STARTS	
ID	# Tests	Time (s)	% Sele	% Time	% Sele	% Time	% Sele	% Time	% Sele	% Time
V1	108.0	55.1	94.4	97.6	100.0	98.4	37.7	32.9	92.0	96.0
V2	287.2	142.6	39.3	68.1	49.9	75.0	3.3	26.8	3.7	26.9
V3	35.0	54.6	60.0	60.3	40.0	39.9	48.7	49.2	40.0	39.7
V4	62.4	61.9	22.7	61.1	58.1	61.2	14.3	54.4	47.8	62.8
V5	2.8	12.4	80.0	77.4	80.0	77.7	80.0	77.1	80.0	77.6
V6	52.8	49.3	63.1	43.0	77.6	58.2	39.0	38.5	39.3	32.9
V7	61.8	6.0	40.0	39.1	60.0	57.5	40.0	40.0	60.0	57.4
V8	8.5	10.3	50.0	50.6	50.0	48.1	50.0	52.0	50.0	48.2
V9	74.4	36.6	64.9	74.5	66.9	75.7	37.6	66.5	52.3	71.6
V10	309.2	82.4	74.1	36.1	88.6	36.3	40.0	33.4	63.0	36.1
V11	310.2	122.2	71.8	36.2	83.6	37.1	24.1	27.4	28.8	30.8
V12	308.2	121.8	72.6	36.4	83.5	37.2	34.4	28.9	42.1	32.7
V13	174.8	13.1	10.3	18.3	62.8	91.6	3.2	15.4	55.1	90.8
V14	20.8	18.5	80.0	76.9	80.0	78.3	72.9	76.6	61.6	76.3
V15	150.4	12.5	80.0	73.6	80.0	73.1	79.7	74.1	79.7	74.0
V16	716.4	18.3	21.8	67.0	23.5	67.3	2.2	25.1	2.2	24.9
V17	693.6	17.7	30.7	64.2	32.1	64.5	19.5	38.8	23.3	54.4
V18	1156.2	338.0	97.5	99.8	97.5	99.4	34.9	89.8	58.9	75.0
V19	1117.6	323.5	78.5	79.5	78.5	79.5	66.0	77.4	78.3	79.6
V20	1052.4	48.7	97.3	97.4	97.3	97.4	3.8	24.4	41.0	54.8
V21	1124.0	330.6	97.5	97.5	97.5	101.0	37.9	90.8	59.5	63.0
V22	13.0	27.2	80.0	82.1	80.0	79.1	80.0	80.0	80.0	78.8
V23	19.8	5.1	80.0	76.1	80.0	74.6	80.0	76.8	80.0	75.7
V24	62.2	31.7	72.6	100.5	72.6	96.5	22.9	29.2	22.9	27.8
V25	11.6	9.0	20.7	28.7	20.7	29.1	20.7	29.1	20.7	28.8
V26	82.0	19.2	100.0	99.7	100.0	98.8	100.0	100.3	100.0	99.0
V27	55.0	93.5	81.9	102.3	44.2	65.9	55.7	84.5	30.7	43.4
Average	**301.1**	**76.8**	**65.4**	**68.4**	**70.0**	**70.5**	**41.7**	**53.3**	**51.6**	**57.8**

not include the additional tests IncIDFlakies would select based on their relation to the affected tests w.r.t. static fields. Table 3 shows the percentage of tests selected by Ekstazi and STARTS (under the columns labeled by the technique). The table also shows the time to run the same 10 rounds for just the affected tests relative to the time to run iDFlakies on the full test suite (note that this time does not include any time to analyze to select additional tests, it is just the time run Ekstazi/STARTS's affected tests). On average, Ekstazi selects 41.7% of the tests while taking only 53.3% of the time needed for iDFlakies, while STARTS selects 51.6% of the tests while taking only 57.8% of the time. In general, STARTS selects more tests than Ekstazi, which is expected given that the static analysis STARTS uses tries to over-approximate which tests are affected by the changes. However, there are still some commits where STARTS would select fewer tests than Ekstazi, which can occur due to dynamic dependencies between tests and classes, such as through reflection [28, 39], which a static technique such as STARTS would not track. Ekstazi, however, would track them.

