iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests

August Shi
University of Illinois
Urbana, IL, USA
awshi2@illinois.edu

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

Reed Oei
University of Illinois
Urbana, IL, USA
reedoei2@illinois.edu

Tao Xie
University of Illinois
Urbana, IL, USA
taoxie@illinois.edu

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

ABSTRACT

Regression testing provides important pass or fail signals that developers use to make decisions after code changes. However, flaky tests, which pass or fail even when the code has not changed, can mislead developers. A common kind of flaky tests are order-dependent tests, which pass or fail depending on the order in which the tests are run. Fixing order-dependent tests is often tedious and time-consuming.

We propose *iFixFlakies*, a framework for automatically fixing order-dependent tests. The key insight in iFixFlakies is that test suites often already have tests, which we call *helpers*, whose logic resets or sets the states for order-dependent tests to pass. iFixFlakies searches a test suite for helpers that make the order-dependent tests pass and then recommends patches for the order-dependent tests using code from these helpers. Our evaluation on 110 truly order-dependent tests from a public dataset shows that 58 of them have helpers, and iFixFlakies can fix all 58. We opened pull requests for 56 order-dependent tests (2 of 58 were already fixed), and developers have already accepted pull requests for 21 of them, with all the remaining ones still pending.

CCS CONCEPTS

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

KEYWORDS

flaky test, order-dependent test, patch generation, automated fixing

ACM Reference Format:
August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338925

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338925

1 INTRODUCTION

Regression testing is an important part of software development, but it suffers from the problem of flaky tests. Developers run regression tests when they make changes to ensure that the changes do not break existing functionality. Flaky tests can pass or fail even when run on the same code, without any changes. These tests are problematic for software testing in general, and they are particularly problematic for regression testing, because they provide misleading signals to developers regarding the effects of their changes [27, 38]. Typically, when a test fails, the failure indicates a fault introduced by a change, and the developers should debug the change. However, with flaky tests, a test failure may not indicate a fault introduced by a change, and developers can waste time trying to debug a fault unrelated to recent changes [17]. Labuschagne et al. [32] found that 13% of the failed builds studied by them in open-source projects using Travis CI are due to flaky tests. The software industry also widely reports major problems with flaky tests [24, 27, 28, 40, 57], e.g., Luo et al. [38] reported that at one point 73K of 1.6M test failures per day at Google were due to flaky tests.

An important kind of flaky tests are *order-dependent* tests, which pass or fail based solely on the order of the sequence in which the tests run [33, 56]. Each order-dependent test has at least one *test order* (a sequence of of tests in the test suite) where the order-dependent test passes, and at least one other different test order where the order-dependent test fails; if the two test orders do not differ, the test is not flaky solely due to the ordering. Prior work [38, 45] showed that order-dependent tests are among the top three most common kinds of flaky tests. As an example, a widely reported case happened when Java projects updated from Java 6 to Java 7. Java 7 changed the implementation of reflection, which JUnit uses to determine the test order to run tests in. Many tests failed due to the tests being run in a different test order from before, requiring developers to manually fix their test suites [1–3]. Prior work, including ours, developed automated techniques for detecting order-dependent tests in test suites [20, 33, 56]. Furthermore, we released a dataset of flaky tests [33], where about half are order-dependent.

In this paper, we propose a framework, *iFixFlakies*, that can automatically fix many order-dependent tests. Our key insight is that test suites often (but not always) already have tests, which we call *helpers*, whose logic (re)sets the state required for order-dependent tests to pass. We first identify that an order-dependent

test can be classified into one of two types based on the result of running the test in isolation from the other tests. One type is a *victim*, an order-dependent test that passes when run in isolation but fails when run with some other tests. The other type is a *brittle*[1], an order-dependent test that fails when run in isolation but passes when run with some other test(s).

More specifically, our insight for iFixFlakies is that running some helper(s) directly before victims and brittles makes these order-dependent tests pass. Therefore, we can use the code from these helpers to fix order-dependent tests so that they pass even if helpers are not run (directly) before the order-dependent tests. iFixFlakies searches for helpers and, when it can find them, uses them to automatically recommend patches for order-dependent tests. As inputs, iFixFlakies takes an order-dependent test, a test order where the test passes, and a test order where the test fails. It outputs a patch that can be applied to the order-dependent test to make it pass even when run in the test order where it was failing before. The code in the patch comes from a helper, and while simply using all the code from the helper can create a patch, such a patch would be complex and undesirable because helpers typically contain many statements irrelevant to why the tests are order-dependent. iFixFlakies produces effective patches by delta-debugging [55] the helpers to produce the minimal patch for order-dependent tests.

We evaluate iFixFlakies on all 110 truly order-dependent tests from a public dataset [33] that includes the order-dependent tests and their corresponding passing and failing test orders. We find that 100 tests are victims and 10 are brittles. We also find that 58 of these 110 order-dependent tests have helpers, allowing iFixFlakies to propose patches for all 58 of these tests (48 victims and 10 brittles). These patches have, on average, only 24.6% of the statements of the original helper, and 69.5% of these patches consist of only *one statement*. The overall time that iFixFlakies takes to find the first helper and to produce a patch using that helper is only 238 seconds on average. When an order-dependent test has no helper, iFixFlakies takes 341 seconds on average to determine that it cannot produce a patch. These time results show that iFixFlakies is efficient.

We opened pull requests for 56 order-dependent tests with helpers (2 of 58 were already fixed in the latest version of the code). While all patches generated by iFixFlakies semantically fixed the flaky test, not all patches were syntactically the most appropriate. For 28 tests, we created the pull requests using exactly the patch recommended by iFixFlakies, while the remaining ones involved some manual changes, mostly refactorings to make the code more similar to the style of the project. Developers have accepted our pull requests fixing 21 order-dependent tests; the pull requests for the remaining 35 order-dependent tests are still under consideration but none have been rejected.

This paper makes the following main contributions:

(1) **Formalization.** We formally define two different types of order-dependent tests and three different roles of tests that can help in the patching of order-dependent tests.
(2) **Technique.** We present a technique to fix order-dependent tests using helpers. Our technique automatically generates a patch for 58 out of 110 order-dependent tests.
(3) **Framework.** We implement our technique and make publicly available a framework, iFixFlakies [7].
(4) **Evaluation.** We evaluate iFixFlakies on a dataset of 110 order-dependent tests. Using iFixFlakies, we break down the order-dependent tests into 100 victims and 10 brittles, where 58 of these tests have helpers. Furthermore, iFixFlakies is able to automatically fix *all* these 58 order-dependent tests.

2 FORMALIZATION OF TESTS

Order-dependent tests are flaky tests whose results can differ depending on the order in which the tests run. An order-dependent test consistently passes when run in one order but then consistently fails when run in a different order [33, 56].

Let T be the set of all tests[2] in the test suite. A *test order* is a sequence of a *subset* of tests from T. For a test order O that has a test $t \in T$, let $\text{run}_t(O)$ be the result of the test t when run in the test order O; the result can be either *PASS* or *FAIL* consistently; we ignore other flaky tests that have results *PASS* and *FAIL* when rerun in the same test order due to other sources of non-determinism. We use $\text{run}(O)$ to refer to the result of the *last* test in O. We use $[t]$ to denote a test order consisting of just one test t, and use $O + O'$ to denote the concatenation of two test orders O and O'.

