Systematically Producing Test Orders to Detect Order-Dependent Flaky Tests

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

M. Mahdi Khosravi
Middle East Technical University
Ankara, Turkey
mahdi.khosravi@metu.edu.tr

Wing Lam
George Mason University
Fairfax, Virginia, USA
winglam@gmu.edu

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

ABSTRACT

Software testing suffers from the presence of flaky tests, which can pass or fail when run on the same version of code. Order-dependent tests (OD tests) are flaky tests whose outcome depends on the order in which they are run. An OD test can be detected if specific tests are run or not run before it, resulting in a difference in test outcome. While prior work has proposed rerunning tests in different random test orders, this approach does not provide guarantees toward detecting all OD tests. Later work that proposed a more systematic approach to ordering tests still fails to account for the relationships between all tests in the test suite.

We propose three new techniques to detect OD tests through a more systematic means of producing test orders. Our techniques build upon prior work in Tuscan squares to cover test pairs in a minimal set of test orders while also obeying the constraints of how tests can be positioned in a test order w.r.t. their test classes. Further, as there are many test pairs that need to be covered, we develop a technique that can take a specified set of test pairs to cover and produce test orders that aim to cover just those test pairs. Our evaluation with 289 known OD tests across 47 test suites from open-source projects shows that our most cost-effective technique can detect 97.2% of the known OD tests with 104.7 test orders, on average, per subject. While all techniques produce a relatively large number of test orders, our analysis of the minimal set of test orders needed to detect OD tests shows a tremendous reduction in the test orders needed to detect OD tests – representing an opportunity for future work to prioritize test orders.

CCS CONCEPTS

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

KEYWORDS

flaky test detection, order-dependent flaky test

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598083

ACM Reference Format:
Chengpeng Li, M. Mahdi Khosravi, Wing Lam, and August Shi. 2023. Systematically Producing Test Orders to Detect Order-Dependent Flaky Tests. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598083

1 INTRODUCTION

Software testing is an important part of the development process, but it suffers from the presence of flaky tests. A *flaky test* is a test that nondeterministically passes or fails even when run on the same code [19]. When a flaky test fails, developers can be misled into thinking they just introduced a fault, even though the test may already fail before their recent changes. Flaky tests are prevalent in open-source software and in industry, with researchers at Facebook suggesting that everyone should "assume all tests are flaky" [10].

While there are many reasons why tests are flaky, one of the more frequent types of flaky tests is *order-dependent tests* (OD tests). An OD test passes or fails depending on the test order in which they are run [13, 29]. Conceptually, the reason OD tests can fail when run in a different test order is because prior tests in the test order modify some shared state, leading to the OD test to run in an unexpected starting state and having a different test outcome. Luo et al., in their prior empirical study on flaky tests in open-source projects, found that OD tests are among the top three most frequently observed types of flaky tests [19]. There has also been a wide range of techniques for detecting [4, 6, 8, 9, 13, 17, 27–29], repairing [18, 24], or accommodating the effects [3, 11] of OD tests.

One popular means to detect OD tests is to run different test orders. For example, Lam et al. previously developed iDFlakies to run tests in random test orders [13]. While their technique was effective in detecting many OD tests, randomly shuffling test orders does not provide strong guarantees on whether all OD tests have been detected. Wei et al. later developed a more systematic way to run test orders by covering pairs of tests [28]. A test pair `(t1, t2)` is covered if there is a test order in which test `t1` is positioned right before `t2`, with no other tests in between. An OD test can be detected if all test pairs are covered since every other test will run right before the OD test. Wei et al.'s approach is based on Tuscan squares [7], which guarantees that all pairs of tests can be covered in N or $(N+1)$ test orders for any N tests. Their approach focuses on covering test-class pairs, i.e., a pair of test classes `(A, B)` is covered

if there is a test order where A comes before B, with no other test classes in between. While this technique provides guarantees on the relative ordering between test-class pairs, it does not guarantee that all test pairs in any given test class will be covered or that all test pairs from different test classes will be covered.

Building upon Wei et al.'s approach, which we call *Tuscan Class-Only*, we propose new techniques to detect OD tests by producing test orders that cover test pairs and provide stronger guarantees. Our first technique, *Tuscan Intra-Class*, leverages Tuscan squares to not just cover all test-class pairs but also cover all test pairs within each test class, which we call intra-class test pairs. Covering these test pairs ensures that OD tests whose outcome depends on other tests in the same test class can be detected. We also develop *Tuscan Inter-Class*, which builds upon Tuscan Intra-Class by producing test orders that also cover all test pairs involving tests across different test classes, which we call cross-class test pairs. Tuscan Inter-Class ensures that every test will be run right after every other test, regardless of what classes the tests are in. Given the restrictions on the positioning between tests across different test classes in a single test order (tests from different test classes cannot be interleaved with one another [13]), Tuscan Inter-Class can produce many test orders to cover all cross-class test pairs (e.g., more than 24 million orders for one test suite in our evaluation).

We further propose *Target Pairs* as an additional technique to reduce the number of test orders while still covering relevant cross-class test pairs. The insight is that not all test pairs need to be covered, but rather only those that potentially have some shared state. Prior work found that the most common type of shared state among tests in Java projects is through in-memory heap reachable from static fields [9, 19, 29]. We develop Target Pairs to produce test orders that cover a specific set of test pairs, namely tests that access at least one static field in common. Our technique is a greedy approach that iteratively produces test orders that cover yet-to-be-covered test pairs until all required test pairs are covered.

We evaluate our techniques using a dataset of 289 OD tests published in prior work [28]. This dataset contains not only the names of OD tests but also the other tests that a given OD test needs to pass or fail. These 289 tests belong to 47 Maven modules, where each module contains its own test suite, and each module belongs to an open-source Java project. For each module, we run Tuscan Class-Only and our three new techniques to produce test orders that may detect the known OD tests from this dataset. Unsurprisingly, we find that Tuscan Class-Only produces the least number of test orders (94.0) and detects the smallest percentage of OD tests (36.0%), on average, while Tuscan Inter-Class produces the most orders (769148.3) but detects all OD tests. Surprisingly, we find that Target Pairs requires more test orders but detects fewer OD tests than Tuscan Intra-Class, suggesting that a nontrivial number of OD tests do not share static fields and that a nontrivial number of non-OD tests also share static fields. Overall, we find that Tuscan Intra-Class is the best technique in that it produces a small number of test orders (104.7), like Tuscan Class-Only, while being able to detect almost as many OD tests as Tuscan Inter-Class (97.2%). We also investigate the number of minimum test orders each technique needs to detect all OD tests, motivating future work on prioritization or selection of test orders for more efficient detection.

This paper makes the following main contributions:

New techniques to detect OD tests. We present three new techniques to systematically detect OD tests.
Evaluation. We evaluate the three new techniques and a technique from prior work, and we find that Tuscan Intra-Class is the most cost-effective technique to systematically detect OD tests. We also investigate the minimal test orders needed to detect all possible OD tests, motivating future work on more efficient detection.
Tools & Dataset. We make the implementation of our new techniques and all test orders generated for our evaluation publicly available for others to use for future work and study replication [2].

2 BACKGROUND

In this section, we provide background on order-dependent tests (OD tests). We define terms and categorizations as well as background on prior work for detecting OD tests.

