Repairing Order-Dependent Flaky Tests via Test Generation

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

Chenguang Zhu
The University of Texas at Austin
Austin, TX, USA
cgzhu@utexas.edu

Wenxi Wang
The University of Texas at Austin
Austin, TX, USA
wenxiw@utexas.edu

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

ABSTRACT

Flaky tests are tests that pass or fail nondeterministically on the same version of code. These tests can mislead developers concerning the quality of their code changes during regression testing. A common kind of flaky tests are order-dependent tests, whose pass/fail outcomes depend on the test order in which they are run. Such tests have different outcomes because other tests running before them pollute shared state. Prior work has proposed repairing order-dependent tests by searching for existing tests, known as "cleaners", that reset the shared state, allowing the order-dependent test to pass when run after a polluted shared state. The code within a cleaner represents a patch to repair the order-dependent test. However, this technique requires cleaners to already exist in the test suite.

We propose ODRepair, an automated technique to repair order-dependent tests even without existing cleaners. The idea is to first determine the exact polluted shared state that results in the order-dependent test to fail and then *generate* code that can modify and reset the shared state so that the order-dependent test can pass. We focus on shared state through internal heap memory, in particular shared state reachable from static fields. Once we know which static field leads to the pollution, we search for reset-methods in the codebase that can potentially access and modify state reachable from that static field. We then apply an automatic test-generation tool to generate method-call sequences, targeting these reset-methods. Our evaluation on 327 order-dependent tests from a publicly available dataset shows that ODRepair automatically identifies the polluted static field for 181 tests, and it can generate patches for 141 of these tests. Compared against state-of-the-art iFixFlakies, ODRepair can generate patches for 24 tests that iFixFlakies cannot.

CCS CONCEPTS

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

KEYWORDS

flaky test, order-dependent test, test generation, automated repair

ACM Reference Format:
Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510173

1 INTRODUCTION

Regression testing is an important part of the software development process, but it suffers from the presence of flaky tests. Flaky tests are tests that nondeterministically pass or fail when run on the same version of code [37]. A flaky test failure can mislead a developer about their recent code changes, leading them to debug their changes when the problems are elsewhere. Previous studies have identified a number of root causes for flakiness [23, 37], with test-order dependence as a common cause.

Test-order dependence results in *order-dependent tests*, whose pass/fail outcomes depend on the test order in which they are run. An order-dependent test always passes in one test order, its *passing test order*; but fails when run in another order, its *failing test order*. Unfortunately, the test order is not guaranteed. For example, Java developers using JUnit for unit testing found their tests failing after a change from Java 6 to Java 7 [1, 2, 31]. After the Java upgrade, JUnit started running tests in different test orders, exposing existing test-order dependence in the test suite that only now manifests after the Java version change (and not due to code changes). There has also been much prior research on developing techniques that can automatically detect order-dependent tests [27, 28, 32, 48].

Shi et al. previously proposed iFixFlakies [43] to automatically repair order-dependent tests. They categorize order-dependent tests into two types: (1) A *brittle* is a test that fails when run on its own but passes when some other tests run before; these tests that run before set up the state for the brittle to pass. (2) A *victim* is a test that passes when run on its own but fails when some other tests run before; these tests that run before are "polluting" the state they share with the victim, leading to the victim's failure. iFixFlakies' insight is that these tests that set the state when run before the brittle contain code that can be made into a patch to repair the brittle. For victims, iFixFlakies can look for *cleaners*, which are tests that, when run between a polluting test and the victim, can reset the shared state to allow the victim to pass. iFixFlakies can create a patch out of these cleaners as to repair the victim. However,

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510173

iFixFlakies relies on developers having already written cleaners; if they do not exist, iFixFlakies cannot generate patches.

We propose ODRepair, a technique to repair order-dependent tests without requiring the existing test suite to contain cleaners. ODRepair first identifies the polluted state in-between tests. ODRepair then generates method-call sequences that modify the polluted state, resetting it for the order-dependent test. In other words, ODRepair aims to automatically generate the cleaners that iFixFlakies needs to generate patches for the order-dependent tests.

ODRepair finds the polluted state between tests by capturing the state between the polluter and the victim order-dependent test (we only focus on victims because brittles are guaranteed to have tests that set the state before). We focus on in-memory heap-state shared between tests, as prior work found test-order dependence to come from shared heap-state, specifically reachable from static fields [28, 37, 48]. The goal is to determine which static field is "polluted", i.e., the heap-state reachable from that field right before the order-dependent test runs differs between the passing and failing test orders, and the difference results in a failing order-dependent test.

Given the polluted static field, ODRepair generates a method-call sequence that can modify the heap-state reachable from that static field to reset the shared state. ODRepair first searches through the code base for reset-methods that can access the polluted static field and either change the static field value directly or modify state that the static field reaches. For each reset-method, ODRepair configures Randoop [10], an automatic test-generation tool, to "test" the reset-method by generating method-call sequences that utilize the reset-method in various ways. Our goal of using the test-generation tool is not the standard goal of covering code and finding bugs, but rather to have it generate tests that are cleaners for the order-dependent test. ODRepair sends these generated tests to iFixFlakies to check if any of them are actually cleaners, which iFixFlakies then uses to generate patches for order-dependent tests.

We evaluate ODRepair on 327 order-dependent tests obtained from IDoFT [6], a public dataset of flaky tests. ODRepair automatically identifies the polluted static field for 181 of these tests. Furthermore, ODRepair automatically generates a patch for 141 order-dependent tests. Compared against iFixFlakies, iFixFlakies can generate a patch for 135 of those 181 order-dependent tests. We find 24 tests (13.3%) where ODRepair generates a patch but iFixFlakies cannot. On the other hand, iFixFlakies generates a patch for 18 tests that ODRepair cannot (9.9%). Based on our findings, we recommend that ODRepair and iFixFlakies be used together in a complementary way. A developer can first apply iFixFlakies to check whether the existing tests already contain cleaners. If iFixFlakies is unsuccessful, the developer can then apply ODRepair to generate a cleaner for repairing the order-dependent test.