Definition 1. *A test $t \in T$ has a **passing test order** or a **failing test order** O if $\text{run}_t(O) = PASS$ or $\text{run}_t(O) = FAIL$, respectively.*

Definition 2. *An **order-dependent test** $t \in T$ has a passing test order O and a failing test order $O' \neq O$.*

We classify an order-dependent test into one of two types: victim or brittle. We also classify other tests related to order-dependent tests into three different *roles*: polluter, cleaner, and state-setter.

2.1 Victim

A *victim* is an order-dependent test that consistently passes when run by itself in isolation from other tests.

Definition 3. *An order-dependent test $v \in T$ is a **victim** if $\text{run}([v]) = PASS$.*

The reason why a victim fails in a failing test order is that there is at least one test that runs before the victim, and these tests "pollute" the state (e.g., global variable, file system, network [56]) on which the victim depends. We call such state-polluting tests *polluters*. Note that a polluter can consist of multiple tests, where the combination of running those tests in a certain order leads to the victim failing.

Definition 4. *A test order (with one or more tests) P is a **polluter** for a victim v if $\text{run}(P + [v]) = FAIL$.*

Figure 1 shows an example (identified by iFixFlakies) of a victim and a polluter from Elastic-Job [4]. The polluter is by itself the test `assertRemoveLocalInstancePath` (or `PT` for short), because it starts the instance (Line 8) but does not shut it down. The victim is the test `assertIsShutdownAlready` (or `VT` for short) that fails on Line 4, which checks whether an instance of a class variable has been shut down. `VT` passes by itself or in test orders where a polluter like `PT` is run after `VT`.

[1] The word "brittle" is commonly used as an adjective but can also be used as a noun.

[2] When we say *test*, for Java we mean *test method* as defined in JUnit.

```
1  // Victim (in ShutdownListenerManagerTest class)
2  @Test public void assertIsShutdownAlready() {
3    shutdownListenerManager.new
       InstanceShutdownStatusJobListener().dataChanged("/test_job
       /instances/127.0.0.1@-@@", Type.NODE_REMOVED, "");
4    verify(schedulerFacade, times(0)).shutdownInstance();
5  }
6  // Polluter (also in ShutdownListenerManagerTest class)
7  @Test public void assertRemoveLocalInstancePath() {
8    JobRegistry.getInstance().registerJob("test_job",
       jobScheduleController, regCenter);
9    shutdownListenerManager.new
       InstanceShutdownStatusJobListener().dataChanged("/test_job
       /instances/127.0.0.1@-@@", Type.NODE_REMOVED, "");
10   verify(schedulerFacade).shutdownInstance();
11 }
12 // Cleaner (in FailoverServiceTest class)
13 @Test public void assertGetFailoverItems() {
14   JobRegistry.getInstance().registerJob("test_job",
       jobScheduleController, regCenter);
15   ... // 12 more lines
16   JobRegistry.getInstance().shutdown("test_job");
17 }
```

Figure 1: Example victim, polluter, and cleaner from Elastic-Job.

A victim may not fail even when a polluter is run before it, as long as a cleaner is run between the two. Intuitively, a *cleaner* is a test order that resets the state polluted by a polluter; when the cleaner is run after a polluter and before its victim, the victim passes.

Definition 5. *A test order C is a **cleaner** for a polluter P and its victim v if $run(P + C + [v]) = PASS$.*

An example of a cleaner is also shown in Figure 1. The test `assertGetFailoverItems` (or `CT` for short) is a cleaner for `PT` and `VT`, because Line 16 of `CT` shuts down the instance that `PT` starts and `VT` checks. Therefore, even if `PT` runs before `VT`, as long as `CT`'s Line 16 successfully executes before `VT`, `VT` passes. We can fix `VT` by inserting the statement from this line of `CT` at the end of `PT`.

2.2 Brittle

In contrast to a victim, an order-dependent test is a *brittle* if the test consistently fails when run by itself in isolation.

Definition 6. *An order-dependent test $b \in T$ is a **brittle** if $run([b]) = FAIL$.*

Intuitively, because a brittle fails in isolation and yet has a passing test order, then its passing test order must contain one or more tests that set up the state for the brittle to pass. We refer to a test order that sets up the state for a brittle as a *state-setter*.

Definition 7. *A test order S is a **state-setter** for a brittle b if $run(S + [b]) = PASS$.*

Figure 2 shows an example identified by iFixFlakies of a brittle and its corresponding state-setter test from WildFly [11]. The test `testPermissions` (or `BT` for short) is a brittle, because it fails when run by itself, due to an `AccessControlException`. The test `testBind` (or `ST` for short) is by itself a state-setter for `BT`, because running `ST` and then `BT` is enough to make `BT` pass.

iFixFlakies finds that the `store.lookup` call of `ST` on Line 18 is the only method call that `BT` needs in order to pass. `store` is a test class variable initialized by the `setup` method of the test class. When `store.lookup` is invoked before Line 11, `BT` passes. When we proposed this fix to the developers of WildFly, they quickly accepted

```
1  // Brittle (in WritableServiceBasedNamingStoreTestCase class)
2  @Test public void testPermissions() throws Exception {
3    ...
4    final String name = "a/b";
5    final Object value = new Object();
6    try {
7      ...
8      store.bind(new CompositeName(name), value);
9    }
10   ...
11   assertEquals(value, testActionWithPermission(JndiPermission.
       ACTION_LOOKUP, permissions, namingContext, name));
12 }
13 // State-setter (also in WritableServiceBasedNamingStoreTestCase class)
14 @Test public void testBind() throws Exception {
15   final Name name = new CompositeName("test");
16   final Object value = new Object();
17   ... // 6 more lines
18   assertEquals(value, store.lookup(name));
19 }
```

Figure 2: Example brittle and state-setter from WildFly.

```
1  def iFixFlakies(odtest, passingorder, failingorder):
2    odtype, polluters, cleaners = minimize(odtest, passingorder,
       failingorder)
3    patches = []
4    if odtype == VICTIM:
5      for polluter in polluters:
6        for cleaner in cleaners[polluter]:
7          patches += [patch(polluter + [odtest], cleaner)]
8    else: # odtype == BRITTLE
9      for statesetter in polluters:
10       patches += [patch([odtest], statesetter)]
11   return patches
```

Figure 3: Pseudo-code for the overall process of iFixFlakies.

our fix and clarified that it works because the lookup call causes "the `WildFlySecurityManager.<clinit>` to run" and running this class constructor resolves the `AccessControlException` of `BT` [12].

Both cleaners (for victims) and state-setters (for brittles) help make order-dependent tests pass when they run in certain test orders. Hence, we refer to cleaners and state-setters as *helpers*. Our insight is that these helpers already contain logic to set the state for their corresponding order-dependent tests.

3 IFIXFLAKIES

We present iFixFlakies to automatically recommend patches for order-dependent tests with helpers. Figure 3 shows the pseudo-code for the overall process. iFixFlakies takes as input an order-dependent test, a passing test order, and a failing test order. By Definition 2, each order-dependent test has at least one passing and one failing test order. Several automated approaches exist for detecting order-dependent tests and their corresponding test orders [20, 33, 56], which can provide all these inputs for iFixFlakies. iFixFlakies has two main components: Minimizer and Patcher. iFixFlakies first calls Minimizer (Line 2) to get the type of the order-dependent test, the minimized polluters/state-setters, and the minimized cleaners. Based on the type of the order-dependent test, iFixFlakies then calls Patcher to create patches corresponding to each helper for the order-dependent test (Lines 7 and 10).