2.1 Order-Dependent (OD) Tests

OD tests are flaky tests that deterministically pass or fail based on the order in which the tests are run [13, 15, 19, 29]. These tests are deterministic in that they either always pass or always fail for any given test order, and there is at least one test order in which the test passes (termed a *passing test order*) and at least one test order in which the test fails (termed a *failing test order*). We also use the term *original test order* to describe a specific, initial passing test order; typically, this test order is arbitrarily decided by the testing framework (e.g., JUnit). Existing techniques often detect OD tests by running different test orders, finding at least one passing test order and one failing test order for each OD test.

Shi et al. previously provided definitions for tests associated with OD tests [24]. They categorized OD tests into two different categories. The first category of OD tests is *brittles*. A brittle fails when run by itself, yet it passes when another test, termed a *state setter*, runs before it. In other words, the brittle requires some other test to set up the state shared between the tests, allowing the brittle to start running in an initial state that results in it passing. There can be many different state setters for a brittle. A passing test order for a brittle has at least one state setter run before the brittle, and a failing test order has no state setter run before the brittle [24, 28].

The second (and most prominent) category of OD tests is *victims*. Unlike brittles, these tests pass when run on their own but fail when some other test, called a *polluter*, runs beforehand. Conceptually, the polluter modifies or "pollutes" some state shared between the polluter and the victim, and the polluter does not reset this shared state after running. As such, the victim fails when run after its polluter. Shi et al. also defined another type of test, called *cleaners*, that also affects the relationship between polluters and victims. When a cleaner runs after the polluter but before the victim, the victim no longer fails. Conceptually, the cleaner "cleans" the shared state between polluter and victim. There can be many different polluters for a victim, and each polluter/victim pair can have different cleaners. A passing test order for a victim has either no polluters running before the victim or polluters running before the victim, but there is at least one cleaner that runs after all the polluters and before the victim. A failing test order for a victim has at least one polluter running before the victim and no cleaners running between the polluter closest to the victim and the victim itself.

Figure 1 shows an OD test example from `ktuukkan/marine-api`, an open-source project on GitHub that is used in our evaluation. The OD test is a victim, `testConstructor` (Line 21). When `testConstructor` runs on its own, the test passes. However, when it runs after `testRegisterParserWithAlternativeBeginChar` (Line 9), `testConstructor` fails. In other words, `testRegisterParserWithAlternativeBeginChar` is a polluter for `testConstructor`. Furthermore, the polluter is in a separate test class as the victim.

In this example, after the polluter runs, it unregisters `VDMParser.class` (Line 11). Unregistering means removing this entry from the shared static field map `parsers` (Line 27). The victim expects this entry to be in the `parsers` map as it tries to create an instance of this parser at the beginning of its execution. With the entry missing, an exception is thrown, and the victim fails.

There is also a cleaner for this victim, namely `testCreateParser` (Line 13). When this cleaner runs after the polluter, and before the victim, the victim passes. The reason is that when the cleaner runs, a setup method within its test class (Line 5) is run beforehand, which invokes the `reset` method for the shared `SentenceFactory` instance (Line 3). The `reset` method (Line 37) repopulates the `parsers` map with all the entries (Line 39), including the `VDMParser.class` that the victim needs. Note that the cleaner is in the same test class as the polluter. As the actual "cleaning" of the shared state is in the `setup` method, any test in the test class of this `setup` method is also a cleaner. However, if the polluter runs as the last test in the test class followed by the victim's test class, then the victim fails because no other test resets the shared state.

```
1 // test class with polluter and cleaner
2 public class SentenceFactoryTest {
3   private final SentenceFactory instance =
4     SentenceFactory.getInstance();
5   @Before public void setUp() throws Exception {
6     instance.reset();
7   }
8   @Test public void
9       testRegisterParserWithAlternativeBeginChar() {
10    ...
11    instance.unregisterParser(VDMParser.class);
12  }
13  @Test public void testCreateParser() { ... }
14 }
15 // test class with victim
16 public class AbstractAISMessageListenerTest {
17   private final SentenceFactory sf =
18     SentenceFactory.getInstance();
19   private final AISSentence AIS_01 = (AISSentence)
20     sf.createParser("VDM");
21   @Test public void testConstructor() { ... }
22 }
23 // class of shared static field
24 public final class SentenceFactory {
25   // map containing parser classes
26   private static
27     Map<String, Class<? extends SentenceParser>> parsers;
28   public void unregisterParser
29       (Class<? extends SentenceParser> parser) {
30     for (String key : parsers.keySet()) {
31       if (parsers.get(key) == parser) {
32         parsers.remove(key);
33         break;
34       }
35     }
36   }
37   public void reset() {
38     ...
39     registerParser(tempParsers, "VDM", VDMParser.class);
40     ...
41     parsers = tempParsers;
42   }
43 }
```

Figure 1: Example OD test from `ktuukkan/marine-api`.

2.2 Detecting OD Tests

Prior work proposed rerunning tests in different, random test orders [13, 29], detecting an OD test when a passing test order and a failing test order is found. However, there are no guarantees on whether a particular OD test can be detected within some number of random test orders. For example, to detect a victim, there must be a test order in which the polluters are before the victim, and no cleaners are after all polluters and before the victim. If there are many cleaners and few polluters, then the chances of obtaining a test order in which the victim fails can be low, requiring many different random test orders to find one where the victim fails. In fact, prior work has reported that the chance of generating a failing order for an OD test can be as low as 1.2% [28].

To provide guarantees on running test orders in which the OD test can both pass or fail and motivated by the fact that OD tests rarely require multiple tests to fail [24], Wei et al. [28] proposed utilizing the theory behind Tuscan squares [7] to systematically produce test orders. Given a natural number N, a Tuscan square consists of N rows, each of which is a permutation of the numbers $\{1, 2, \ldots, N\}$, and every pair $\langle a, b\rangle$ of distinct numbers occurs consecutively in some row. Given a set of N tests, Wei et al. used Tuscan squares to produce N permutations of these tests, such that for all pairs of tests $\langle t_1, t_2\rangle$, there exists a permutation where t_1 is positioned right before t_2 (no other element is positioned in-between the two) and there exists another permutation where t_2 is positioned right before t_1. Tuscan squares can be used to construct exactly N permutations for all values of N, except $N = 3$ or $N = 5$. For $N = 3$ and $N = 5$, Tuscan squares will have to use four and six test orders, respectively, to cover all test pairs (e.g., for $N = 3$, Tuscan squares will need four orders to cover all six pairs: $\{\langle t_1, t_2, t_3\rangle, \langle t_2, t_1, t_3\rangle, \langle t_3, t_2, t_1\rangle, \langle t_3, t_1, t_2\rangle\}$. In theory, pairs $\{\langle t_1, t_2\rangle, \langle t_2, t_1\rangle\}$ need not be run in the last two orders. However, running a few more pairs typically adds negligible runtime cost, and yet doing so can help better detect if t_1 and t_2 are nondeterministic, flaky tests or OD tests that require multiple tests as polluters.