Overall, our results show that a large number of additional tests would be selected due to their relation to the affected tests via shared static fields. While IncIDFlakies does require a nontrivial amount of additional time to run with more tests, we see later (Section 5.2) that this additional time is necessary as to ensure the newly-introduced order-dependent flaky tests can be detected.

> ***RQ1:*** *Overall, IncIDFlakies outperforms iDFlakies, providing a reduction in both tests selected and time to run. Specifically, IncIDFlakies$_E$ selects 65.4% of the tests and take 68.4% of the time iDFlakies needs; IncIDFlakies$_S$ selects 70.0% of the tests and takes 70.5% of the time iDFlakies.*

5.2 RQ2: Detecting Order-Dependent Flaky Tests

Table 4 shows the detection results from running IncIDFlakies across the 27 flakiness-introducing commits (note that we only evaluate on the flakiness-introducing commits here because these are the commits where a technique could actually detect the flaky test). We show the percentage of the known order-dependent flaky tests (total number of order-dependent flaky tests shown in Table 2) at each flakiness-introducing commit that can potentially be detected by IncIDFlakies, shown under columns marked "% P", meaning both it and any of its corresponding polluters or state-setters are part

Table 4: Order-dependent flaky tests detected using IncIDFlakies

	iDFlakies	**IncIDFlakies$_E$**		**IncIDFlakies$_S$**		**Ekstazi**		**STARTS**	
ID	% D	% P	% D	% P	% D	% P	% D	% P	% D
V1	100.0	100.0	0.0	100.0	100.0	0.0	0.0	100.0	100.0
V2	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V3	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V4	100.0	100.0	100.0	100.0	100.0	100.0	100.0	50.0	50.0
V5	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V6	50.0	100.0	50.0	100.0	50.0	100.0	100.0	100.0	50.0
V7	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V8	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V9	0.0	100.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0
V10	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V11	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V12	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
V13	100.0	100.0	100.0	100.0	0.0	100.0	100.0	100.0	100.0
V14	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0	0.0
V15	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V16	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0	0.0
V17	100.0	100.0	0.0	100.0	0.0	0.0	0.0	0.0	0.0
V18	100.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
V19	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V20	100.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
V21	50.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0	0.0
V22	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V23	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V24	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V25	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V26	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
V27	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Average	**91.0**	**100.0**	**85.1**	**100.0**	**73.1**	**71.6**	**67.2**	**73.1**	**67.2**

of the selected tests (Section 4.3). Overall, we see that IncIDFlakies can select the tests necessary to detect *all* 67 order-dependent flaky tests, for both IncIDFlakies$_E$ and IncIDFlakies$_S$.

In contrast, we also show the percentage of order-dependent flaky tests that can be detected by using just the affected tests from RTS techniques Ekstazi and STARTS. Use of Ekstazi can only detect 71.6% of the order-dependent flaky tests, and STARTS can only detect 73.1% of the order-dependent flaky tests. We also find that, on average, Ekstazi's affected tests only include 70.6% of all the polluters/state-setters per order-dependent flaky test; the percentage is 73.1% for STARTS's affected tests. Meanwhile, IncIDFlakies$_E$ and IncIDFlakies$_S$ both end up selecting all the polluters/state-setters per order-dependent flaky test (though selecting all polluters and state-setters is not guaranteed). These results show the importance of IncIDFlakies's extra analysis to select the additional tests related to the affected tests, as to ensure the order-dependent flaky tests can be detected from running in different test orders.