Prior to developing iFixFlakies, we attempted to manually fix some order-dependent tests using just their passing and failing test orders. In this manual process, we found it difficult to *understand* why each test fails, let alone *fix*. However, as part of this process,

```
1  def minimize(odtest, passingorder, failingorder):
2    isolation = run([odtest])
3    # Run in isolation multiple times to confirm it is order-dependent
4    for i in range(RERUN):
5      if not isolation == run([odtest]):
6        raise Exception('Incorrectly classified as order-dependent')
7
8    # Passing in isolation means victim, failing means brittle
9    if isolation == PASS:
10     odtype = VICTIM
11     startingorder = failingorder
12     expected = FAIL
13   else: # isolation == FAIL
14     odtype = BRITTLE
15     startingorder = passingorder
16     expected = PASS
17
18   polluters = set() # State-setters for brittles
19   cleaners = {} # Empty map from polluters to cleaners
20
21   # Get minimal test order that causes odtest to match expected result
22   prefix = startingorder[0:indexOf(odtest, startingorder)]
23   while run(prefix + [odtest]) == expected:
24     polluter = deltadebug(prefix, lambda o: run(o + [odtest]) == expected)
25     polluters.add(polluter)
26     if odtype == VICTIM:
27       cleaners[polluter] = findcleaners(odtest, polluter, passingorder,
           failingorder)
28     # If not configured to find everything, stop
29     if not FIND_ALL:
30       break
31     prefix.remove(polluter)
32   return odtype, polluters, cleaners
```

Figure 4: Pseudo-code for finding minimal test orders.

we found ourselves manually searching for the polluter, cleaner, and state-setter tests for order-dependent tests, which inspired Minimizer. Once we realized the importance of the helpers and how they can be used as the basis for patches, we developed Patcher. Overall, we found that the manual steps that we undertook could be automated by a tool, leading to iFixFlakies, and such automation can save developers' time for fixing order-dependent tests.

3.1 Minimizer

Minimizer aims to find the minimal subsequence[3] of tests, called *minimal test order*, from a passing test order or a failing test order to make the order-dependent test pass or fail, respectively. The minimal test order is "1-minimal", meaning removing any test from the minimal test order will no longer satisfy the criterion [25, 55].

Figure 4 shows the pseudo-code for Minimizer. The input is an order-dependent test and its two test orders. As shown in Lines 4-6, Minimizer first checks whether the order-dependent test consistently passes or fails by itself, rerunning the test `RERUN` number of times (default is 10). If the test consistently passes or fails, it is likely order-dependent. This check should not be needed when the input test is correctly classified as order-dependent, but our evaluation finds that we incorrectly classified one test in our previous work [33]. If the test is truly order-dependent, the isolation result determines whether it is a victim or a brittle (Lines 9-16).

Next, Minimizer proceeds to delta-debug [25, 55] the prefix to find the minimal test order (Line 24). Delta debugging iteratively splits a sequence of elements to find a smaller subsequence that satisfies a criterion. Our general delta-debugging method takes two

[3]The term "subsequence" refers to a potentially non-consecutive subset of elements in relation to the original ordering.

```
1  def findcleaners(victim, polluter, passingorder, failingorder):
2    # Determine cleaner candidates from passing and failing orders
3    candidates = []
4    polluterpos = indexOf(polluter, passingorder)
5    victimpos = indexOf(victim, passingorder)
6    if polluterpos < victimpos:
7      candidates += [passingorder[polluterpos + 1:victimpos]]
8
9    polluterpos = indexOf(polluter, failingorder)
10   victimpos = indexOf(victim, failingorder)
11   candidates += [failingorder[0:polluterpos]]
12   candidates += [failingorder[victimpos + 1:len(failingorder)]]
13
14   # Add all tests as single candidates
15   candidates += [[c] for c in failingorder]
16
17   # Filter out candidates to find actual cleaners
18   cleaners = []
19   for c in candidates:
20     if run(polluter + c + [victim]) == PASS:
21       # If not configured to find everything, just return the first one
22       if not FIND_ALL:
23         return [deltadebug(c, lambda o: run(polluter + o + [victim]) ==
             PASS)]
24       cleaners += [c]
25
26   # Minimize the cleaners, so <polluter, cleaner, victim> passes
27   return unique(map(lambda c: deltadebug(c, lambda o: run(polluter + o +
       [victim]) == PASS), cleaners))
```

Figure 5: Pseudo-code for finding cleaners.

parameters: (1) the sequence to start delta debugging and (2) the criterion (in the form of a function) to check the current subsequence validity at each iteration. For Line 24, a subsequence is valid when running it before the order-dependent test matches the expected result for that test. The delta-debugging output is a minimal test order representing a polluter for a victim or a state-setter for a brittle; ideally the polluter or state-setter consists of only one test. The search for finding a polluter or a state-setter is the same, so our code assigns the final result to the variable named `polluter`, but it is actually a state-setter if the order-dependent test is a brittle.

In practice, after Line 24, a developer would proceed to find a cleaner for the polluter if the order-dependent test was a victim, or proceed to Patcher if the order-dependent test was a brittle. However, for the sake of our experimental evaluation, we introduce the option to find *all* polluters or state-setters from these test orders. If the `FIND_ALL` option is set (Line 29), Minimizer proceeds to find more polluters or state-setters by first removing the found polluter or state-setter and then continuing with the loop that calls delta debugging again (Lines 23-31). The process stops when running the prefix before the order-dependent test no longer matches the expected result. Our evaluation (Section 5.2) shows that finding more polluters or state-setters does not provide substantial benefits in terms of patching order-dependent tests, so in practice one can just use the first handful of found tests of each type.

3.1.1 Finding Cleaners. After finding a polluter for a victim, Minimizer proceeds to find cleaners (Lines 26-27 of Figure 4). Figure 5 shows the `findcleaners` method. It takes as input a victim, a polluter for the victim, a passing test order, and a failing test order. The returned cleaners make the victim pass when they are run between the polluter and the victim.

First, `findcleaners` determines *cleaner candidates*, which are test orders that are potentially cleaners. `findcleaners` finds cleaner candidates using the passing and/or failing test order, depending on

```
1  def patch(order, helpertests):
2    statements = []
3    # Grab statements from helper methods, including setups and teardowns
4    for h in helpertests:
5      statements += get_setup(h) + get_body(h) + get_teardown(h)
6
7    # Create a method within the last helper's class with these statements
8    patchmethod = insert_new_method(test_class(helpertests[-1]))
9
10   # Insert call to patchmethod at start of flaky test (last test in order)
11   insert_call_at_start(patchmethod, order[-1])
12
13   # Delta debug statements such that the order (that was failing) can pass
14   minimalstatements = deltadebug(statements, lambda s:  patchmethod.
       setbody(s).compile() and run(order) == PASS)
15
16   patchmethod.setbody(minimalstatements)
17   return patchmethod
```