Wei et al. proposed treating test classes as the elements in a set, and so computing the Tuscan square for these test classes results in test orders of test classes in which each pair of test classes runs right before and right after each other test class in some permutation of test classes. The reason they treat test classes as elements, as opposed to individual tests, is that when tests are run using JUnit and Surefire (the main unit-testing infrastructure used for Maven-based Java projects), tests from different test classes cannot be interleaved. For example, test class `A` contains two tests, `t1` and `t2`, and another test class `B` contains test `t3`, the test order `[A.t1, B.t3, A.t2]` cannot be run in one execution of JUnit and Surefire, because

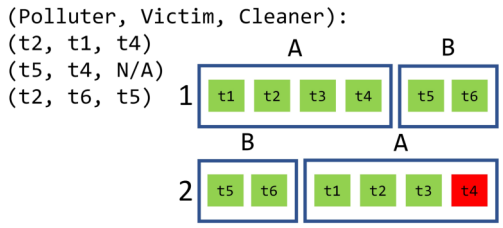


Figure 2: The test orders produced by Tuscan Class-Only.

tests from one test class B cannot be run in-between tests in another test class A. However, test classes can be freely ordered relative to each other. We call this strategy of producing test orders using Tuscan squares on test classes as *Tuscan Class-Only*. Essentially, Tuscan Class-Only produces test orders that cover all test-class pairs, where a test-class pair (A, B) is covered if there is a test order where test class A is positioned right before test class B, and no other test classes are positioned in-between.

Tuscan Class-Only guarantees detecting victims where its polluters are in a separate test class, and there are no cleaners in the polluter or victim test classes. Tuscan Class-Only will produce a failing test order when any of the test classes containing polluters are run right before the victim test class. For Tuscan squares to generate only N orders, Tuscan squares must position the victim's test class first in one order, thereby ensuring that there is at least one passing test order where the victim's test class will run before all test classes that contain polluters. The total number of test orders to try is the number of test classes (except for three or five test classes, where the number of test orders is four or six, respectively).

As Tuscan Class-Only does not change the order of tests in the same test class, it may not detect any OD tests that require a different ordering between tests in the same test class. For example, if a victim's only polluter is in the same test class, and the original test order has the victim run before the polluter, then Tuscan Class-Only will never produce a test order where the victim fails. Furthermore, if the polluter and victim are in different test classes, but there are cleaners that are in the same test class as the polluter or victim, then Tuscan Class-Only is also not guaranteed to produce a failing test order because the original test order can have the cleaner always be run before the victim or after the polluter, so no matter how the test classes are reordered, the victim will always have a cleaner run in-between the polluter and itself. As such, we present better techniques that can produce test orders that guarantee the detection of OD tests even when there are cleaners in the same test class as the victim or its polluters.

Example. Figure 2 illustrates an example set of tests and what test orders Tuscan Class-Only would produce. In this example, there are six tests, t1 through t6, where tests t1 through t4 are in test class A and tests t5 and t6 are in test class B. The original test order goes from t1 through t6. The figure illustrates the relationship between tests in terms of OD tests: test t1 is a victim whose polluter is t2, test t4 is a victim whose polluter is t5, and test t6 is a victim whose polluter is t2. Furthermore, test t4 is a cleaner for the polluter/victim pair of t2/t1, and test t5 is a cleaner for the polluter/victim pair of t2/t6.

Focusing only on test classes, Tuscan Class-Only produces two test orders: the first test order has all tests in test class A run before B and the second test order has all tests in test class B run before A. All tests within a test class are in the same test order relative to each other as in the original test order. We see that using Tuscan Class-Only results in detecting the victim t4 since t4 runs after t5 in the second test order. However, Tuscan Class-Only does not produce any test order that detects t2 since it does not change the relative order within test class A, and it does not produce any test order that detects t6 because cleaner t5 always runs before t6.

3 DETECTION TECHNIQUES

We next describe three new techniques for detecting OD tests via systematic generation of test orders.

3.1 Tuscan Intra-Class

We propose *Tuscan Intra-Class* to additionally systematically change the ordering of tests in the same test class. We leverage Tuscan squares again to produce the permutation of tests within each test class. Essentially, Tuscan Intra-Class ensures producing test orders that cover all test-class pairs (as Tuscan Class-Only does) and all test pairs within each test class.

First, Tuscan Intra-Class uses Tuscan squares to compute permutations of test classes to cover all test-class pairs, similar to Tuscan Class-Only. Then, for each test class, Tuscan Intra-Class uses Tuscan squares to compute all the permutations of tests within the test class to cover those test pairs; we call the test pairs involving only tests in the same test class as *intra-class test pairs*. Then, at each iteration, Tuscan Intra-Class produces a test order by first positioning the test classes based on the computed permutation of test classes, and then for each test class, positioning the tests within based on the computed permutation of tests.

At each iteration, if all test-class pairs are covered, then Tuscan Intra-Class loops back to position test classes according to the first permutation of test classes. Similarly, if all intra-class test pairs for a test class are covered, it loops back to that first permutation of tests for that test class. This process ensures that Tuscan Intra-Class will continue to produce test orders if there are still uncovered test-class pairs or intra-class test pairs. Tuscan Intra-Class continues producing test orders until all test-class pairs and all intra-class test pairs within each test class have been covered.

Figure 3 shows how Tuscan Intra-Class produces test orders for the same tests from Figure 2. The highest number of test methods within a test class is four (test class A), which is more than the number of test classes, so Tuscan Intra-Class produces four test orders. Tuscan Intra-Class iterates through the permutations of tests within each test class following the computed Tuscan squares to ensure all intra-class test pairs are covered. Further, the test-class pairs are covered as well, as at least one test order has test class A come before B, and another has the opposite. By the second test order, all intra-class test pairs for B have been covered, so the later test orders loop through permutations already covered for B.

Same as Tuscan Class-Only, the victim t4 can be detected. Further, Tuscan Intra-Class ensures that if the victim has a polluter in the same test class, it can produce a failing test order. Whether or not there is a cleaner for the polluter/victim pair does not matter

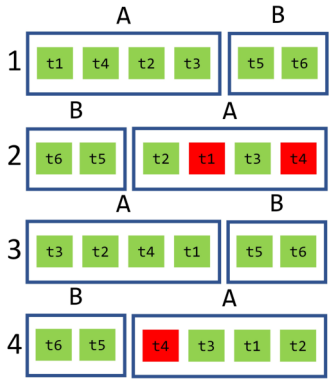


Figure 3: The test orders produced by Tuscan Intra-Class.

since Tuscan Intra-Class ensures that there is a test order where the polluter is positioned right before the victim, with no other tests in between. We see in this example that victim `t1` is detected (in the second test order) because `t1` runs right after its polluter `t2`, unlike in the third test order where the cleaner `t4` runs in between.

However, Tuscan Intra-Class is not guaranteed to detect OD tests that require a specific ordering of tests across test classes. Consider the case where a victim and polluter are in separate test classes, but there are cleaners that are in the same test class as either the victim or polluter. Although Tuscan Class-Only and Tuscan Intra-Class guarantee a test order where the polluter test class is right before the victim test class, a cleaner may still be run between the polluter and victim. We see in this example that Tuscan Intra-Class does not detect victim `t6` because, in all test orders where the polluter test class is run before, the cleaner `t5` also runs before `t6`.