Besides determining whether it is possible to detect the order-dependent flaky tests in different test orders, we also measure whether running those tests in 10 rounds actually detects the order-dependent flaky test. In Table 4, we also show under columns "% D" the percentage of known order-dependent flaky tests that were actually detected within the 10 rounds. We also show this percentage for the baseline iDFlakies, because, given the random nature of using a limited number of different test orders, it may not be possible to detect all the order-dependent flaky tests. Indeed, iDFlakies that runs on the full test suite detects 91.0% of the order-dependent flaky tests within 10 rounds. Note that we only measure the percentage of the known 67 order-dependent flaky tests detected in our evaluation; we do not consider additional order-dependent flaky tests detected across the rounds.

IncIDFlakies also does not detect all the order-dependent flaky tests, detecting 85.1% and 73.1% for IncIDFlakies$_E$ and IncIDFlakies$_S$, respectively. We also see that running just the tests selected by Ekstazi or STARTS for the same number of rounds results in even fewer order-dependent flaky tests detected, which is expected given the order-dependent flaky tests that they cannot possibly detect (e.g., the example in Figure 1). While IncIDFlakies does not detect as many order-dependent flaky tests as baseline iDFlakies in the same number of rounds, they are still able to detect most of the order-dependent flaky tests within a shorter amount of time compared against iDFlakies (Section 5.1). Given some more resources (or different seeds for random number generation), they could still potentially detect all newly-introduced order-dependent flaky tests.

Indeed, when we inspect the cases where IncIDFlakies did not detect order-dependent flaky tests within the 10 rounds, we confirm that, if we were to run the selected tests for more rounds (up to 100), IncIDFlakies could actually detect all order-dependent flaky tests. In all these cases, we found the order-dependent flaky tests would each have many "cleaners". While an order-dependent flaky test normally fails when run after a polluter, if there is another test, called a cleaner, that runs in-between, the order-dependent flaky test would pass [40]. Large number of cleaners reduces the number of test orders that can fail, which explains why it is so difficult to induce a failing test order, even with fewer tests to run overall.

RQ2: *IncIDFlakies can potentially detect all newly-introduced order-dependent flaky tests from our evaluation at the flakiness-introducing commit by selecting the order-dependent flaky tests along with at least one corresponding polluter/state-setter.*

5.3 Discussion

Finer-grained dependency analysis. Currently, we rely on the dependencies provided by Ekstazi and STARTS, which both collect class-level test dependencies. Collecting test dependencies at a finer granularity, such as at the method level, could result in more precisely identifying the relationship between tests and static fields, resulting in fewer tests to run. In our preliminary work, we used Soot [5] to compute a method-level call graph connecting tests to static fields. Starting with the initial affected tests from an RTS technique, we would find the additional tests by navigating this more precise call graph. For example, when we used this call graph to select additional tests on top of the STARTS affected tests, the average percentage of tests to run across all commits is 67.3% (vs. the 70.0% using STARTS' class-level dependencies). However, the combination of analysis and running selected tests took on average 154.8% of the time baseline iDFlakies would take across commits. In other words, using the finer-grained analysis resulted in higher runtime than the baseline. Future work for more precise analysis should focus on improving its cost.

Polluted shared state. Recall that IncIDFlakies employs a heuristic that ignores dependencies on classes in third-party libraries (Section 3.1). Using ODRepair [31], a technique that can identify shared static fields for order-dependent flaky tests, we find two order-dependent flaky tests where the relevant shared static field is found in third-party library code. We also find one order-dependent flaky test where the shared static field likely is from third-party library code (but not confirmed through ODRepair).

Interestingly, we see several cases where we believe the polluted shared state is not due to static fields. For example, the tests in wildfly/wildfly seem to depend on shared state accessible using the JNDI service [3], not through static fields. However, the RTS technique selects all the tests due to the change. In general, the combination of both RTS and the analysis for determining additional tests helps with selecting the necessary tests to detect newly-introduced order-dependent flaky tests (Section 5.2). The over-approximation from the coarser-grained analysis of class-level dependencies, while selecting more tests, helps with increasing the chance of selecting all relevant tests. In the future, we can extend the analysis to consider not just static fields but also other code elements that indicate using different kinds of shared state, e.g., code that accesses files.