Figure 6: Pseudo-code for finding a patch.

```
1     // Victim (in ShutdownListenerManagerTest class)
2     @Test public void assertIsShutdownAlready() {
3   +   // Call to patch method
4   +   new FailoverServiceTest().patch();
5       ...
6     }
7   + // Starting patch method (in FailoverServiceTest class)
8   + public void patch() {
9   +   // statements from @BeforeClass or @Before
10  +   ...
11  +   // 13 statements from cleaner, assertGetFailoverItems
12  +   ...
13  +   JobRegistry.getInstance().shutdown("test_job");
14  +   // statements from @AfterClass or @After
15  +   ...
16  + }
```

Figure 7: Starting code of Patcher for example in Figure 1.

```
1     // Victim (in ShutdownListenerManagerTest class)
2     @Test public void assertIsShutdownAlready() {
3   +   // Call to patch method
4   +   new FailoverServiceTest().patch();
5         ...
6     }
7   + // Final patch method (in FailoverServiceTest class)
8   + public void patch() {
9   +   JobRegistry.getInstance().shutdown("test_job");
10  + }
```

Figure 8: Final code of Patcher for example in Figure 1.

the index of the polluter and victim in these test orders. For the passing test order, if the victim is run after the polluter, then a cleaner must be among the tests that run between the polluter and victim, so these tests in between become a cleaner candidate (Lines 4-7). For the failing test order, a cleaner can be run before the polluter or after the victim, so tests that run before the polluter and after the victim both become cleaner candidates (Lines 9-12). Finding a cleaner is crucial to enable automated search for a patch. To maximize the chance to find at least one cleaner, `findcleaners` also considers *every* individual test as a cleaner candidate, including even both the polluter and the victim (Line 15).

By considering every test as a cleaner candidate, `findcleaners` may even find a cleaner that JUnit would never run between the polluter and the victim. More specifically, when a polluter and victim are in the same class, `findcleaners` may find a cleaner consisting of tests from a different class; JUnit will never run this cleaner between the polluter and victim. `findcleaners` still searches for such cleaners, because their code can be used by Patcher.

For each cleaner candidate, `findcleaners` runs the polluter, the cleaner candidate, and then the victim, checking whether the victim passes in this test order. If the victim passes, then the cleaner candidate is an actual cleaner; `findcleaners` proceeds to delta-debug the cleaner candidate to find the minimal test order (Line 23), with the delta-debugging criterion being that running the polluter, the subsequence from the cleaner, and the victim passes.

If the `FIND_ALL` option is not set, then the first cleaner found is returned. Otherwise, `findcleaners` checks the remaining cleaner candidates, for the set of all unique cleaners. We use this option to find all cleaners as part of our evaluation (Section 5); our results suggest that finding just a few cleaners suffices.

Minimizer takes the returned cleaners from `findcleaners` and adds them to a map from found polluters to found cleaners (Line 27 in Figure 4). The final return value for Minimizer is a tuple of (1) the type of the order-dependent test (victim or brittle), (2) the polluters or state-setters for the order-dependent test, and (3) the map from polluters to cleaners (empty if the order-dependent test is a brittle).

3.2 Patcher

Patcher automatically recommends patches for fixing an order-dependent test using code from helpers. Patcher takes as input (1) the minimal test order where the order-dependent test fails: for a victim, this order is the polluter followed by the victim, and for a brittle, this order is just the brittle; and (2) a helper for the order-dependent test (note that a helper can consist of multiple tests). Figure 6 shows the pseudo-code for Patcher.

First, Patcher obtains all of the statements from the tests in the helper (Line 5). These statements come from not just the body of the tests themselves but also from all the setup and teardown methods of these helper tests. We use JavaParser [8], a library for parsing Java source code, to obtain these statements. Patcher keeps these statements in the order that JUnit runs them in (i.e., statements in `@Before` run first, then statements in the test, and lastly, statements in `@After`). More specifically, `get_setup` obtains the statements from the setup methods (annotated with `@BeforeClass` or `@Before` in the test class or super-classes), `get_body` obtains all statements in the helper test's body, and `get_teardown` obtains statements from the teardown methods (annotated with `@AfterClass` or `@After` in the test class or super-classes). If the helper test has the annotation `expected` [9], which indicates that the test expects a particular exception to be thrown for it to pass, then `get_body` also wraps the statements from the test in an appropriate try-catch block.

Next, Patcher adds code to run the helper code before the order-dependent test in two steps. First, Patcher creates an empty method, referred to as the `patch` method, to store all of the statements from the helper (Line 8). Second, Patcher inserts a call to the `patch` method at the start of the order-dependent test (Line 11). The inserted code creates an instance of the test class using the default constructor and uses that instance to call `patch`. Note that the code shows inserting this call at the start of the order-dependent test, but for a victim, the call can also be inserted at the end of the polluter. Users can configure Patcher to insert the patch at the beginning of the order-dependent test, or at the end of the polluter for victims.

Table 1: Breakdown of the 184 likely order-dependent tests from a public dataset [6].

# of tests	Category
22	in a class with `@FixMethodOrder`
49	`reuseForks` is set to false in `pom.xml`
1	non-order-dependent test
2	out-of-memory when run with iFixFlakies
110	truly order-dependent tests

Figure 7 shows an example of the starting code to be minimized by Patcher. This code is adapted from the example in Figure 1. Line 8 shows the declaration of the new `patch` method. The body of the `patch` method contains all of the statements from (1) the setup method of `FailoverServiceTest`, (2) the cleaner test body (`assertGetFailoverItems`), and (3) the teardown method of `FailoverServiceTest`. The inserted line (Line 4) calls `patch` using a new instance of the helper's test class.

Finally, Patcher delta-debugs the statements from the helper to find the minimal list of statements that can make the order-dependent test pass when run in the minimal test order (Line 14 of Figure 6); the minimal list of statements is also "1-minimal", and the finest granularity is at the level of statements as defined by JavaParser [8]. The delta-debugging method is the same general one as in Minimizer, except this time it is minimizing the list of statements from the helper tests instead of test orders. The delta-debugging criterion for Patcher is that the `patch` method compiles, and the inserted code makes the order-dependent test pass when run in the minimal test order. Patcher returns the `patch` method with the minimal list of statements for the order-dependent test to pass. Figure 8 shows the final code after Patcher runs; the `patch` method contains only one statement (Line 9).

While the order-dependent test can already be fixed by inserting a call to the `patch` method at the start of the order-dependent test, a developer using iFixFlakies can choose to *inline* the statements from the `patch` method directly into the order-dependent test or into the polluter. In some cases, it may be trivial to just inline these statements into the order-dependent test body. However, in general, a developer should decide whether it is best to inline the statements of the helper into the order-dependent test or polluter, or leave them in a separate method. Factors that may influence the developer's decision include the applicability of the `patch` method to other tests and the data encapsulation of the `patch` method.

To further refine how statements invoked from helpers fix the order-dependent test, Patcher could potentially minimize and inline the statements of methods (indirectly) invoked by the helper tests. By minimizing those statements, the developer can be given a patch that is much more specific to the cause of the flakiness. However, it can be difficult to inline statements from code further away from the helper tests. Also, the number of statements in the final patch will likely increase. As such, Patcher currently does not minimize the statements of these invoked methods, and we leave such investigation for future work.

4 EVALUATION SETUP

We released a public dataset of flaky tests, including order-dependent tests, as part of our prior work [33]. Our dataset is split into two sets, comprehensive and extended. For our evaluation, we use the comprehensive set. This set consists of 184 likely order-dependent tests from 19 Maven modules; a Maven module consists of code and tests from the project that the developers organized to be built and run together. Our dataset also has at least one passing and one failing test order for each order-dependent test.