This paper makes the following contributions:

- **Automated pollution detection.** ODRepair automatically identifies the polluted state by comparing heap-state between different test orders and loading state captured from the passing test order as to isolate the actual polluted state.
- **Generating patches via test generation.** ODRepair guides the use of a test-generation tool to create method-call sequences that invoke reset-methods for the polluted state, creating tests that contain patches for order-dependent tests.
- **Evaluation.** We evaluate ODRepair on 327 order-dependent tests, where we successfully identify the polluted state for 181 and could generate a patch for 141 of those tests.

2 BACKGROUND AND EXAMPLE

An *order-dependent test* is a flaky test whose pass/fail outcome depends on the test order in which it runs. In other words, there exists a test order where the order-dependent test passes (we refer to it as a *passing test order*) and another different test order where the order-dependent test fails (we refer to it as a *failing test order*). Furthermore, to be qualified as an order-dependent test, the test order is the only determining factor for whether the test passes or fails when run on the same code, i.e., running the same test order multiple times always results in the same pass/fail outcome; otherwise, while the test is still flaky, it would be a non-order-dependent test [32].

Shi et al. [43] previously categorized order-dependent tests into two types: brittles and victims. A *brittle* has a failing test order where the test runs by itself, and it has a passing test order where some other tests run before it. Conceptually, these other tests when run before are "setting" up the correct initial state for the brittle test; Shi et al. referred to such tests as "state-setters". On the other hand, a *victim* has a passing test order where the test runs by itself, and it has a failing test order where some other tests run before it. Conceptually, these other tests when run before are "polluting" some shared state between the tests, leading the victim test to run in a polluted state that results in a test failure; Shi et al. referred to such tests as "polluters".

Shi et al. further developed a technique, iFixFlakies, that automatically repairs order-dependent tests. First, iFixFlakies' Minimizer component finds the state-setters and polluters per brittle and victim, respectively, by using delta-debugging [46] on the given passing test order or failing test order for the order-dependent test. Next, for brittles, the insight is that state-setters must exist by definition, and these tests contain code that would set up the state for the brittle to pass. For victims, the insight is that there exist other tests, which Shi et al. termed "cleaners", that, when run in-between a polluter and its corresponding victim, would result in the victim passing. Unfortunately, unlike brittles, such cleaners are not guaranteed to exist. Essentially, iFixFlakies relies on developers to have already written such tests that contain the needed logic for cleaning the polluted state. Once the state-setter/cleaner has been identified, iFixFlakies' Patcher component generates a patch per order-dependent test by minimizing the statements within the identified state-setter/cleaner (also through delta-debugging), resulting in the minimal set of statements that can be added to the order-dependent test so it passes even in the failing test order.

Given that brittles are guaranteed to have state-setters, we focus solely on generating patches for victims. Moreover, Shi et al. found that victims constitute a much larger proportion of order-dependent tests compared against brittles [43], so it is more reasonable to focus on repairing these types of order-dependent tests. For the rest of this paper, the order-dependent tests we refer to are more precisely victims unless otherwise noted.

Example. Consider the example code and tests illustrated in Figure 1, taken from open-source project elasticjob/elastic-job-lite,

```
1  // polluter/victim test class
2  public final class ShutdownListenerManagerTest {
3    @Mock private SchedulerFacade sf;
4    private ShutdownListenerManager slm;
5
6    @Before
7    public void setUp() {
8      JobRegistry.getInstance()
9        .addJobInstance("test_job", ...);
10     slm =
11       new ShutdownListenerManager(..., "test_job");
12     ReflectionUtils
13       .setFieldValue(slm, "schedulerFacade", sf);
14   }
15
16   // polluter
17   @Test public void assertRemoveLocalInstancePath() {
18     JobRegistry.getInstance()
19       .registerJob("test_job", ...);
20     slm.new Listener().dataChanged(...);
21     verify(sf).shutdownInstance();
22   }
23
24   // victim
25   @Test public void assertIsShutdownAlready() {
26     slm.new Listener().dataChanged(...);
27     verify(sf, times(0)).shutdownInstance();
28   }
29 }
30
31 public final class ShutdownListenerManager {
32   private final String jobName;
33   private final SchedulerFacade schedulerFacade;
34
35   class Listener {
36     protected void dataChanged(...) {
37       if (!JobRegistry.getInstance()
38             .isShutdown(jobName)) {
39         schedulerFacade.shutdownInstance();
40       }
41     }
42   }
43 }
44
45 // class with polluted state
46 public final class JobRegistry {
47   private static volatile JobRegistry instance;
48   private final Map<...> schedulerMap;
49
50   public static JobRegistry getInstance() {
51     return instance;
52   }
53   public void registerJob(final String jobName, ...) {
54     schedulerMap.put(jobName, ...);
55   }
56   public void shutdown(final String jobName) {
57     ... = schedulerMap.remove(jobName);
58   }
59   public boolean isShutdown(final String jobName) {
60     return !schedulerMap.containsKey(jobName);
61   }
62 }
```

Figure 1: Example order-dependent test from elasticjob/elastic-job-lite.

which we use in our evaluation. The order-dependent test in this example is `ShutdownListenerManagerTest.assertIsShutdownAlready`. This test is a victim that passes when run on its own but fails when run after the polluter `ShutdownListenerManagerTest.assertRemoveLocalInstancePath`.

The polluter invokes the `registerJob()` method (Line 19), which adds a new entry to the `schedulerMap` (Line 54) contained within the shared `JobRegistry.instance` static field (obtained using `JobRegistry.getInstance()`). This new entry is mapped to the String key `"test_job"`, used as input to `registerJob()` by the polluter. The polluter invokes `dataChanged()` (Line 20), which in turn checks whether the job name (set to `"test_job"` in the `setUp()` method that runs before, Line 11) is shutdown already. The job name is considered shutdown if the `schedulerMap` does not contain an entry for the job name (Line 60). Since the `schedulerMap` indeed contains an entry for `"test_job"`, the method `shutdownInstance()` is invoked on the `schedulerFacade` field (Line 39).

When the victim runs afterwards, it also invokes `dataChanged()` (Line 26). However, the victim expects that the method `shutdownInstance()` is not invoked on the mock object `schedulerFacade` (Line 27). Since the victim did not register any job for `"test_job"`, the execution should not result in `shutdownInstance()` being invoked. However, because the polluter ran beforehand and registered `"test_job"` without removing it from the `schedulerMap` in the shared `JobRegistry.instance`, the check for the presence of `"test_job"` in the map returns true, and `shutdownInstance()` is still invoked once, resulting in the victim failing.