3.2 Tuscan Inter-Class

We propose *Tuscan Inter-Class* to cover all test pairs, including those involving tests between test classes, which we call *cross-class test pairs*. Recall that tests from different test classes cannot be interleaved with each other (Section 2.2). To cover a cross-class test pair, their respective test classes must be positioned right next to each other, and the tests in the cross-class test pair must be positioned at the "boundaries" of their respective test classes. As such, two cross-class test pairs, such as `(A.t1, B.t3)` and `(A.t2, B.t4)`, cannot both be covered in the same test order because only one test can be at the boundaries of each test class, and test classes cannot repeat in the same test order.

Tuscan Inter-Class first computes the permutations of test classes that cover all test-class pairs. Starting with the first permutation of test classes, it iterates through each consecutive pair of test classes. For each test class of a consecutive test-class pair, Tuscan Inter-Class computes the permutation of tests that would cover the intra-class test pairs within that test class. It then maintains one permutation for the second test class in that pair while iterating the permutations in the first test class. After iterating through all the permutations of the first test class, it iterates to the next permutation for the second test class and loops through the permutations for the first test class again. In other words, Tuscan Inter-Class explores the complete combination of test permutations between the first and second test classes of a test-class pair. Tuscan squares within a test class ensures for each test at least one test order in which it is first and another in which it is last. As such, every test in the first test class of the test-class pair can be positioned right before every test in the second test class. Therefore, all cross-class test pairs involving tests in the first test class coming before tests in the second test class would be covered. The intra-class test pairs of each test class would also get covered as Tuscan Inter-Class iterates through all permutations of tests in each test class according to Tuscan squares. After covering all cross-class test pairs for the current pair of test classes, Tuscan Inter-Class changes to the next consecutive test-class pair, exploring the combinations of their permutations. Eventually, all cross-class test pairs involving all consecutive test-class pairs are covered, and Tuscan Inter-Class moves on to the next permutation of test classes.

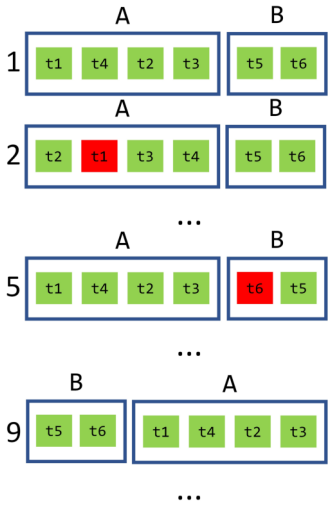


Figure 4: The test orders produced by Tuscan Inter-Class.

Figure 4 shows how Tuscan Inter-Class produces test orders for the same example of six tests. The first test order matches that of Tuscan Intra-Class, but for the second test order, only the order of tests in `A` changes, while the order of tests in `B` remains the same. This design decision ensures that the cross-class test pair `(t4, t5)` is covered. Tuscan Inter-Class iterates through all four permutations of tests for test class `A`, so by the fifth test order, it finally starts changing the ordering of tests in `B`, and we see that the permutations of tests in `A` loop back to the first permutation. The victim `t6` fails because the cleaner `t5` does not run between the polluter `t2` and `t6` (another test order will also position `t2` right before `t6`, guaranteeing that `t2` will fail). Eventually, for the ninth test order, the test classes `A` and `B` switch positions, and Tuscan Inter-Class loops through the permutation of tests within each test class again. Overall, Tuscan Inter-Class tries 16 test orders, covering all cross-class test pairs and intra-class test pairs.

3.3 Target Pairs

We propose *Target Pairs* as an alternative technique for producing test orders while still covering cross-class test pairs. To reduce the number of test orders, we leverage the insight that not all test pairs

need to be positioned next to each other in a test order if they do not share any state. If a test does not share a state with another, then they cannot have any polluter/victim or state setter/brittle relationship. Target Pairs takes as input a set of test pairs, with the goal to produce test orders that cover those test pairs. Target Pairs iteratively produces test orders by greedily producing test orders that cover the most uncovered test pairs that share a state.

Prior work found that tests in Java can share states in a number of ways, such as through files, databases, or external services, but the main type of shared state is through shared heap-state accessed through static fields [18, 19, 29]. Therefore, we focus on test pairs that access at least one static field in common between them. We first instrument the code to track when a test executes a `getstatic` or `putstatic` Java bytecode instruction, indicating it is accessing or writing to some static field. From the bytecode instruction, we obtain the static field name. We run each test on this instrumented code to map each test to the static fields it accesses or writes to. Note that even if two tests only execute `getstatic` on the same static field, it is still possible for one of the tests to be an OD test as certain method calls that write to static fields can do so using only `getstatic` (e.g., adding to a list uses `getstatic` as it is "rewriting" the list and not "reassigning" the list). With this mapping of tests to static fields, we pair together tests that have at least one static field in common. These test pairs are the input to Target Pairs.

Figure 5 shows the pseudocode for the Target Pairs algorithm. The entry function is `TargetPairs`, which takes as input the set of test pairs to cover. While there are uncovered test pairs, the algorithm produces a new test order (Line 4), with the aim to cover the most number of test pairs, adding the produced test order to the growing list of test orders. At each iteration, the covered test pairs are removed, and then the algorithm produces a new test order. The final output is the list of produced test orders.

The function `produce_best_order` produces a test order that greedily covers the most uncovered test pairs. The function first divides up the test pairs into two subsets, the cross-class test pairs, and the intra-class test pairs. We distinguish between the two because the cross-class test pairs are the most restrictive, as we cannot cover more than one cross-class test pair involving the same test classes in a single test order. As such, the goal is to prioritize covering cross-class test pairs first. Once the relative ordering between test classes is set, we then focus on covering intra-class test pairs within each test class.