We implement iFixFlakies as a plugin for Maven [10]. For each module in the Maven project, iFixFlakies takes as input order-dependent tests in the module to fix along with a passing test order and failing one for each one. iFixFlakies uses a custom JUnit test runner to run the tests, so iFixFlakies currently recommends patches for only JUnit order-dependent tests in Maven-based projects.

Unfortunately, not all tests in the dataset are well suited for our goal of submitting patches to developers. First, 22 tests are in test classes annotated with `@FixMethodOrder`. This annotation tells JUnit to run the tests within that test class in a fixed order. Since the developers are already aware of the order-dependent tests in their test suite and have taken measures to address them, we omit these tests from our evaluation. Second, 49 tests from the dataset are in modules that use the Maven Surefire parameter `reuseForks` to run each test class isolated in its own JVM. Such isolation removes many of the dependencies between tests and is another way used by developers to accommodate order-dependent tests.

We run iFixFlakies on all the remaining 113 purported order-dependent tests using the passing and failing test orders from our dataset. Some order-dependent tests have more than one passing test order and/or failing test order in the dataset, and we need only one of each for iFixFlakies, so we arbitrarily choose one of each test order to run iFixFlakies. We configure iFixFlakies to find *all* polluters, cleaners, and state-setters for every order-dependent test. For each order-dependent test, we run iFixFlakies on Microsoft Azure with the virtual machine size `Standard_D11_v2`, which consists of 2 CPUs, 14GB of RAM, and 100GB of hard disk space.

Overall, we find 110 truly order-dependent tests. 1 test is misclassified as order-dependent (found to be non-order-dependent through our reruns) and, due to the large number of polluters and cleaners, 2 tests encounter out-of-memory errors from iFixFlakies. Table 1 shows the summary breakdown of the tests from the dataset.

5 EVALUATION

To evaluate the effectiveness and efficiency of iFixFlakies, we address the following research questions:

RQ1: What are the numbers of victims, brittles, polluters, cleaners, and state-setters found by iFixFlakies among test suites with order-dependent tests? How many tests can iFixFlakies fix?

RQ2: What are the characteristics (e.g., size, uniqueness) of the patches generated by iFixFlakies?

RQ3: How much time does iFixFlakies take to find polluters, cleaners, state-setters, and patches?

We address RQ1 primarily to inform researchers and tool developers on which types of order-dependent tests and roles of tests are the most common so that they can be prioritized appropriately. With the main insight of iFixFlakies being to use helpers to propose patches for order-dependent tests, RQ1 also evaluates the frequency of tests that have helpers and therefore the applicability of our insight on order-dependent tests. We address RQ2 to evaluate the effectiveness in terms of size and accepted pull requests concerning the patches proposed by iFixFlakies, and RQ3 to evaluate the

Table 2: Characteristics of the order-dependent tests (OD) in the projects used in our study.

		Number of					Average number of		
ID	Project Name - Module	tests	OD	victims	brittles	victims w/ cleaners	polluters per victim	cleaners per victim	state-setters per brittle
M1	alibaba/fastjson	178	11	4	7	1	1.8	69.8	51.6
M2	apache/incubator-dubbo - m1	110	4	4	0	4	2.5	6.8	n/a
M3	- m2	65	4	3	1	3	5.3	152.0	2.0
M4	- m3	21	1	1	0	1	1.0	2.0	n/a
M5	- m4	40	3	3	0	0	1.0	0.0	n/a
M6	apache/jackrabbit-oak	3,178	2	1	1	0	1.0	0.0	1.0
M7	apache/struts	61	4	4	0	4	1.0	16.0	n/a
M8	dropwizard/dropwizard	80	1	1	0	1	2.0	16.0	n/a
M9	elasticjob/elastic-job-lite	511	6	6	0	5	1.0	29.8	n/a
M10	jfree/jfreechart	2,176	1	1	0	0	1.0	0.0	n/a
M11	kevinsawicki/http-request	163	28	28	0	28	1.0	1.0	n/a
M12	undertow-io/undertow	79	1	1	0	1	4.0	12.0	n/a
M13	wildfly/wildfly	82	44	43	1	0	1.0	0.0	36.0
Total/Average per test		**6,744**	**110**	**100**	**10**	**48**	**1.3**	**10.6**	**40.0**

efficiency of iFixFlakies and thus how it could be integrated into a practical software development process.

5.1 RQ1: Characteristics of Tests

Table 2 shows some summary information about the projects and modules that contain at least one order-dependent test. For each module, the table lists the total number of tests, the number of order-dependent tests, and the breakdown of the number of victims and brittles among those order-dependent tests. Overall, we find that out of 110 order-dependent tests, 100 tests are victims and 10 tests are brittles, so most order-dependent tests are victims.

Table 2 also shows the average number of polluters per victim, cleaners per victim (that have cleaners), and state-setters per brittle that iFixFlakies finds. Each victim has at least one polluter. In the final row for averages, we show the averages computed per test (not per module). On average, we find 1.3 polluters per victim, with a total of 126 polluters for the 100 victims. Note that our search does *not* exhaustively find all polluters for a victim; the polluters that it finds depend on the position of the victim in the failing test order. On average across all victims, the position of a victim in its failing test order is 54.5% (i.e., a victim is just over the halfway position in the failing test order). 91 of the victims have just one polluter, while 9 victims have more than one polluter; the max number of polluters per victim is 6, and the median is 4 polluters per victim. While a polluter can consist of multiple tests that only when run together before the victim lead to it failing (Section 2), we find that only 3 polluters consist of more than one test. Because most polluters consist of only one test, it is practical to assume only one test pollutes the state for a victim, and future work on finding polluters may benefit from focusing on individual tests.

We hypothesized the existence of cleaners among the order-dependent tests in our prior work [33], and using iFixFlakies we find and show the actual number of cleaners per victim and polluter; different polluters for the same victim may have cleaners in common, but we report each cleaner separately per polluter for the same victim, because each one indicates a potential different patch for that victim. We find that 48 victims of the total 100 victims have at least one cleaner, so almost half of all victims can be fixed using the code from their corresponding cleaners. Of these 48 victims, 29 have just one cleaner, while the remaining 19 have more than one cleaner. The average number of cleaners per victim with at least one cleaner is 10.6, and the median number of cleaners per victim is 16 cleaners. In total, we find 1,063 cleaners for all these 48 victims, where each cleaner consists of only one test. As described in Section 3.1.1, when iFixFlakies searches for cleaners, it considers *every* test as a potential cleaner, even when JUnit would not run such a test in between the polluter and victim. From the 48 victims with cleaners, we find 6 with cleaners that JUnit would not run between the polluter and the victim. Interestingly, two of the cleaners are actually the polluters of a victim as well!

We also find that 8 victims have more than one polluter with cleaners. Interestingly, all polluters of these victims have exactly the same cleaners. Based on these results, a developer should use iFixFlakies to search for cleaners in just one polluter to know whether a victim contains a cleaner or not. Different cleaners can produce different patches, but we find that the numbers of statements produced by different cleaners are similar (Section 5.2).