To repair the victim in this example, we would need to remove the `"test_job"` entry from the shared `schedulerMap` after the polluter finishes and before the victim starts. Fortunately, there is a method, `shutdown()`, that can be invoked to remove the entry (Line 57). In fact, just adding the statement

`JobRegistry.getInstance().shutdown("test_job");`

to run after the polluter can ensure the victim passes regardless of its order relative to the polluter. (It is interesting to note that `schedulerFacade.shutdownInstance()` normally does remove the entry from the shared map, but because `schedulerFacade` is a mock object, it actually does not do that.)

3 ODREPAIR

We propose ODRepair to automatically repair order-dependent tests. ODRepair has two main components. The first component, Debugger, takes an identified order-dependent test and its identified polluter, and it automatically detects the polluted state that causes the order-dependent test to fail. The second component, Generator, uses the polluted state identified by the Debugger to generate a patch that will reset that polluted state. The goal is that applying the patch will make the order-dependent test also pass when run in its original failing test order.

3.1 Debugger

The Debugger takes as input a known order-dependent test `t` and its corresponding polluter `p`. Figure 2 shows the overall algorithm for Debugger. The polluter `p` can be obtained using the Minimizer component of iFixFlakies (Section 2). Note that ODRepair is meant to only handle victim order-dependent tests, so there must exist a

```
1  # Input: order-dependent test t,
2  #        polluter p
3  # Output: polluted field
4  def iFixPlus(t, p):
5      # Capture state of t in failing test order using p
6      xmls_f = state_capture([p, t])
7
8      # Capture state of t in passing test order
9      test_result = run_tests([t, t])
10     if test_result == PASS:
11         xmls_p = state_capture([t, t])
12     else:
13         xmls = state_capture([t, t])
14         xmls_p = state_capture_eager([t], xmls.keys())
15
16     # Find fields where xmls differ
17     diff_fields = diffing(xmls_f, xmls_p)
18
19     # Try to set each diff field in the failing order
20     for field in diff_fields:
21         test_result = run_and_set_field([p, t],
22                                 field, xmls_p[field])
23         if test_result == PASS:
24             return field
25
26     return NOTFOUND
```

Figure 2: Algorithm for the Debugger.

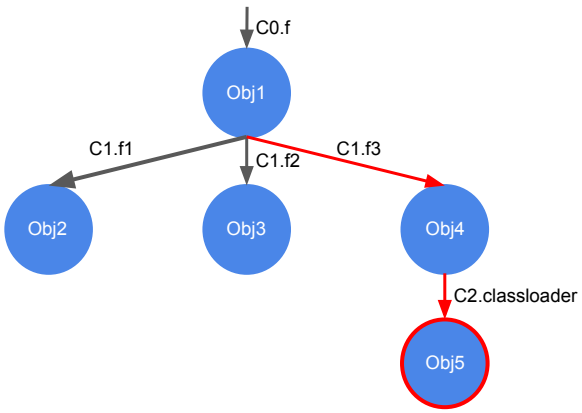

Figure 3: Object graph rooted at static field `C0.f`.

polluter. The goal of the Debugger is then to identify the shared state that the polluter modifies in such a way that the order-dependent test fails in the failing test order. While there can be many types of shared state, such as heap-state, file system, databases, etc., heap-state was found to be one of the most common shared state that leads to test-order dependencies [37]. Furthermore, much prior work has focused on order-dependent tests due to polluted heap-state [15, 17, 18, 28]. As such, we focus on polluted heap-state, particularly pollution reachable from static fields in Java, which prior work found to be the main means for JUnit tests to share heap-state with each other [17, 28].

First, we capture the heap-state right before the order-dependent test `t` in a failing test order (Line 6). We use the minimal failing test order that consists of just the polluter `p` running before `t`, i.e., `[p, t]`. To capture the heap-state, we use the same logic from PolDet [28]. PolDet is a technique that captures and saves heap-state at points during test execution to later compare whether the execution modified the heap-state. PolDet captures heap-state in Java by first finding all the classes loaded at the capture point. PolDet then finds all the static fields contained within these classes. These static fields form the roots of an object graph representing the current heap-state. Figure 3 illustrates an object graph rooted at static field `C0.f`, where nodes are objects and the edges represent instance fields that point to other objects; the edges are labeled by their field name (including the class name). For example, if we follow fields `C0.f` -> `C1.f3`, we would reach the object `Obj4`.

PolDet efficiently creates a representation of this object graph by serializing the objects the static fields reference into an XML format. We use the same approach for our `state_capture()` function, where we put the capture point right before the running of the last test in the test order, namely the order-dependent test. We store the serialized forms of the object states in the failing test order into `xmls_f`, which is a map from the static field name to its corresponding object's XML representation.

We also capture the heap-state right before the order-dependent test in a passing test order. While there can be many possible passing test orders, we want to avoid running too many other tests before the order-dependent test, because these other tests may not be using any classes or static fields that are shared with the order-dependent test. As such, there is no point in capturing heap-state rooted from these static fields, because they should not have an effect on the order-dependent test outcome. Another option is to use the minimal passing test order, i.e., running just the order-dependent test by itself. However, we cannot use this test order either, due to Java's lazy classloading [36]. In lazy classloading, classes are not loaded until they are used for the first time. As such, when `t` runs without any tests before it, no classes would be loaded right before it runs, meaning there would also be no static fields from which to capture heap-state. To overcome this challenge, we instead use a different passing test order consisting of running the order-dependent test twice in a row, i.e., `[t, t]`. After running `t` the first time, all the classes it uses would be loaded. We can then capture the state after the first `t` runs but before the second `t` starts.