6 THREATS TO VALIDITY

The results of our evaluation may not generalize to other projects. Our evaluation dataset was obtained from prior work that evaluated on a large dataset of order-dependent flaky tests [27]. These order-dependent flaky tests are from popular open-source Java projects, spread across a wide range of application domains.

The flakiness-introducing commits we use in our evaluation have the risk of not being the exact commit where the test is flaky. We obtained these flakiness-introducing commits from prior work [27], where they started from the commit where the flaky test was introduced and worked their way forward while running iDFlakies to find which commit is the flakiness-introducing commit. We also confirm that the tests are indeed flaky at the flakiness-introducing commit by obtaining both a passing and failing test order there. Further, we check that the order-dependent flaky tests are not flaky on the commit before the flakiness-introducing commit by running iDFlakies on that prior commit for 100 rounds without obtaining both a passing and failing test order.

Our implementation may contain bugs. To mitigate this threat, we build IncIDFlakies upon existing tools Ekstazi, STARTS, and iDFlakies. These tools have been utilized extensively in research [18, 19, 24–28, 30, 40, 42, 48]. For the newer analysis parts, we reviewed code and execution logs to confirm the implementation correctness.

7 RELATED WORK

Flaky tests. Luo et al. conducted the first empirical study on flaky tests in open-source projects [32]. They studied the common root causes for fixed flaky tests along with how they could manifest and how developers would fix them. Eck et al. conducted a similar study with a focus on the developers' perspectives [12]. Flaky tests are also prevalent in industry, to the point that researchers at Facebook propose everyone to "assume all tests are flaky" [21].

To help with the issue of flaky tests, there has been numerous work in detecting different types of flaky tests, such as detecting order-dependent flaky tests [16, 24, 45, 47], flaky tests that make assumptions on determinism in specifications [38], flaky tests that depend on random number generation [11], or time-constrained flaky tests [43]. Our work focuses on detecting order-dependent flaky tests, but unlike prior work that detects through analysis on a single code version, we consider the changes between code versions. There has also been work in detecting flaky tests without having to rerun tests but rather through machine learning techniques [6]. However, there is risk that tests classified as flaky are actually not. Our work builds upon iDFlakies to detect flaky tests through reruns [24], ensuring detected flaky tests are actually flaky, but doing so more efficiently by leveraging code evolution. Bell et al. [10] proposed DeFlaker, which detects flaky tests by checking whether a test has a different outcome after a change yet does not cover that change. Such a test must be flaky since its outcome is not related to the change. However, DeFlaker can only detect flaky tests if it so happens to have a different outcome after a change, unlike our work that reruns tests in different test orders to more easily manifest flakiness. Indeed, Bell et al. found that flakiness manifests more easily if they would change up the running environment.

Order-dependent flaky tests. Luo et al. found that order-dependent flaky tests are one of the top three kinds of flaky tests [32]. Zhang et al. [47] developed DTDetector to detect order-dependent flaky tests proactively by running tests in random test orders. Lam et al. [24] followed up with iDFlakies to both detect and partially classify flaky tests as order-dependent or non-order-dependent flaky tests. We build upon iDFlakies to make it evolution-aware and run more efficiently by only selecting the necessary tests between code versions. Our evolution-aware analysis could potentially be applied to other techniques that aim to detect order-dependent flaky tests [16, 45], but across multiple code versions. Lam et al. [27] also used iDFlakies in their longitudinal study to find the flakiness-introducing commit for known order-dependent flaky tests. They start with the commit that first introduced the test into the project and then moved forward through commits until reaching one where using iDFlakies detects the order-dependent flaky test (which could be at the test-introducing commit). We reuse their dataset for our work on detecting such tests at the flakiness-introducing commit.