Concerning state-setters, each brittle must have at least one state-setter, and we find a brittle has on average 40.0 state-setters, and the median number of state-setters is 47. The 10 brittles have a total of 400 state-setters. Because all 10 brittles can be fixed using code from one of their state-setters, and 48 victims have cleaners, iFixFlakies can recommend patches for a total number of 58 tests, over half of the 110 truly order-dependent tests. In total, iFixFlakies finds 1,463 helpers to use to recommend patches for the 58 tests.

5.2 RQ2: Characteristics of Patches

Table 3 shows the characteristics of the patches that iFixFlakies recommends, with one patch per helper; we do not show rows for modules with no helpers, namely M5 and M10. For each module, we show the average number of patches per order-dependent test. We also show the average number of unique patches, based on statements, for each order-dependent test per module. For example, M7 (`apache/struts`) has 16.0 patches per order-dependent test, but

Table 3: Characteristics of patches proposed by iFixFlakies; for each module, averages per order-dependent test are shown.

ID	# Patches	# Unique Patches	# Unique Patch Sizes	First Patch		All Patches	
				Avg. # Stmts	Avg. % Stmts from Original	Avg. # Stmts	Avg. % Stmts from Original
M1	80.0	8.9	1.9	1.1	21.4%	2.0	40.4%
M2	6.8	2.2	1.0	7.0	65.8%	5.4	38.6%
M3	114.5	3.5	1.0	1.5	12.6%	1.0	8.2%
M4	2.0	2.0	2.0	5.0	71.4%	4.5	69.0%
M6	1.0	1.0	1.0	2.0	28.6%	2.0	28.6%
M7	16.0	2.0	2.0	2.0	13.3%	4.1	8.0%
M8	16.0	8.0	4.0	2.0	25.0%	4.5	32.6%
M9	35.8	5.8	2.8	1.2	15.0%	1.5	15.3%
M11	1.0	1.0	1.0	1.0	16.7%	1.0	16.7%
M12	12.0	9.0	4.0	1.0	20.0%	2.7	32.5%
M13	36.0	8.0	4.0	13.0	86.7%	1.9	14.4%
Average	**25.2**	**3.2**	**1.5**	**1.9**	**22.6%**	**1.8**	**24.6%**

only 2.0 unique patches per order-dependent test. Overall, while iFixFlakies recommends, on average, 25.2 patches for each order-dependent test across all modules, only 3.2 are actually unique. The overall average in the final row is the average per test across all modules, not the (unweighted) average of averages per module.

Table 3 also shows the average number of unique patch sizes among all patches for each order-dependent test per module; several patches with different statements can have the same number of statements. If the patch size is the most important for a good patch, then it suffices to find just one patch of a certain size instead of finding all the different patches of that size. With only 1.5 unique patch sizes per order-dependent test on average, many patches actually have the same size.

Table 3 also shows some statistics about the sizes of patches for only the first patch (from iFixFlakies trying the first cleaner of the first polluter or the first state-setter) and across all patches. The table shows the average number of statements and the average percentage of the number of statements w.r.t. the number of statements in the original helper (Section 3.2). Across all patches, iFixFlakies recommends a patch with only 1.8 statements on average, and these statements comprise only 24.6% of the statements in the original patch method. In fact, of the 1,463 total patches, 1,013 (69.5%) contain just one statement! When we look into the spread of the patch sizes per order-dependent test, we find that, on average, each order-dependent test has around 90% of their patches with the same size, most often being the smallest size. For example, the average number of statements in the first patch (1.9) is almost equal to the average number of statements across all patches (1.8). Overall, the results suggest that iFixFlakies should search for a few helpers, but not all of them, because the majority of the helpers lead to the same size of patches.

5.2.1 Submitted Patches. We submitted pull requests for 56 of the 58 order-dependent tests with helpers; 2 of the 58 had already been fixed before we submitted pull requests. Table 4 shows the breakdown of the tests corresponding to our pull requests. Developers already accepted pull requests for 21 tests.

While *all* our pull requests are based on the patches generated by iFixFlakies, we sent patches for half of the tests (28) exactly as iFixFlakies recommended, and the remaining half required small, manual changes. When we had to make changes to the patch for the pull requests, the effort was roughly 1-3 minutes per patch, mostly refactorings or simple changes to match the style of the existing code. Existing techniques and tools [13, 14, 46, 49] could help with such manual effort. We believe that developers using iFixFlakies could use such tools for more automation but still examine the patches and manually apply small changes if necessary. We make available the patches that iFixFlakies generates, a more detailed breakdown describing the changes that we made to the patches, and links to the corresponding pull requests on our website [7].

Table 4: Number of tests addressed by pull requests (PRs) based on iFixFlakies patches.

ID	# of Test Fixed by		
	Pending PRs	Accepted PRs	Patcher
M1	1	7	8
M2	0	2	2
M3	0	4	4
M4	0	1	1
M6	1	0	1
M7	0	4	4
M8	0	1	1
M9	5	0	5
M11	28	0	28
M12	0	1	1
M13	0	1	1
Total	**35**	**21**	**56**

Because iFixFlakies fixes an order-dependent test using statements from a helper, the recommended patches may reduce the order-dependent test's fault-detection capability, i.e., make the test miss a fault. However, if a patch does reduce an order-dependent test's fault-detection capability, then the passing test order in which iFixFlakies (may have) found the helper could likely miss the fault as well. iFixFlakies assumes that each passing test order is correct, and the failing test order indicates a fault in the test code, not a fault in the code under test. We do not believe that the scenario where the failing test order indicates a fault in the code under test actually occurred in our evaluation, particularly because developers did not reject our pull requests to fix the order-dependent tests.

Table 5: Average time in seconds that iFixFlakies takes; '*' denotes that the time includes finding some test(s) with no cleaner.

ID	Test suite time	Avg. time to find first				Avg. time to find all			
		polluter	cleaner	state-setter	patch	polluters	cleaners	state-setters	patches
M1	203	92	*523	42	299	113	*1,443	1,748	25,424
M2	8	22	52	n/a	294	48	465	n/a	2,473
M3	206	143	178	21	130	389	7,089	39	54,856
M4	1	2	4	n/a	395	2	25	n/a	572
M5	3	19	*104	n/a	n/a	19	*104	n/a	n/a
M6	189	218	*5,710	50	416	218	*5,710	50	416
M7	4	13	11	n/a	411	13	159	n/a	14,478
M8	7	36	16	n/a	714	57	589	n/a	4,960
M9	24	45	*293	n/a	258	45	*1,434	n/a	11,854
M10	22	16	*1,695	n/a	n/a	16	*1,695	n/a	n/a
M11	2	15	5	n/a	56	15	227	n/a	56
M12	17	32	79	n/a	232	109	1,025	n/a	2,617
M13	3	21	*114	19	463	21	*114	186	4,749
Average	**35**	**29**	**176**	**37**	**186**	**39**	**592**	**1,154**	**9,737**

It should be noted that for M13 (`wildfly/wildfly`), iFixFlakies actually helps fix victims *without* cleaners as well! None of the 43 victims had a cleaner. However, they all share the same polluter, and that polluter is itself the single brittle found. When we apply a recommended patch for the brittle, not only is the brittle fixed, but all of the victims are also fixed. This example showcases one of the complexities of order-dependent tests and how iFixFlakies can even help fix order-dependent tests that do not have helpers themselves. We do *not* count these 43 tests as fixed in our evaluation, because iFixFlakies fixes these tests indirectly.