However, we need to check whether the order-dependent test still passes after running it twice (Line 10). There is the risk that the order-dependent test `t` is also a polluter for itself, meaning the order-dependent test fails when run the second time; Wei et al. termed these tests as *non-idempotent-outcome tests* [45]. The state after a non-idempotent-outcome test run would not be the correct state, for it has "polluted" itself. For these non-idempotent-outcome tests, we still capture the state at the same point, but we only record the static fields available at that capture point (the keys in the returned map from field to XML representation, Line 13). We then use those fields as reference and do a form of eager classloading, similar to as proposed in PolDet to overcome challenges from lazy classloading [28]. Namely, we record the classes that are used and forcefully load them at the capture point during the minimal passing test order, `[t]` (Line 14). Eager classloading does not adhere to how

Java by default loads classes, which can cause unexpected behavior, e.g., a class may have static state whose initialization depends on the order in which the classes are loaded. As such, we only rely on this approach when we cannot simply run the test twice without it polluting itself. Fortunately, we did not encounter many non-idempotent-outcome tests in our evaluation, with only 33 out of 327 order-dependent tests being non-idempotent-outcome tests.

With both the static fields and serialized heap-state from the failing test order and passing test order, we proceed to diff the XML serialized form of the objects rooted at each field shared in common between the the two test orders (Line 17). However, not all static fields with different object states necessarily lead to the failing order-dependent test. To isolate the true static field that references the polluted heap-state, we iterate over each differing field and deserialize the XML representation corresponding to that static field from the passing test order back into an object. We then load this deserialized object into its corresponding static field when running in the failing test order `[p, t]`, loading it in right before the order-dependent test runs (Line 21). If the test passes now in the failing test order after setting the static field state to its original value from the passing test order, we identify that field to be the root of the pollution, which we refer to as the *polluted static field*. If we cannot use any of the static fields to make the order-dependent test pass, we simply return `NOTFOUND` (Line 26) and cannot proceed further with repairing this order-dependent test.

3.1.1 Reusing State for Deserialization. Deserializing XML back into objects presents additional challenges. The object graph rooted at a static field can eventually reach objects unique to the running JVM at the capture point, e.g., reaching the classloader or dynamically-generated proxy objects. While the entire state can be serialized into an XML format, that same XML cannot be deserialized back into a viable object in a different running JVM. As long as any part of the XML representing heap-state rooted at a static field cannot be deserialized, we fail to deserialize that entire object graph for the static field.

To overcome this challenge, we introduce a means to reuse heap-state from the currently running JVM. The insight is that we are trying to deserialize XML representing an object graph that is the same structure as the object graph currently loaded in memory. For example, in Figure 3, the object referenced by following fields `C0.f -> C1.f3 -> C2.classloader` is actually an instance of a `ClassLoader`. While this object can be serialized into XML, that same XML representation cannot be safely deserialized into another running JVM. When we encounter such an object during deserialization, we instead navigate through the object graph in the currently running JVM (during the failing test order) and follow the same field structure, i.e., by following the highlighted red arrows in Figure 3. We eventually reach what object is currently referenced by that chain of fields, i.e., `Obj5`. We then use that current object in place of what object the deserialization process was planning on creating, essentially allowing that object to be loaded back in place.

Note that while we can use this process to avoid issues where we cannot safely deserialize parts of the captured heap-state from a previous JVM run, there is the risk that the actual pollution in the heap-state is exactly in the part we are reusing. However, given the typical types of objects we cannot deserialize, namely internal Java classes, it is unlikely that changes to these objects result in pollution that makes an order-dependent test to fail. Furthermore, we would not have been able to deserialize them anyway without this additional recovery attempt.

3.2 Generator

The Generator takes as input the order-dependent test and its polluted static field as identified by the previous Debugger. The goal of the Generator is to generate a patch that, when applied right before the order-dependent test, can make the order-dependent test pass when run after the polluter in the failing test order.

The Generator itself consists of three sub-components: Reset-Method Finder, Test Generator, and Patch Generator.

3.2.1 Reset-Method Finder. Given the the polluted static field, Reset-Method Finder scans the bytecode of the code under test (plus any additional third-party libraries) to find potential methods that can reset the state for that polluted static field. We refer to such methods as *reset-methods*. Reset-Method Finder determines potential reset-methods based on the following heuristics.

First, if a method uses the `PUTSTATIC` bytecode instruction where the operand is the polluted static field, then Reset-Method Finder considers the method as a potential reset-method. The reasoning is that the method is directly setting the value of the polluted static field. As such, the method may either directly reset the field back to a default state, or sets that state based on some input arguments.

Second, from our initial inspection of order-dependent tests in our dataset, we found a substantial ratio of them (27 out of 68 tests from 8 projects) involve pollution of fields of a Java Collection data structure type, e.g., type `Map`. For example, the polluter would add entries to a shared `Map` while the order-dependent test expects that map to be empty when it starts to run (similar to the example illustrated in Figure 1). The way to reset these data structures is to use its API methods that modify the data structure state, e.g., `clear()`, `put()`, `putAll()`, `putIfAbsent()`, and `remove()` for a `Map`. We therefore look for methods that invoke these predetermined data-structure-modifying methods while operating on fields (which can now be instance fields) defined within the type of the polluted static field. The reasoning is that these methods can be invoked in ways such that they can eventually modify the polluted data structures back to the correct state.

3.2.2 Test Generator. Given the set of potential reset-methods, the Test Generator aims to generate tests that invoke the reset-methods in ways that can potentially reset the state referenced by the polluted static field when run after the polluter and before the order-dependent test. For this task, we leverage automatic test-generation tools. An automatic test-generation tool would generate method-call sequences, focused on a specific part of the code being tested, with the goal to cover the code under test and find bugs in the code. For our implementation of Test Generator, we specifically use Randoop [10, 39], a tool that generates unit tests effectively and efficiently by randomly constructing method-call sequences. Randoop needs as input the method it should focus on testing. We therefore pass each reset-method found from the Reset-Method Finder as an input method to Randoop. Randoop then generates

a test class per each reset-method, where individual tests within invoke the reset-method in various ways.

However, we need to configure Randoop in additional ways so it can effectively generate useful method-call sequences.

Mining literals from the code under test. A reset-method may need to be invoked with input arguments. Randoop has a default constants pool it uses to provide the input arguments, but often these defaults are insufficient for the reset-method to actually reset the polluted static field. To help provide more meaningful inputs, Test Generator mines literals, such as String constants, from the bytecode of the code under test. The insight is that these literals are likely already used by the polluter to modify the shared state. Consider the example code in Figure 1, where the polluter calls `registerJob()` with the literal String `"test_job"`, which is used as a key to a `Map` in the backend. A reset-method that we found, `shutdown()`, takes a String input, which in this case has to match with the same literal the polluter used to modify the shared state, re-using that literal as a key for resetting the `Map`.