Huo and Clause [23] proposed OraclePolish to detect tests with assertions that depend on input data not controlled by the tests themselves, meaning they could potentially fail when run by themselves (these assertions are "brittle"). Gyori et al. [20] developed PolDet to find tests that could "pollute" the state for other tests (which may not even exist in the test suite yet) by tracking whether shared state via static fields have changed between when a test starts and when a test finishes. Shi et al. [40] would define specific tests related to order-dependent flaky tests (with terms "brittle" and "polluter" inspired by previous work) and developed iFixFlakies to find the polluters and state-setters for order-dependent flaky tests. iFixFlakies would ultimately propose patches for these order-dependent flaky tests by leveraging code from the existing test suite. We use iFixFlakies to find all polluters and state-setters to help with our evaluation. Li et al. [31] proposed ODRepair as another means to repair order-dependent flaky tests without relying on test-suite code. ODRepair identifies the polluted shared state between tests and then constructs method sequences that reset that state. We use ODRepair to check the shared state between tests.

Bell and Kaiser [8] proposed VmVm as a runtime environment to reset shared state from static fields as to prevent pollution between tests. Arcuri et al. [7] also noticed issues with tests sharing state when automatically generating tests. They also proposed a runtime to reset shared state between generated tests. We also focus our analysis on how tests depend on each other through static fields.

Regression test selection. Regression test selection (RTS) aims to reduce the cost of regression testing by selecting to run only the tests that are affected by recent changes [46]. Early work on RTS focused on reducing the number of test selected through finer-grained analysis of the relationship between tests and code elements, such as statements or methods [22, 35–37, 46]. Recent work found that coarser-grained analysis, such as tracking the dependencies between test classes and classes under test, can be more efficient due to the faster analysis despite selecting more tests [18]. Industry has taken this finding further, focusing on even coarser-grained analysis for RTS, at the target/module level [13, 15, 34, 41, 49].

For Java projects, RTS techniques such as Ekstazi [17, 18] and STARTS [28, 29] are available to perform RTS by tracking class-level dependencies dynamically and statically, respectively. We use Ekstazi and STARTS to analyze changes and help select the relevant tests for detecting newly-introduced order-dependent flaky tests. Extensive work in recent years have leveraged Ekstazi and STARTS for different goals, e.g., evolution-aware runtime monitoring [30], detecting faults in RTS tools [48], or evaluating large-scale ecosystem RTS [19]. There have also been efforts to improve these tools, such as making Ekstazi refactoring-aware [44] or improving STARTS to approximate dependencies from reflection [39].

Recent work has also started exploring RTS that leverages machine learning algorithms as to predict which tests could fail based on a change before running them [14, 33]. These ML-based RTS techniques do not explicitly analyze the code changes to perform its prediction. For the purposes of our work, we cannot use such ML-based RTS techniques, because our goal is not to select which tests would newly fail after a change but rather identify new flakiness.

8 CONCLUSIONS

We propose IncIDFlakies, a technique for efficiently detecting order-dependent flaky tests introduced in a change. Existing technique iDFlakies must run all tests in different test orders without any knowledge of changes between code, resulting in high overheads if it has to run after every change, which happens frequently. IncIDFlakies improves upon iDFlakies by selecting to run in different test orders the subset of tests that can ensure detecting newly-introduced order-dependent flaky tests after a change. IncIDFlakies leverages RTS techniques to analyze the changes to find the tests whose behavior would be affected by the changes. However, running just these affected tests are not enough to ensure detecting order-dependent flaky tests given the nature of dependencies between tests. We augment the analysis to also include tests that are related to the affected tests through static fields, a common way that state is shared between tests in Java. Our evaluation on 67 order-dependent flaky tests where prior work has identified the corresponding flakiness-introducing commit shows that IncIDFlakies can select to run 65.4% of tests while running only 68.4% of the time baseline iDFlakies would need to run the same number of test orders with the full test suite. Furthermore, IncIDFlakies ensures all order-dependent flaky tests in our evaluation can be detected when running using its selected tests.