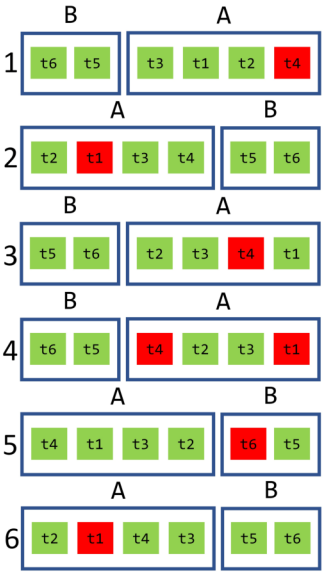
The function `produce_best_order` starts with an empty list of tests for the test order and first finds the test that occurs the most among uncovered cross-class test pairs (Line 19). The algorithm gets the corresponding test classes for such a test and marks it as scheduled within the test order. It then sets the chosen test as either the first or last test within the test class. The test is the first test if it occurs the most as the second test among cross-class test pairs (so it has the most number of "connections" going left), and it is the last test if it occurs the most as the first test among cross-class test pairs (so it has the most number of "connections" going right). Once the test is set at one of these boundaries for its test class, we extend test classes in the corresponding direction of that test class if there are cross-class test pairs that we need to cover. The algorithm then keeps track of the right-most and left-most test classes in the test order in two queues (Lines 28 to 29). While there are still test classes

```
1  def TargetPairs(pairs):
2    testorders = []
3    while len(pairs) > 0:
4      order = produce_best_order(pairs)
5      testorders.append(order)
6      update_pairs(order, pairs) # remove covered pairs
7    return testorders
8
9  def produce_best_order(pairs)
10   # split test-pairs between cross-class and intra-class
11   cc_pairs = get_crossclass_pairs(pairs)
12   ic_pairs = get_intraclass_pairs(pairs)
13   # keep track of already scheduled test classes,
14   #  as they cannot repeat
15   sched_classes = set()
16   best_order = []
17   # find test involved with most cross-class pairs
18   #  from yet-to-be-scheduled test classes
19   best_test = get_best_test(cc_pairs, sched_classes)
20   while best_test:
21     test_class = get_test_class(best_test)
22     test_classes_order = [test_class]
23     sched_classes.add(test_class)
24     # set test as first or last based on freq in pairs
25     set_test_in_class(best_test, test_class, cc_pairs,
26       sched_classes)
27     # queues to keep track of right-most and left-most
28     right_queue = [test_class]
29     left_queue = [test_class]
30     # keep extending right and left,
31     #  and set boundary tests to cover cross-class pairs
32     while right_queue.peek() and left_queue.peek():
33       if right_queue.peek():
34         test_class = right_queue.pop()
35         other_test_class = best_right_pair(test_class,
36           cc_pairs, sched_classes)
37         if other_test_class:
38           sched_classes.add(other_test_class)
39           test_classes_order.append(other_test_class)
40           right_queue.push(other_test_class)
41       if left_queue.peek():
42         test_class = left_queue.pop()
43         other_test_class = best_left_pair(test_class,
44           cc_pairs, sched_classes)
45         if other_test_class:
46           sched_classes.add(other_test_class)
47           test_classes_order.prepend(other_test_class)
48           left_queue.push(other_test_class)
49     # get test-order by populating tests within
50     # test classes
51     order = fill_tests(test_classes_order,
52       cc_pairs, ic_pairs)
53     best_order += order
54     # prepare next iteration for next test
55     best_test = get_best_test(cc_pairs, sched_classes)
56   best_order = fill_other_tests(best_test, icpairs)
57   return best_order
```

Figure 5: Target Pairs algorithm.

in the queues, the algorithm continues to extend test classes to the right and left of the current test classes in the test order, covering as many cross-class test pairs as possible (Lines 32 to 48).

At each iteration of the loop that adds new test classes, the algorithm first tries to extend right by choosing a test class that can connect with the current right-most test class based on cross-class test pairs and yet-to-be-scheduled test classes. If the right-most test




```
Test-Pairs:
 Cross-Class: (t3, t5), (t5, t3),
              (t4, t5), (t5, t4),
              (t2, t6), (t6, t2)
 Intra-Class: (t1, t2), (t2, t1)
```

Figure 6: The test orders produced by Target Pairs.

class already has a test set as the last test within, then only test classes that can connect with this test among the cross-class test pairs can be considered. Otherwise, any test class with a test in a cross-class test pair that connects with some test in the current right-most test class can be chosen (Line 35). The boundary tests are set to cover the cross-class test pair, and the extending test class is scheduled and pushed into the queue as the right-most test class. A similar logic is applied for extending towards the left. The loop ends once no more cross-class test pairs can be covered by extending right and left from the current test classes. The remaining tests within each scheduled test class so far are ordered to cover intra-class test pairs while respecting the tests already positioned at the boundaries of each test class (Line 51). All the tests in this order are appended to the growing test order (Line 53). Then, the algorithm tries to find the next best test as to concatenate another sequence of tests to this growing test order. Once no more tests can be included to cover cross-class test pairs, any remaining test classes are added to the end, and the tests within each of those test classes are permuted to cover intra-class test pairs (Line 56). We choose to add the remaining test classes to the end as running extra tests can help detect non-deterministic, flaky tests or OD tests that require multiple tests as polluters.

Figure 6 shows how Target Pairs would produce test orders for the same example of six tests. The set of test pairs to cover is shown in the figure. For the first test order, Target Pairs chooses a test that is involved in the most cross-class test pairs, which in this case is `t5`. Choosing a cross-class test pair to extend to the right, Target Pairs chooses `(t5, t3)` and extends test class `A` to the right. It fills in the tests in each test class, covering intra-class test pair `(t1, t2)` as well. For the next iteration, Target Pairs produces a new test order that first tries to cover cross-class test pair `(t4, t5)` and then fills in tests to cover intra-class test pair `(t2, t1)`. Eventually, all intra-class test pairs are covered, and only cross-class test pairs remain. The subsequent test orders all aim to cover one of these cross-class test pairs, randomly filling in the remaining tests in each test class as it does not need to consider intra-class test pairs any longer. Overall, Target Pairs produces six test orders that detect all victims, including victim `t6`, which is run right after its polluter `t2` from another test class in the fifth test order.

3.4 Guarantees for Each Technique

Each technique has different guarantees concerning the OD tests they can detect with the test orders they produce. Tuscan Class-Only is guaranteed to detect all victims that have all polluters in another test class and there are no cleaners in either the polluter's or victim's test class. Tuscan Class-Only is also guaranteed to detect brittles if there are state setters in a test class different than the brittle and there are no state setters in the brittle's test class.

Tuscan Intra-Class guarantees to detect all OD tests that Tuscan Class-Only can detect. Further, it can also detect all victims whose polluters are in the same test class as the victim. Concerning brittles, Tuscan Intra-Class also guarantees to detect brittles that have all state setters that are in the same test class as the brittle. The reason it cannot guarantee detecting brittles with state setters in both the same test class and different test class as the brittle is because a brittle only fails if it is run before all state setters. Tuscan Intra-Class will guarantee (1) that the brittle will run before every test in its own test class and (2) that the brittle's test class will run before every other test class, but guarantees (1) and (2) may not be in the same test order. If guarantees (1) and (2) are satisfied in separate orders, then a brittle may still go undetected.

Tuscan Inter-Class is guaranteed to detect *all* OD tests. It can detect victims with polluters in other test classes and with cleaners in the same test class as the polluter or victim (which is not handled by Tuscan Class-Only), because Tuscan Inter-Class is guaranteed to produce a test order where each test across test classes is right next to each other. Tuscan Inter-Class is also guaranteed to detect all brittles as well (which is not handled by Tuscan Intra-Class), because it produces test orders where the brittle is the absolute first test in the test order (the brittle's test class is the first test class, and the brittle is the first test in that test class).

Target Pairs is guaranteed to detect all victims whose single polluter modifies shared state via static fields, resulting in different test outcomes. Target Pairs is also guaranteed to detect brittles with just one state setter that sets the shared state via static fields. Note that Target Pairs cannot guarantee the detection of brittles with more than one state setter since it cannot guarantee the brittle runs before all other tests (same as Tuscan Intra-Class).

4 EVALUATION SETUP

For our evaluation, we use a dataset of known OD tests from prior work [28]. Prior work also categorized the OD tests into brittles and victims and reported the corresponding state setters/polluters /cleaners for these tests. We obtain 47 subjects for our evaluation (a subject is a GitHub Maven project and its module that contains OD

Table 1: Subjects used in evaluation.