We limit the scope of this literal mining to the code in the order-dependent test class and the class that contains the polluted static field as to restrict the search space of possible inputs Randoop would use when creating method-call sequences. We pass these literals to the constants pool of Randoop, which are utilized to generate input values of method-call sequences (through its literals file [11]).

Additional helper methods. Besides reset-methods needing specific literal inputs to reset the polluted static field, they may also need more complex inputs. For example, if the reset-method is an instance method, the reset-method would need a callee instance on which it can be invoked. As such, Randoop needs to invoke other methods that return the callee instances on which to invoke the reset-method. However, letting Randoop try all possible methods in the project is not a good idea, as it becomes less likely for Randoop to generate the right method-call sequence to reset the state.

For each reset-method provided by the Reset-Method Finder, Test Generator configures Randoop to use only that reset-method along with other helper methods that return relevant values that Randoop should use along with the reset-method. These helper methods include those that return the callee type of the reset-method and any methods that use `GETSTATIC` of the polluted static field. These latter methods are potentially getter methods that return the polluted static field, which the reset-method may need access to as to reset state reachable from that same polluted static field. The output is a test class for each reset-method, containing test methods that invoke the reset-method along with the helper methods.

Coarse-grained test generation. Randoop can also receive as input entire classes for testing, where it would then generate tests that invoke combinations of methods from the provided classes. We therefore also try using Randoop to generate tests based on likely relevant classes. The intuition is that we may still miss important methods through the more fine-grained search on reset-method and helper methods, so letting Randoop use all methods in relevant classes may be enough to cover all important methods. This alternate approach provides to Randoop all the classes referenced within the class that contains the polluted static field and all the classes that are transitively referenced by the class type of the polluted static field. The approach then focuses on the top K classes

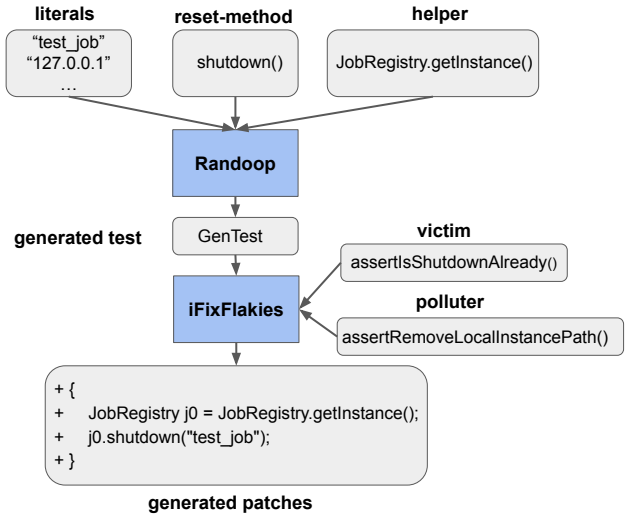

Figure 4: Overview of Generator.

among this set of classes, where the classes are ranked by their distance from the polluted static field type, i.e., the classes directly referenced by the polluted static field type are ranked first, and other classes that these classes in turn reference are ranked second, and so on. Heuristically, we choose 10 as the value of K, restricting the search space as to avoid generating tests that use too many different methods, especially as the ones "further away" are less likely to have an effect on the pollution. Randoop takes as input all these classes for which to generate tests.

Ultimately, Test Generator outputs a separate test class for each reset-method, along with an additional test class generated by Randoop using the classes relevant to the polluted static field.

3.2.3 Patch Generator. Finally, Patch Generator aims to check whether any of the generated tests from Test Generator contain code that actually resets the shared state, essentially consisting of a patch for the order-dependent test. Patch Generator leverages iFixFlakies [43] to do the check and generate the patch. Patch Generator sets up iFixFlakies to run just the order-dependent test, its polluter, and the set of generated tests to determine if any of those generated tests are actually cleaners for the order-dependent test and polluter pair. For each generated test class, we run iFixFlakies with the test methods within that generated test class along with the polluter and order-dependent test to see whether any of those generated test methods can reset the shared state and make the order-dependent test pass when run after the polluter. If one of them does turn out to reset the shared state, then iFixFlakies minimizes the statements within the test, producing a patch that, when applied at the beginning of the order-dependent test, allows it to pass even when run after the polluter.

Figure 4 illustrates the overall Generator process (using provided reset-methods), once again following the example from Section 2, Figure 1. The inputs are the reset-method (in this case `shutdown()`), literals mined from the code (including String `"test_job"`), and

a list of helper methods (including `JobRegistry.getInstance()` that returns the relevant polluted static field). These inputs are passed to Randoop, which in turn generates a test class (`GenTest`). This test class is passed as input to iFixFlakies to try and generate a patch out of the test methods from that test class, finally producing a patch that repairs the order-dependent test, using both `JobRegistry.getInstance()` and `shutdown()`.

4 EVALUATION SETUP

We evaluate ODRepair on a large set of order-dependent tests. We start with the flaky tests contained within IDoFT [6], a publicly available dataset of flaky tests found in open-source GitHub Java projects, detected using automated flaky-test detection tools such as iDFlakies [4] or NonDex [3]. We are only interested in the 598 flaky tests marked as order-dependent tests in this dataset. We ran iDFlakies [4], a tool to detect order-dependent tests by rerunning tests in different test orders, to obtain the passing and failing test orders per order-dependent test (we configured iDFlakies to run 100 random test orders, as Lam et al. did in prior work [32]). We found one test's project to no longer exist, and we found 211 tests to be non-order-dependent, i.e., their flaky pass/fail outcome is not completely due to test order, or we could not obtain both passing and failing test orders; we exclude these tests from our evaluation.

We ran each order-dependent test in isolation to determine brittles and victims (Section 2). We found that 59 of these tests are brittles and 327 are victims, in line with Shi et al.'s prior findings that the majority of order-dependent tests are victims [43]. Given that iFixFlakies can repair brittles but not always victims, we use for our evaluation only the 327 victims, spread across 42 projects.