5.3 RQ3: Performance

Table 5 shows the time that it takes for iFixFlakies to find polluters, cleaners, and state-setters, along with the time to create patches. The table shows for each module the average time (across all order-dependent tests in the module) that iFixFlakies takes to find/create (1) the first polluter, cleaner, state-setter, and patch; and (2) all polluters, cleaners, state-setters, and patches. The time to create patches assumes that a helper has been found and does *not* include the time to find the helper. As a reference for the time taken by iFixFlakies, the table shows the time to run each module's test suite.

Effective use of iFixFlakies would only require finding the first polluter and cleaner (for a victim) or state-setter (for a brittle) so that iFixFlakies can recommend a patch (Section 5.2). If the victim has more than one polluter, then the time for the first cleaner is for the first cleaner of the first polluter. Similarly, the time to the first patch for such a victim is then the patch created from the first cleaner for the first polluter. If the victim has no cleaners, the table reports the time taken by iFixFlakies to search for cleaners for that first polluter, eventually not finding any; we mark such time with a '*' in the table. The overall average time to find the first polluter, cleaner, state-setter, and patch is 29, 176, 37, and 186 seconds, respectively. Once again, the overall averages are over all order-dependent tests, not over modules. Likewise, the overall average for running all tests in a module is "weighted" by the number of tests as well, so modules with more than one order-dependent test have their test suite time counted multiple times, once per each order-dependent test. Compared to this weighted average time to run all the tests, the time to find the first polluter, cleaner, state-setter, and patch is about 0.8x, 5.0x, 1.0x, and 5.3x the time to run all tests, respectively.

On average, iFixFlakies takes 39, 592, 1,154, and 9,737 seconds to find/create all polluters, cleaners, state-setters, and patches, respectively; once again, '*' denotes time that includes searching for cleaners where there are none. Compared to the time to find/create just the first corresponding test/patch, the time is 1.3x, 3.4x, 31.2x, and 52.3x larger. The average time for finding all state-setters is particularly larger than the time for finding the first state-setter due to the large number of state-setters in M1 (`alibaba/fastjson`). The average time for creating all patches is also particularly larger due to the large number of helpers (one per patch).

Note that iFixFlakies performance can be improved, e.g., Patcher could modify the bytecode of the patch code in-memory [23, 35] to avoid compilation during delta debugging, or could instrument the code to allow turning statements on or off during delta debugging similar to metamutants [31, 50]. In general, considering the large amount of time to create all patches and there being fewer unique patches than all patches, we do *not* recommend developers to use iFixFlakies to create all patches using all helpers for each order-dependent test; obtaining just a few appears to suffice.

Overall, the average end-to-end time for iFixFlakies to try to create a patch for an order-dependent test is 287 seconds; the end-to-end includes the time to find the first helper (including the time to find the first polluter for victims) and then to create the corresponding patch. This end-to-end time also includes the cases where an order-dependent test has no cleaner, and iFixFlakies spends time looking for it. If we split the order-dependent tests between those with and without helpers, the time to create a patch for an order-dependent test with a helper is 238 seconds, while the time to fail to create a patch for one without a helper is 341 seconds.

6 THREATS TO VALIDITY

The results of our study concerning the frequency of victims, brittles, polluters, cleaners, and state-setters may not generalize to other projects. We attempt to mitigate this threat by using a dataset of popular and diverse projects from our prior work [33]. We generated this dataset of order-dependent tests using 13 projects from

earlier work on flaky tests [17, 45], and 150 Java projects deemed the most popular on GitHub [5] based on the number of stars that the projects have. Furthermore, iFixFlakies itself or tools that it uses (e.g., JavaParser [8]) may have faults that could have affected our results. We used extensive logging in iFixFlakies, and at least two authors reviewed iFixFlakies's code and logs.

The metrics that we use to evaluate the patches that iFixFlakies creates, e.g., patch size and uniqueness, may not be the most important metrics for determining the quality of patches. Other important metrics include the time taken to run the patched-in code. The patches that iFixFlakies recommends may also not lead the order-dependent test to pass for test orders other than the failing test orders that iFixFlakies checks. To mitigate these two threats, we submitted pull requests for the patches that iFixFlakies recommends. So far, developers have already accepted pull requests for 21 order-dependent tests, and the rest are pending with none rejected.

7 RELATED WORK

Luo et al. [38] reported the first extensive academic study of flaky tests; they categorized flaky tests by studying historical commits of fixes for flaky tests and found order-dependent tests to be among the top three most common categories. Palomba and Zaidman [45] also studied flaky tests and categorized the ones that they found, with order-dependent tests claimed to similarly be in the top three categories. Gao et al. [22] studied flaky GUI tests, and they found tests that change the configurations for later-run tests, resulting in GUI order-dependent tests. We recently released a dataset [6] of flaky tests that we found by rerunning test suites while randomizing their test orders [33]; almost half of the flaky tests found are order-dependent tests, and we evaluate iFixFlakies using these tests.

Zhang et al. [56] proposed discovering order-dependent tests through randomizing the test orders. Huo and Clause [30] studied tests whose assertions depend on input data not controlled by the tests themselves. They called these assertions "brittle", inspiring our naming of brittles as tests with similar kinds of assertions[4]. The difference is that their brittle assertions *may* fail due to the tests using wrong input data that they do not control, while our brittles are tests that *always* fail when run in isolation (without a state-setter running before them). Gyori et al. [26] proposed a technique, PolDet, for detecting tests that change shared state so the state at the end of their run differs from the state at the start of their run. They call these tests "polluters", and our polluters are similar in nature. The difference is that their polluters *may* pollute the state so other tests (potentially future ones) fail, while our polluters *always* pollute the state for some existing victims. Bell et al. [16] proposed a technique, ElectricTest, to detect data dependencies between existing tests in a test suite, and Gambi et al. [20] followed up on ElectricTest with PraDet, which detects when dependencies between tests can actually lead to tests failing in different orders. In this paper, instead of detecting order-dependent tests, our goal is to automatically *fix* order-dependent tests.

Bell and Kaiser [15] proposed VMVM, a technique to tolerate order-dependent tests by restoring the state of the heap between test runs. VMVM adds instrumentation that re-initializes static fields shared between tests to isolate tests from one another with regards to their heap state when run in the same JVM. Muşlu et al. [43] proposed an even more extreme technique for isolation in that each test should not only run in a separate JVM but also in a fresh environment, e.g., a fresh file system. Bell et al. [17] also evaluated how various forms of isolation can help in test reruns to detect which test failures are due to flaky tests. However, all forms of isolation add extra overhead on top of executing tests. In this paper, we use code from helpers that are already in the test suite to (re)set the state for order-dependent tests to pass even when *not* run in isolation but together with the other tests.

Automatic patch generation is a well-studied topic [34, 36, 37, 39, 42, 44, 51, 52]. The goal is to automatically patch faults in the code, exposed by failing tests. These techniques generate patches using a variety of mechanisms such as systematically mutating code, learning from example patches, and symbolic execution. To validate the success of the patches, most of the techniques rely on the outcomes of tests. In this paper, we aim to patch *tests* as opposed to the code under test. We create these patches by searching for helpers among the existing tests, which have code that can be used to make order-dependent tests pass in their respective failing test orders. Daniel et al. [18, 19], Mirzaaghaei et al. [41], and Yang et al. [54] also fixed test code, while Gao et al. [21] and Stocco et al. [47] fixed test scripts for GUI. However, they all fixed tests that become broken due to code evolution, not flaky tests.