ID	Project-Module Name	# Test Class	# Test Method	# OD Tests
1	activiti-spring-boot-starter	11	45	16
2	fastjson	2308	4953	2
3	dubbo-cluster	23	112	3
4	dubbo-common	70	520	1
5	dubbo-config-api	23	263	37
6	dubbo-filter-cache	5	9	1
7	dubbo-rpc-api	23	60	2
8	dubbo-rpc-dubbo	16	74	2
9	dubbo-serialization-fst	4	21	1
10	hadoop-auth	22	125	1
11	hadoop-hdfs-nfs	12	64	28
12	hadoop-mapreduce-client-app	49	394	4
13	hadoop-mapreduce-client-core	60	293	5
14	hadoop-mapreduce-client-hs	26	206	2
15	portlet	12	61	4
16	c2mon-server-elasticsearch	11	19	1
17	cukes-http	4	14	1
18	integration-test-2_1	40	233	9
19	integration-test-2_2	4	23	7
20	integration-test-3_10	3	24	2
21	integration-test-3_7	1	6	6
22	dropwizard-logging	18	80	1
23	elastic-job-lite-core	98	511	3
24	hsac-fitnesse-fixtures	49	251	1
25	spring-data-ebean	2	48	1
26	jhipster-registry	14	53	1
27	lib	2	163	25
28	marine-api	71	926	12
29	openpojo	213	1185	3
30	spring-boot-actuator-auto...	166	733	8
31	spring-boot	230	2255	2
32	spring-boot-test-autoconfigure	103	234	4
33	spring-boot-test	115	721	1
34	spring-data-envers	4	10	2
35	spring-ws-core	145	971	15
36	spring-ws-security	43	122	2
37	aismessages	18	44	2
38	unix4j-command	30	290	1
39	compute	74	488	1
40	request	49	325	1
41	wdtk-dumpfiles	9	50	3
42	wdtk-util	5	23	2
43	naming	11	82	44
44	subsystem	10	25	1
45	wro4j-core	137	851	15
46	carbon-apimgt	46	530	1
47	riptide-spring-boot-starter	25	40	2
Average × 2 | Sum × 1		93.9	394.3	289

tests). Table 1 shows the breakdown and statistics of the subjects. For each subject, we show an ID for the subject to use in subsequent tables, the GitHub project and module name for the subject[1], the number of test classes and the number of tests, and the number of OD tests previously detected for the subject. Overall, we see that each subject has, on average, 93.9 test classes and 394.3 tests. There are a total of 289 OD tests in this dataset.

For each subject, we pass the original test orders as input to each technique. Original test orders are obtained from running the test suite once and recording the order in which tests are run. For Target Pairs, we also run the tests once to collect the static fields each test depends upon to construct the test pairs that need to be covered. We also run Tuscan Class-Only as a baseline technique for comparison purposes. We measure the time it takes to compute the test orders for each technique. However, we find that for some subjects, Tuscan Inter-Class and Target Pairs would take an immense amount of time to produce test orders, given the large number of test pairs to be covered (see Section 5 for more details). For Tuscan Inter-Class cases, we compute the theoretical number of test orders for subject ID 2 due to its large number of possible orders, and we run the actual technique for all other subjects. However, we cannot compute the theoretical for Target Pairs, so we set a timeout of 24 hours per subject when producing test orders.

Since we know which tests are OD tests as well as their corresponding state setters/polluters/cleaners, we can simulate how effective a technique is at detecting the OD test by checking each of the produced test orders. An OD test is detected if there is at least one passing test order and one failing test order. We can compute whether an OD test passes or fails in a test order based on the relative positions of state setters/brittles and polluters/victims/cleaners (Section 2.1). Further, we simulate how long it takes to run a technique on a subject by measuring the time it takes to run all the tests in a subject (averaged over five runs). The total time is the sum of the time to produce the test orders and the product of the time to run all tests and the number of test orders.

5 EVALUATION

Our evaluation addresses the following research questions:
RQ1: How many test orders does each technique produce?
RQ2: How effective is each technique at detecting OD tests?
RQ3: How cost-effective is each technique?
RQ4: What is the minimal number of test orders needed by each technique to detect known OD tests?

5.1 RQ1: Number of Test Orders

Table 2 shows the results from running each technique on all 47 subjects. We show under the columns "# orders total" the number of test orders each technique produces for each subject ("t/o" means timeout). Overall, we see that Tuscan Class-Only produces the least number of test orders across all subjects, with an average of 94.0 test orders, while Tuscan Inter-Class produces the most, with an average of 769148.3 test orders. This trend matches our expectations.

The number of test orders that Tuscan Intra-Class produces is, on average, a few more than those produced by Tuscan Class-Only (104.7 vs. 94.0). For individual subjects, the two techniques

[1]Maven projects can have multiple modules

Table 2: Detection results from running the techniques.

	Tuscan Class-Only			Tuscan Intra-Class			Tuscan Inter-Class			Target Pairs		
	# orders		% OD	# orders		% OD	# orders		% OD	# orders		% OD
ID	total	min.	det.	total	min.	det.	total	min.	det.	total	min.	det.
1	11	2	18.8	11	4	100.0	1794	5	100.0	217	3	100.0
2	2308	2	100.0	2308	2	100.0	24501496	2	100.0	t/o	t/o	t/o
3	23	2	33.3	27	2	100.0	11244	2	100.0	1616	2	100.0
4	70	0	0.0	70	2	100.0	261418	2	100.0	1005	2	100.0
5	23	0	0.0	38	37	100.0	64268	37	100.0	469	18	48.6
6	6	0	0.0	6	2	100.0	70	2	100.0	4	0	0.0
7	23	0	0.0	23	2	100.0	3386	2	100.0	90	2	100.0
8	16	0	0.0	16	2	100.0	5036	2	100.0	512	2	100.0
9	4	0	0.0	14	2	100.0	224	2	100.0	70	2	100.0
10	22	2	100.0	26	2	100.0	14270	2	100.0	726	2	100.0
11	12	2	100.0	22	2	100.0	3330	2	100.0	757	2	100.0
12	49	2	25.0	49	2	100.0	148542	2	100.0	13500	2	100.0
13	60	2	80.0	60	2	100.0	82814	2	100.0	5228	2	100.0
14	26	2	100.0	26	2	100.0	39500	2	100.0	4500	2	100.0
15	12	0	0.0	12	2	100.0	3306	2	100.0	353	2	100.0
16	11	0	0.0	11	2	100.0	312	2	100.0	50	2	100.0
17	4	0	0.0	6	2	100.0	122	2	100.0	10	2	100.0
18	40	2	11.1	40	8	100.0	50658	8	100.0	7488	8	100.0
19	4	0	0.0	8	7	100.0	382	7	100.0	147	7	100.0
20	4	0	0.0	16	2	100.0	316	2	100.0	150	2	100.0
21	1	0	0.0	6	6	100.0	6	6	100.0	10	4	66.7
22	18	0	0.0	22	0	0.0	5640	2	100.0	406	2	100.0
23	98	2	33.3	98	2	100.0	256130	2	100.0	1875	2	100.0
24	49	0	0.0	49	2	100.0	60816	2	100.0	1979	2	100.0
25	2	0	0.0	41	0	0.0	574	2	100.0	574	2	100.0
26	14	2	100.0	14	2	100.0	2572	2	100.0	12	0	0.0
27	2	0	0.0	161	2	100.0	644	2	100.0	278	2	100.0
28	71	0	0.0	71	2	100.0	841810	2	100.0	733	2	100.0
29	213	2	100.0	213	2	100.0	1392896	2	100.0	29154	2	100.0
30	166	0	0.0	166	2	100.0	531042	2	100.0	13419	2	100.0
31	230	0	0.0	230	2	100.0	5015642	2	100.0	t/o	t/o	t/o
32	103	0	0.0	103	2	100.0	53474	2	100.0	826	2	50.0
33	115	0	0.0	200	2	100.0	472332	2	100.0	1026	2	100.0
34	4	0	0.0	6	0	0.0	58	2	100.0	0	0	0.0
35	145	2	6.7	145	8	100.0	932108	8	100.0	6709	8	100.0
36	43	0	0.0	43	0	0.0	14452	2	100.0	80	0	0.0
37	18	0	0.0	18	2	100.0	1788	2	100.0	216	2	100.0
38	30	0	0.0	41	2	100.0	79464	2	100.0	10004	2	100.0
39	74	0	0.0	74	2	100.0	232368	2	100.0	13102	2	100.0
40	49	0	0.0	49	2	100.0	101578	2	100.0	10298	2	100.0
41	9	0	0.0	11	2	66.7	2116	2	100.0	411	2	100.0
42	6	0	0.0	9	2	100.0	492	2	100.0	6	2	100.0
43	11	2	90.9	20	2	97.7	5712	2	100.0	738	3	100.0
44	10	2	100.0	10	2	100.0	554	2	100.0	39	2	100.0
45	137	2	93.3	137	3	100.0	713822	3	100.0	24269	2	100.0
46	46	0	0.0	169	2	100.0	237882	2	100.0	34433	2	100.0
47	25	0	0.0	25	2	100.0	1512	2	100.0	38	2	100.0
Avg.	94.0	2.0	36.0	104.7	3.4	97.2	769148.3	3.3	100.0	4167.3	2.9	89.8