For the Debugger component of ODRepair, we use XStream [13] to serialize objects into XML form and deserialize them back into objects. We further modify XStream to implement our technique of reusing existing state when we are unable to deserialize objects from a previous JVM run (Section 3.1.1). To compare the XML representations of objects, we use XMLUnit [12]. We incorporate the state capture logic into iDFlakies as to ensure we can control the test order and to capture state at the right point, right before the order-dependent test. We also set a timeout of 24 hours per order-dependent test for identifying the polluted static field.

For the Generator component of ODRepair, we use Apache BCEL [5] to analyze the Java bytecode, both for finding reset-methods and to mine the literals from the project code. When configuring the automatic test-generation tool Randoop, we limit the maximum number of test methods per test class to 500, and we limit the maximum lines of code per test method to be 100. We also allocate at most 60 seconds for Randoop to generate tests. We use a much smaller timeout for Randoop, because we found Randoop would actually terminate much earlier, on average 28 seconds per use, meaning 60 seconds is typically more than enough.

5 EVALUATION

To evaluate ODRepair, we address the following research questions:

- **RQ1:** How effective is ODRepair at automatically identifying the polluted static field for order-dependent tests?
- **RQ2:** How effective is ODRepair at generating patches for repairing order-dependent tests?

5.1 Identifying Polluted Static Fields

Table 1 shows the results of running the Debugger component on the 327 order-dependent tests from our dataset. The table shows per project the number of order-dependent tests, average number of static fields at the capture point, the average number of static fields referencing state that differed between the passing and failing test orders at the capture point, and the number of order-dependent tests for which we could successfully identify the polluted static field. The table also shows the average time for the Debugger per order-dependent test (successful or not). While order-dependent tests access a large number of static fields (average 2149.9), not many of them reference differing state between the passing and failing test orders (average 30.8). Overall, ODRepair can automatically identify the polluted static field for 181 order-dependent tests. The overall average time to run is 1820.9 seconds (median 33.4 seconds). This large time is mostly from projects with a large number of static fields per order-dependent test, e.g., spring-projects/spring-boot or vmware/admiral. Furthermore, there are several outlier tests whose overall time is much higher than the others (more than 1.5 times the interquartile range higher than the third quartile). If we exclude these outlier tests, the average time drops down to 51.5 seconds.

There are 146 order-dependent tests where ODRepair does not identify the polluted static field. From our inspection, the reasons that ODRepair does not identify the polluted static field for these tests can be broken down into three main categories.

Out-of-Resources. ODRepair had out-of-memory errors for 46 tests and timed out for 2 tests. We note that all these tests are from the same project, apache/hadoop, one of the largest projects from our dataset. While apache/hadoop does not have the most number of static fields per order-dependent test, the reachable state from each of the static fields is very large, and our approach to serialize all that state in XML form results in the numerous out-of-memory errors. The timeouts also occur due to having to process this large state (note that ODRepair does not time out for any other project, suggesting our preset timeout value is reasonable).

Serialization/Deserialzation Errors. For 46 tests, ODRepair does not identify the polluted static field but also encounters errors from using XStream and cannot proceed with serializing/deserializing the state. The polluted static field could be among these fields where XStream cannot serialize/deserialize properly, but we cannot identify it automatically due to those errors. We encounter serialization errors for 13 tests and deserialization errors for 33 tests. In the case of serialization errors, we find the serialization would interact poorly with the underlying test running process, preventing the process to exit properly. Note that a serialization error would actually prevent ODRepair from proceeding to deserialization (there is no captured state to deserialize). Concerning deserialization errors, some examples include trying to deserialize instances of anonymous lambda classes or proxy classes, where these types are dynamically generated during the execution of the passing test order. As such, we cannot deserialize instances of these classes in a new JVM execution, and we also cannot re-use any object state from the current running JVM for these parts of the heap-state (Section 3.1.1).

Miscellaneous. The remaining 52 tests are such that there are no errors from serializing/deserializing, but still ODRepair cannot

Table 1: Breakdown of debugging polluted state on order-dependent tests.

Project	# OD Tests	Avg # Static Fields	Avg # Static Fields w/ Diff	Avg Time (s)	# Successfully Debugged Tests
Activiti/Activiti	11	5616.8	0.0	95.3	0
alibaba/fastjson	1	487.0	4.0	6.3	1
alibaba/Sentinel	3	879.0	9.0	356.9	3
apache/cayenne	1	2969.0	49.0	592.9	0
apache/geronimo-batchee	1	123.0	16.0	22.4	1
apache/hadoop	74	3533.8	39.3	891.7	26
apache/hbase	1	802.0	21.0	115.4	0
apache/incubator-dubbo	19	1215.8	45.7	499.0	12
apache/incubator-ratis	1	650.0	15.0	45.1	0
apache/jackrabbit-oak	1	727.0	14.0	22.3	1
apache/karaf	1	1048.0	17.0	401.5	1
apache/shardingsphere-elasticjob	2	2054.0	13.0	325.9	1
apache/skywalking	1	38.0	3.0	19.6	1
c2mon/c2mon	1	9078.0	0.0	107.8	0
CloudSlang/cloud-slang	3	620.0	21.0	27.6	0
ConsenSys/tessera	6	1760.2	15.7	90.3	3
ctco/cukes	1	1599.0	779.0	1406.9	0
danfickle/openhtmltopdf	1	1220.0	30.0	129.5	0
dropwizard/dropwizard	1	1673.0	5.0	23.4	0
elasticjob/elastic-job-lite	4	784.0	12.8	30.4	4
google/java-monitoring-client-library	1	493.0	9.0	115.1	1
jenkinsci/remoting	9	42.9	11.7	14.0	0
jitsi/jicofo	1	3356.0	0.0	32.1	0
jnr/jnr-posix	4	1438.0	30.0	32.8	0
kevinsawicki/http-request	28	439.0	16.6	24.9	28
ktuukkan/marine-api	12	485.0	3.0	6.9	12
networknt/light-4j	14	1647.5	12.5	86.1	10
openpojo/openpojo	5	351.8	20.8	38.0	5
rest-assured/rest-assured	1	3613.0	105.0	1026.0	1
spring-projects/spring-boot	11	17456.9	237.5	43046.1	8
spring-projects/spring-data-envers	2	4251.0	69.5	302.8	0
spring-projects/spring-ws	2	96.0	4.0	5.1	2
tbsalling/aismessages	2	81.0	5.0	7.0	0
Thomas-S-B/visualee	46	33.5	3.0	3.9	46
tools4j/unix4j	1	1477.0	2.0	12.3	0
undertow-io/undertow	1	2133.0	23.0	50.2	1
vaadin/flow	1	821.0	7.0	33.6	1
vmware/admiral	1	6980.0	120.0	32078.8	1
wikidata/wikidata-toolkit	2	29.0	3.0	2.9	2
wildfly/wildfly	37	636.4	17.0	22.4	0
winder/Universal-G-Code-Sender	6	2454.5	24.7	372.9	5
yangfuhai/jboot	6	869.7	17.3	62.6	4
Sum | Mean x3 | Sum	327	2149.9	30.8	1820.9	181