8 CONCLUSION

Flaky tests provide misleading signals to developers during regression testing. Prior work has found order-dependent tests to be among the top three common kinds of flaky tests. We present iFixFlakies, a framework for automated fixing of order-dependent tests. Our main insight for iFixFlakies is that test suites already have *helper tests* whose code can help fix order-dependent tests. iFixFlakies searches for helpers and uses their code to propose relatively small patches for order-dependent tests. Our evaluation on 110 order-dependent tests from a public dataset shows that iFixFlakies can automatically recommend patches for 58 of 110 tests. The recommended patches are effective, with 69.5% of them having just one statement. Also, iFixFlakies is efficient, requiring only 238 seconds on average to produce the first patch for an order-dependent test with a helper. The effectiveness and efficiency of iFixFlakies show promise that it may be integrated into a practical software development process. We used patches recommended by iFixFlakies to open pull requests for 56 order-dependent tests (2 of the 58 had already been fixed); developers have already accepted pull requests for 21 tests, and the remaining ones are pending.

ACKNOWLEDGMENTS

We thank Angello Astorga, Owolabi Legunsen, and Lingming Zhang for discussions about flaky tests and their comments on this paper. This work was partially supported by NSF grant nos. CCF-1421503, CNS-1513939, CNS-1564274, CNS-1646305, CNS-1740916, CCF-1763788, CCF-1816615, and OAC-1839010. We acknowledge support for research on flaky tests and test quality from Facebook, Futurewei, Google, and Microsoft.

[4]The term "brittle" or "fragile" test was also used to describe GUI tests that fail due to changes in the interface [29, 48, 53].

REFERENCES

[1] 2012. JUnit and Java 7. http://intellijava.blogspot.com/2012/05/junit-and-java-7.html.
[2] 2013. JUnit test method ordering. http://www.java-allandsundry.com/2013/01/.
[3] 2013. Maintaining the order of JUnit3 tests with JDK 1.7. https://coderanch.com/t/600985/engineering/Maintaining-order-JUnit-tests-JDK.
[4] 2019. Elastic-Job. https://github.com/elasticjob/elastic-job-lite.
[5] 2019. GitHub. https://github.com.
[6] 2019. iDFlakies: Flaky Test Dataset. https://sites.google.com/view/flakytestdataset.
[7] 2019. iFixFlakies Framework. https://sites.google.com/view/ifixflakies.
[8] 2019. JavaParser. http://javaparser.org/.
[9] 2019. JUnit expected annotation. https://junit.org/junit4/javadoc/4.12/org/junit/Test.html.
[10] 2019. Maven. https://maven.apache.org.
[11] 2019. WildFly Application Server. https://github.com/wildfly/wildfly.
[12] 2019. WildFly Bug Report. https://issues.jboss.org/browse/WFLY-11323.
[13] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *FSE*. Hong Kong, China, 281–293.
[14] Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. 2015. Automated software transplantation. In *ISSTA*. Baltimore, MD, USA, 257–269.
[15] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*. Hyderabad, India, 550–561.
[16] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*. Bergamo, Italy, 770–781.
[17] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*. Gothenburg, Sweden, 433–444.
[18] Brett Daniel, Tihomir Gvero, and Darko Marinov. 2010. On test repair using symbolic execution. In *ISSTA*. Trento, Italy, 207–218.
[19] Brett Daniel, Vilas Jagannath, Danny Dig, and Darko Marinov. 2009. ReAssert: Suggesting repairs for broken unit tests. In *ASE*. Auckland, New Zealand, 433–444.
[20] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*. Vasteras, Sweden, 1–11.
[21] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M. Memon. 2016. SITAR: GUI test script repair. *TSE* 42, 2 (2016), 170–186.
[22] Zebao Gao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE*. Florence, Italy, 55–65.
[23] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *ISSTA*. Beijing, China. to-appear.
[24] Google. 2008. Avoiding Flakey Tests. http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html.
[25] Alex Groce, Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. 2014. Cause reduction for quick testing. In *ICST*. Cleveland, OH, USA, 243–252.
[26] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*. Baltimore, MD, USA, 223–233.
[27] Mark Harman and Peter O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM*. Madrid, Spain, 1–23.
[28] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In *ICSE*. Florence, Italy, 39–48.
[29] Clint Hoagland. 2014. Fixing the brittleness problem with GUI tests. https://www.stickyminds.com/articles/fixing-brittleness-problem-gui-tests.
[30] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*. Hong Kong, 621–631.
[31] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *ISSTA*. San Jose, CA, USA, 433–436.
[32] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE*. Paderborn, Germany, 821–830.
[33] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*. Xi'an, China, 312–322.
[34] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*. Zürich, Switzerland, 3–13.
[35] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. In *OOPSLA*. Vancouver, 92:1–92:30.
[36] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *ESEC/FSE*. Paderborn, Germany, 727–739.
[37] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. St. Petersburg, Florida, 298–312.
[38] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*. Hong Kong, 643–653.
[39] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*. Austin, TX, USA, 691–701.
[40] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE*. Buenos Aires, Argentina, 233–242.
[41] Mehdi Mirzaaghaei, Fabrizio Pastore, and Mauro Pezzè. 2012. Supporting test suite evolution through test case adaptation. In *ICST*. Montreal, QC, Canada, 231–240.
[42] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Comput. Surv.* 51, 1 (Jan. 2018), 17:1–17:24.
[43] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *ESEC/FSE*. Szeged, Hungary, 496–499.
[44] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *ICSE*. San Francisco, CA, USA, 772–781.
[45] Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSME*. Shanghai, China, 1–12.
[46] Danilo Silva, Ricardo Terra, and Marco Tulio Valente. 2014. Recommending automated extract method refactorings. In *ICPC*. Hyderabad, India, 146–156.
[47] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual web test repair. In *ESEC/FSE*. Lake Buena Vista, FL, USA, 503–514.
[48] Suresh Thummalapenta, Pranavadatta Devaki, Saurabh Sinha, Satish Chandra, Sivagami Gnanasundaram, Deepa D. Nagaraj, and Sampathkumar Sathishkumar. 2013. Efficient and change-resilient test automation: An industrial case study. In *ICSE*. San Francisco, CA, USA, 1002–1011.
[49] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of move method refactoring opportunities. *TSE* 35, 3 (2009), 347–367.
[50] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation analysis using mutant schemata. In *ISSTA*. Cambridge, MA, USA, 139–148.
[51] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *ISSTA*. Trento, Italy, 61–72.
[52] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *ICSE*. Vancouver, BC, Canada, 364–374.
[53] Rahulkrishna Yandrapally, Suresh Thummalapenta, Saurabh Sinha, and Satish Chandra. 2014. Robust test automation using contextual clues. In *ISSTA*. San Jose, CA, USA, 304–314.
[54] Guowei Yang, Sarfraz Khurshid, and Miryung Kim. 2012. Specification-based test repair using a lightweight formal method. In *FM*. Paris, France, 455–470.
[55] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *TSE* 28, 2 (2002), 183–200.
[56] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*. San Jose, CA, USA, 385–396.
[57] Celal Ziftci and Jim Reardon. 2017. Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale. In *ICSE*. Buenos Aires, Argentina, 113–122.