often produce the same number of test orders, which suggests that the number of test orders is dominated by the number of test classes as opposed to tests within a test class. Target Pairs produces substantially more test orders (4167.3), though still much fewer than Tuscan Inter-Class. We observe that Target Pairs still needs to cover a large number of test pairs, with an average of 499399.4 per subject. Further, they are dominated by the number of cross-class test pairs, with an average of 494869.7 per subject.

While Tuscan Inter-Class produces a large number of test orders, this number is still much less than the number of all possible valid test orders for a subject. The number of possible test orders for a subject would be the number of permutations of test classes multiplied by the product of the number of permutations of tests per test class, which well exceeds 10 million for most subjects.

5.2 RQ2: Detected OD Tests

Table 2 shows how effective each technique is at detecting OD tests, measured by the percentage of the known OD tests detected (shown under column "% of OD det."). We see that Tuscan Class-Only produces the fewest number of test orders, but it detects the least number of OD tests, detecting 36.0% of the known 289 OD tests. Tuscan Intra-Class detects more OD tests, detecting 97.2%. When we look into the OD tests that Tuscan Intra-Class does not detect, we find that they are victims that have polluters in separate test classes but have cleaners in the same test class as the polluter or victim (Section 3.1), or they are brittles with multiple state setters in both the same test class as brittles and different test class than brittles (Section 3.4). Meanwhile, Target Pairs detects 89.8% of known OD tests, which is lower than what Tuscan Intra-Class detects. Our inspection shows that Target Pairs suffers similar problems with brittles as Tuscan Intra-Class. In addition, for some victims, the shared state between polluter and victim is not from static fields, so Target Pairs does not attempt to cover the relevant test pairs. Future work may track more dependencies to provide additional test pairs for Target Pairs to cover. However, we see that Target Pairs can detect four victims that Tuscan Intra-Class does not detect, showing some value in specifying the relevant test pairs to cover.

5.3 RQ3: Cost-Effectiveness

Figure 7 shows boxplots that illustrate the distribution of the number of seconds that each technique takes to produce test orders and then run those test orders across all subjects. Note that, for presentation purposes, the y-axis is broken into parts because some techniques need a large amount of time relative to others, particularly for Tuscan Inter-Class and Target Pairs. The black line in the boxes represents the median, while the red triangle represents the mean. We see that the mean is always much larger than the median because there are a few subjects that need much more time than others, generally because the techniques all need to produce many more test orders for those subjects. Unexpectedly, we find that Tuscan Inter-Class takes much more time compared against other techniques due to the large number of test orders it produces.

We also evaluate the cost-effectiveness of each technique by measuring the average time it takes to detect an OD test, computed as the time to produce test orders, then run all test orders divided by the number of detectable OD tests per each technique. We find that

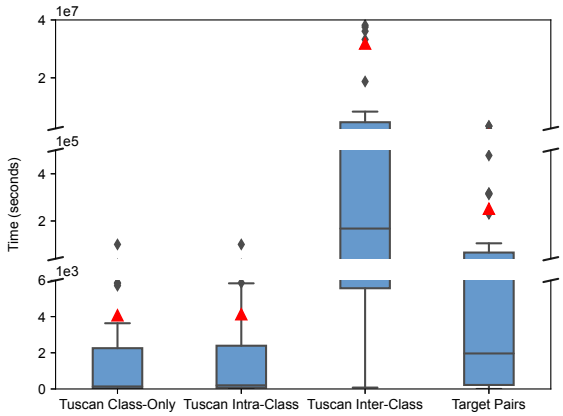


Figure 7: Time to run for all four techniques.

Tuscan Intra-Class is the most cost-effective technique, detecting an OD test every 2398 seconds, on average. While Tuscan Class-Only does not detect as many OD tests as Tuscan Intra-Class, it does produce fewer test orders. Yet, on average, Tuscan Class-Only still takes 4634 seconds to detect an OD test.

We find that Target Pairs is quite ineffective at detecting OD tests, detecting an OD test every 177518 seconds, on average. With Target Pairs detecting fewer OD tests and being less cost-effective than Tuscan Intra-Class, Tuscan Intra-Class is likely the best technique for developers to balance the cost and detectability of OD tests. Lastly, we find that Tuscan Inter-Class is the most cost ineffective technique to detect OD tests, detecting an OD test every 16112163 seconds, on average. The main advantage of using Tuscan Inter-Class is the guaranteed detection of all OD tests that depend on just one other test (Section 3.4). That being said, unless one must detect all possible OD tests, Tuscan Intra-Class detects slightly fewer OD tests while being much more cost-effective.

5.4 RQ4: Minimal Test Orders Needed

While each technique produces many test orders, not all of those test orders are needed to detect possible OD tests. For each technique, we also compute the ideal, minimal number of test orders needed to detect the possible OD tests the technique is able to detect. We consider an OD test detected if there is at least one passing test order and one failing test order, so a minimum of two test orders are needed. Since we know the exact polluters, victims, cleaners, brittles, and state setters [28], we also know which test orders pass/fail for each OD test. We compute the minimal set of test orders that can detect all known OD tests by greedily selecting the produced test order that detects the most undetected OD tests, continuing until all are detected by this minimal set of test orders.

We see that on average the number of minimal test orders needed is rather small, fewer than 4 for each technique, This number is in sharp contrast to the total number of test orders each technique would produce, e.g., the average of 769148.3 test orders for Tuscan Inter-Class drops all the way down to 3.3.