automatically identify the static field where using its value from the passing test order can make the order-dependent test pass in the failing test order. From our inspection, first, two tests access files outside the heap-state, and changes to these shared files result in failed order-dependent tests. We currently do not track state outside the heap. Second, 48 tests initialize variables that reference services that they start up. For example, order-dependent tests in project wildfly/wildfly all rely on the underlying JNDI service [7]. Tests interact with these services and their own shared state not through static fields but rather through specific API calls. Currently, we only track changes to heap-state accessible through static fields. Third, one test from danfickle/openhtmltopdf changes the warning level of global logging by modifying environment variables that we do not track. Finally, for one test in dropwizard/dropwizard, the polluter changes a static field to reference a new output stream instead of the default `System.out`. The later order-dependent test specifically checks if that static field references `System.out`, not just whether the two streams are equivalent in state or not. Our approach only checks if states are equivalent and does not try to match the same field references as in the passing test order.

RQ1: *Overall, ODRepair can automatically identify the polluted static field for 181 out of the 327 order-dependent tests. ODRepair takes on average 1820.9 seconds to analyze all 327 order-dependent tests. However, excluding the order-dependent tests with outlier times makes the average drop to 51.5 seconds.*

5.2 Generating Patches

We use the 181 order-dependent tests for which we identified the polluted static field as input to the Generator. Table 2 shows the results of running the Generator on these tests. For each project, we show the number of order-dependent tests for which we could find the polluted static field (taken from Table 1) and the average number of test classes and test methods generated per order-dependent test in that project. Note that we generate one test class per reset-method and then one additional test class by analyzing the related classes (Section 3.2.2). There may only be one test class generated if the Generator cannot identify any reset-methods for the polluted static field. We see on average 3.0 test classes and 679.0 test methods generated per order-dependent test. We also show the number of order-dependent tests for which we could generate a patch. Overall, we generate patches for 141 order-dependent tests. We also show the time to prepare inputs for Randoop (identify reset-methods, helper methods, and literals) under column "Preproc" (average 14.3 seconds) and for iFixFlakies to produce a patch under column "Patch Gen" (average 99.6 seconds).

We compare against prior work iFixFlakies [43] for generating patches, also shown in Table 2 under the "# Fixed" column. Overall, iFixFlakies could generate a patch for 135 order-dependent tests. In comparison with ODRepair, both ODRepair and iFixFlakies could generate a patch for 117 (64.6%) in-common order-dependent tests. ODRepair exclusively repairs 24 tests (13.3%), and iFixFlakies exclusively repairs 18 tests (9.9%). We inspect more into the tests where neither ODRepair nor iFixFlakies can generate a patch and where the two differ in being able to generate a patch for the test.

Neither technique can generate a patch. There are 22 order-dependent tests from six projects that cannot be repaired by either technique. There are two main reasons for these cases. (1) The polluted static field is not directly accessible outside its defining class (e.g., a private field) and there are no public (or protected) methods that modify the field. For example, for order-dependent test `MainTest.executions` in project apache/geronimo-batchee, the polluted static field `SingletonLocator.MANAGER` is private, and the only `PUTSTATIC` instruction for the field is in a `static` block, i.e., the `<clinit>` of its defining class. We cannot invoke `<clinit>` as it should only be invoked once when the class is first loaded. Therefore, any technique that does not change the code under test cannot repair this order-dependent test. (2) Although there exists a public method that can modify the field, calling that method does not reset the state. For example, for the order-dependent test `RequestBrokerKubernetesServiceTest.testRequestStateHasK8sInfo` in project vmware/admiral, the polluted static field is the `List MockKubernetesApplicationAdapterService.CREATED_DEPLOYMENT_STATES`. There is a method `addDeploymentState` in its defining class that can add more elements to the `List`, but it cannot reset the state back to how it would be in the passing test order. For this test, ODRepair generated four different test methods that invoke this method, but none of them repair the order-dependent test.

Only ODRepair can generate a patch. There are 24 order-dependent tests that can be repaired using ODRepair but not iFixFlakies. The main reason for such cases is that there is no existing test method that iFixFlakies can use as a cleaner. However, ODRepair is not limited to just the developers' test suite, as it can search the code under test as well for the potential reset-methods. From our empirical results, we see that ODRepair can find reset-methods to repair 141/181 tests with known polluted static fields. We inspected the remaining 40 order-dependent tests that ODRepair could not repair, and 14 of them can also be repaired using a reset-method. However, ODRepair could not generate the proper method-call sequence that uses the reset-method correctly. For example, for the order-dependent test `TestTimelineReaderWebServices.testAbout` in project apache/hadoop, the polluted field `TimelineReaderMetrics.instance` is an instance of class `MetricSystem`. While the public method `DefaultMetricsSystem.shutdown()` can reset the static field, no existing test invokes this method, so iFixFlakies cannot find any cleaners and cannot generate any patch. However, ODRepair successfully generates 13 different tests that invoke this method, and all of them can reset the polluted state. This example demonstrates the advantage of ODRepair over iFixFlakies: it does not rely on the developers' test suite and can also be applied to projects with lower-quality test suites or a small number of tests.