In the future, we plan on additional strategies to prioritize relative orderings between selected tests as to increase the chances of detecting newly-introduced order-dependent flaky tests within a limited number of test orders. We also plan to apply similar evolution-aware analysis to augment other techniques beyond iDFlakies for detecting order-dependent flaky tests [16, 45].

ACKNOWLEDGEMENTS

We thank Jaeseong Lee for his help in formulating some of the initial ideas of this work. We would like to acknowledge NSF grant no. CCF-2145774.

REFERENCES

[1] [n. d.]. iDFlakies. https://github.com/idflakies/iDFlakies.
[2] [n. d.]. IDoFT. http://mir.cs.illinois.edu/flakytests.
[3] [n. d.]. JNDI overview. https://docs.oracle.com/javase/jndi/tutorial/getStarted/overview/index.html.
[4] [n. d.]. NonDex. https://github.com/TestingResearchIllinois/NonDex.
[5] [n. d.]. Soot - A framework for analyzing and transforming Java and Android applications. http://soot-oss.github.io/soot/.
[6] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *International Conference on Software Engineering*.
[7] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *International Conference on Automated Software Engineering*.
[8] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *International Conference on Software Engineering*.
[9] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *International Symposium on Foundations of Software Engineering*.
[10] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *International Conference on Software Engineering*.
[11] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *International Symposium on Software Testing and Analysis*.
[12] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[13] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering*.
[14] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. In *International Symposium on Software Testing and Analysis*.
[15] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *International Conference on Software Engineering Companion*.
[16] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *International Conference on Software Testing, Verification, and Validation*.
[17] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight test selection. In *International Conference on Software Engineering (Tool Demonstrations Track)*.
[18] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*.
[19] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *International Symposium on Software Reliability Engineering*.
[20] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *International Symposium on Software Testing and Analysis*.
[21] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *International Working Conference on Source Code Analysis and Manipulation*.
[22] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software.
[23] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *International Symposium on Foundations of Software Engineering*.
[24] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification, and Validation*.
[25] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-test-aware regression testing techniques. In *International Symposium on Software Testing and Analysis*.
[26] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *International Symposium on Software Reliability Engineering*.
[27] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020).
[28] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *International Symposium on Foundations of Software Engineering*.
[29] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *International Conference on Automated Software Engineering (Tool Demonstrations Track)*.
[30] Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Rosu, and Darko Marinov. 2019. Techniques for evolution-aware runtime verification. In *International Conference on Software Testing, Verification, and Validation*.
[31] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing order-dependent flaky tests via test generation. In *International Conference on Software Engineering*.
[32] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*.
[33] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *International Conference on Software Engineering, Software Engineering in Practice*.
[34] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *International Conference on Software Engineering, Software Engineering in Practice*.
[35] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*.
[36] Gregg Rothermel and Mary Jean Harrold. 1994. A framework for evaluating regression test selection techniques. In *International Conference on Software Engineering*.
[37] Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology* 6, 2 (1997).
[38] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *International Conference on Software Testing, Verification, and Validation*.
[39] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-aware static regression test selection. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019).
[40] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[41] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjørner, and Jacek Czerwonka. 2017. Optimizing test placement for module-level regression testing. In *International Conference on Software Engineering*.
[42] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and improving regression test selection in continuous integration. In *International Symposium on Software Reliability Engineering*.
[43] Denini Silva, Leopoldo Teixeira, and Marcelo d'Amorim. 2020. Shake It! Detecting flaky tests caused by concurrency with Shaker. In *International Conference on Software Maintenance and Evolution*.
[44] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *International Conference on Software Engineering*.
[45] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2021. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *Tools and Algorithms for the Construction and Analysis of Systems*.
[46] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability* 22, 2 (2012).
[47] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis*.
[48] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *International Conference on Software Engineering*.
[49] Celal Ziftci and Jim Reardon. 2017. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale. In *International Conference on Software Engineering, Software Engineering in Practice*.