The number of minimal test orders needed suggests that a form of prioritization or selection of test orders can be quite effective

in helping to detect OD tests efficiently. There are some few test orders that can detect a large percentage of OD tests in a subject, so heuristics that can favor running such test orders first can greatly decrease the cost of detection. A developer can then just run the higher priority test orders up to a desired threshold and still feel confident about detecting all the OD tests. These results motivate future work in test order prioritization or selection.

6 THREATS TO VALIDITY

The results of our evaluation may not generalize to other subjects. Our evaluation subjects utilize a prior dataset of OD tests with an extensive categorization of the tests within [28]. These OD tests were detected from popular open-source Java projects.

Our experimental results are based on simulations that require knowledge of the characteristics of the tests, e.g., which tests are polluters and cleaners. We rely on the previous dataset's classifications, but there may be misclassifications or incomplete data. We checked and reran the categorized tests in the dataset to confirm their characteristics, and we corrected some misclassifications, which we shared with the authors of the prior work. We also simulate runtime based on the average time to run all tests in a single test order. We believe the time for running all tests, even in different test orders, remains roughly similar.

7 RELATED WORK

Luo et al. previously conducted the first empirical study on flaky tests [19]. They found OD tests to be among the top three most prominent categories of flaky tests. Since then, there have been other empirical studies on flaky tests in different domains and applications, such as pursuing developers' perspectives on flaky tests [5] or their effects on the development lifecycle [12], on Android applications [25], or on UI testing [23]. Flaky tests are prevalent in both open-source projects and in industry [10–12, 16, 21, 22].

Zhang et al. proposed DTDetector, which detects OD tests by rerunning tests in random test orders or running pairs of tests [29]. Lam et al. developed iDFlakies, a tool for randomizing test orders and partially classifying the flaky tests into OD tests or non-OD tests [1, 13] in Java projects. Gruber et al. developed FLAPY to detect OD tests in a similar manner for Python projects [8]. Li et al. proposed IncIDFlakies, which makes iDFlakies evolution-aware, by analyzing only tests that are affected by code changes to detect newly-introduced OD tests [17]. Our techniques do not leverage code changes, and therefore we do not compare against IncIDFlakies. Wei et al. proposed detecting non-idempotent-outcome tests, which are tests that fail when run twice in the same JVM [27]; these tests may be indicative of future problems with shared states, resulting in OD tests. Other work has focused on detecting OD tests by analyzing shared states. Gyori et al. proposed PolDet to detect tests that modify shared heap-state without resetting it after execution, meaning they are potential polluters [9]. We also analyze dependencies on heap-state through static fields for Target Pairs. Bell et al. proposed ElectricTest, which tracks what state tests read from and write to, forming dependencies between tests [4]. Gambi et al. followed up on ElectricTest with their technique PraDet, which would additionally run tests that share states together to check whether a test fails [6]. We also track test dependencies, and our techniques aim to cover relevant test pairs. Our techniques aim to cover as many test pairs as possible in a single test order to reduce the overall cost instead of only running test pairs on their own.

Our work builds upon Wei et al.'s prior work on systematically covering test pairs to detect OD tests [28]. Their approach specifically targets covering test-class pairs, which we evaluate as Tuscan Class-Only in this work. We propose Tuscan Intra-Class and Tuscan Inter-Class to cover more test pairs than what Tuscan Class-Only handles. Due to the large number of test orders necessary to cover all test pairs given the constraints of test classes, we further propose Target Pairs to reduce the number of test orders needed by reducing the necessary test pairs that should be covered.

One way to prevent OD test failures is to run each test in isolation to prevent the sharing of states, but doing so greatly increases the cost of testing [20]. Bell and Kaiser proposed a runtime environment to automatically reset the state between tests without running tests in isolation [3]. However, the runtime environment still introduces some extra overhead. Shi et al. proposed iFixFlakies [24] to automatically repair OD tests. They use cleaners to generate patches. In this work, we use their categorization of tests involved in OD tests to conduct our simulations to determine whether an OD test can be detected. Inspired by iFixFlakies, Wang et al. developed iPFlakies for Python projects to automatically repair Python OD tests [26]. Li et al. later improved upon iFixFlakies, proposing ODRepair to repair OD tests that do not have cleaners [18]. They track the shared state from static fields and generate patches that use methods from code-under-test that interact with those static fields. All these techniques require knowing about the OD test and the related other tests (e.g., polluter for a victim). Our techniques that identify a passing test order and failing test order for each OD test enable these other approaches to generate patches. Besides repairing OD tests, Lam et al. proposed enhancing regression testing algorithms to accommodate the effects of OD tests [14]. Their enhancements update the test orders proposed by regression testing techniques, such as regression test selection or test-case prioritization, to include and order tests based on known test dependencies, ensuring the tests do not fail due to the test order changes.

8 CONCLUSIONS

Order-dependent tests (OD tests) are a prominent category of flaky tests and are tests whose outcome depends on the order in which they are run. Prior work has proposed numerous techniques to detect OD tests, often resorting to the generation and execution of random test orders. Recently, a more systematic approach has been proposed to generate test orders so that all test class pairs are covered. In this work, we expand on the systematic generation of test orders by proposing three new techniques. Our techniques can detect 97.2% of known OD tests compared to just 36.0% of OD tests from prior work while running a similar number of test orders. Our evaluation of the number of minimal test orders needed by each technique to detect all possible OD tests reveals a tremendous opportunity for future work to prioritize test orders.

ACKNOWLEDGEMENTS

We would like to acknowledge NSF grant no. CCF-2145774 and Dragon Testing for their software testing research support.

REFERENCES

[1] 2019. iDFlakies. https://github.com/idflakies/iDFlakies.

[2] 2023. Systematically Producing Test-Orders to Detect Order-Dependent Flaky Tests Dataset. https://sites.google.com/view/systematically-detecting-od.

[3] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *International Conference on Software Engineering*.

[4] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[5] 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*.

[6] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *International Conference on Software Testing, Verification, and Validation*.

[7] Solomon W. Golomb and Herbert Taylor. 1985. Tuscan squares – A new family of combinatorial designs. *Ars Combinatoria* 20, B (1985).

[8] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An empirical study of flaky tests in Python. In *International Conference on Software Testing, Verification, and Validation*.

[9] 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*.

[10] 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*.

[11] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *International Symposium on Software Testing and Analysis*.

[12] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *International Conference on Software Engineering*.

[13] 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*.

[14] 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*.

[15] 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*.

[16] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. 2021. When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[17] Chengpeng Li and August Shi. 2022. Evolution-aware detection of order-dependent flaky tests. In *International Symposium on Software Testing and Analysis*.

[18] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing order-dependent flaky tests via test generation. In *International Conference on Software Engineering*.

[19] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*.

[20] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[21] Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[22] Maaz Hafeez Ur Rehman and Peter C. Rigby. 2021. Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[23] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An Empirical Analysis of UI-Based Flaky Tests. In *International Conference on Software Engineering*.

[24] 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*.

[25] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in Android apps. In *International Conference on Software Maintenance and Evolution*.

[26] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A framework for detecting and fixing Python order-dependent flaky tests. In *International Conference on Software Engineering (Tool Demonstrations Track)*.

[27] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting flaky tests via non-idempotent-outcome tests. In *International Conference on Software Engineering*.

[28] 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*.

[29] 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*.

Received 2023-02-16; accepted 2023-05-03