Only iFixFlakies can generate a patch. There are 18 order-dependent tests that can be repaired by iFixFlakies but not ODRepair. There are two main reasons for such cases. (1) The patch requires complex method-call sequences that involves multiple classes, even including classes from third-party libraries. For example, the patch that iFixFlakies generates for order-dependent test `NpmTemplateParserTest.should_FindCorrectDataInStats` in project vaadin/flow involves 17 different method calls, including methods from the Mockito mocking library [8]. Note that this method-call sequence is already minimized by iFixFlakies, so there is no shorter method-call sequence that constitute a patch. iFixFlakies can repair this order-dependent test because developers happen to have written a test that contains such a method-call sequence. Generating such a complex method-call sequence is difficult for a random test generator like Randoop. Therefore, when running ODRepair, Randoop did not generate this method-call sequence or a similar method-call sequence. (2) The patch requires complex, hard-to-generate inputs to a method. For example, for the order-dependent test `DirectoryManagerFactoryTest.createDefaultDirectoryManagerPath` in the project wikidata/wikidata-toolkit, the correct patch includes a call to `DirectoryManagerFactory.setDirectoryManagerClass` with the argument `DirectoryManagerImpl.class`. In this case, the argument to the method is a class literal. During the test generation, Randoop could try all the class literals of all the classes that it can access (i.e., all the classes on the classpath), so the search space is extremely large, making it difficult to stumble upon the relevant class literal.

We also sent one pull request per each of the two projects (apache/hadoop and apache/incubator-dubbo) where ODRepair could repair the order-dependent tests but iFixFlakies could not

Table 2: Results of generating tests that can reset polluted static fields.

Project	# OD Tests	# Gen Tests		Time (s)		# Fixed	
		# Test Classes	# Test Methods	Preproc	Patch Gen	ODRepair	iFixFlakies
alibaba/fastjson	1	1.0	325.0	41.7	0.0	0	1
alibaba/Sentinel	3	4.0	701.0	67.0	10.7	1	1
apache/geronimo-batchee	1	2.0	373.0	23.8	0.0	0	0
apache/hadoop	26	2.0	328.0	53.0	49.8	23	1
apache/incubator-dubbo	12	3.0	211.0	21.3	49.4	6	11
apache/jackrabbit-oak	1	1.0	263.0	9.5	0.0	0	0
apache/karaf	1	1.0	360.0	2.3	0.0	0	0
apache/shardingsphere-elasticjob	1	2.0	2.0	4.6	0.0	0	1
apache/skywalking	1	1.0	196.0	2.9	0.0	0	0
ConsenSys/tessera	3	1.0	2.0	0.6	0.0	0	3
elasticjob/elastic-job-lite	4	8.0	1320.0	6.2	154.8	4	4
google/java-monitoring-client-library	1	4.0	240.0	1.1	245.9	1	1
kevinsawicki/http-request	28	2.0	465.0	0.9	68.8	28	28
ktuukkan/marine-api	12	6.0	1330.0	3.2	164.6	12	12
networknt/light-4j	10	6.0	1206.0	2.9	89.8	9	9
openpojo/openpojo	5	1.0	314.0	1.6	33.5	5	5
rest-assured/rest-assured	1	8.0	372.0	8.6	0.0	0	1
spring-projects/spring-boot	8	1.0	298.0	13.3	0.0	0	0
spring-projects/spring-ws	2	1.0	345.0	2.8	0.0	0	2
Thomas-S-B/visualee	46	4.0	892.0	1.2	219.4	46	46
undertow-io/undertow	1	4.0	8.0	289.7	38.3	1	1
vaadin/flow	1	3.0	276.0	8.0	0.0	0	1
vmware/admiral	1	5.0	351.0	21.0	0.0	0	0
wikidata/wikidata-toolkit	2	4.0	320.0	0.6	0.0	0	2
winder/Universal-G-Code-Sender	5	5.0	2500.0	3.4	28.3	5	5
yangfuhai/jboot	4	4.0	467.0	7.9	0.0	0	0
Sum | Mean x2 | Mean x2 | Sum x2	181	3.0	679.0	14.3	99.6	141	135

(we avoid sending more than one pull request per project until first hearing back from its developers, as to avoid bothering them until we know they are interested). So far, the developers of apache/incubator-dubbo accepted our pull request [9].

> ***RQ2:*** *Overall, ODRepair generates patches for 141 order-dependent tests out of 181 for which it could identify the polluted static field. Compared against prior work iFixFlakies, both techniques can generate a patch for 117 (64.6%) in-common order-dependent tests. ODRepair exclusively repairs 24 tests (13.3%), and iFixFlakies exclusively repairs 18 tests (9.9%). We believe ODRepair and iFixFlakies can be used complementary to one another. The envisioned use scenario is that a developer runs iFixFlakies first, and if it cannot generate patch, they can apply ODRepair to generate a method-call sequence that can be used to generate a new patch.*

6 THREATS TO VALIDITY

ODRepair may not apply to all order-dependent tests for all projects. We evaluate on a large, publicly available dataset of order-dependent tests collected from open-source GitHub projects, detected using automated detection tool iDFlakies [4]. We also use iDFlakies to confirm test-order dependencies and build upon iDFlakies.

For our experiment, we allocated 60 seconds for Randoop. The same budget may not be generalizable to all Java projects, but it is sufficient in our case. Given 60 seconds, Randoop took only 28 seconds on average (15 seconds median), terminating early for most (94/141) subjects. In addition, we tried increasing the time budget for the tests that Randoop did not generate a valid cleaner test, and Randoop would on average terminate within 5 minutes (without generating a valid cleaner test).

Our implementation may contain bugs. To mitigate this threat, we build upon existing and mature toolsets, including XStream, XMLUnit, and Randoop. Furthermore, we reviewed code and execution logs to confirm that data processing was done correctly.

7 RELATED WORK

Luo et al. performed the first empirical study on flaky tests by studying bug reports from open-source projects [37]. Eck et al. conducted another empirical study focusing on developers' perspectives on flaky tests [23]. There has also been extensive work on automatically detecting different kinds of flaky tests [21, 27, 32, 42, 48], mitigating the effects of flaky tests on testing tasks [15, 33, 41], or even to automatically repairing flaky tests [22, 43, 47].

Luo et al. found that order-dependent tests are one of the most common kinds of flaky tests in their empirical study [37], and much work has followed that focuses on such flaky tests [27, 28, 32–35, 43–45, 48]. Our work is most similar in nature to iFixFlakies [43], with

the common goal to automatically repair order-dependent tests. iFixFlakies relies on the insight that developers may have written tests, termed cleaners, that reset polluted state for the failing order-dependent test. iFixFlakies searches for these cleaners and minimizes the code within any found cleaner to generate a patch. However, cleaners are not guaranteed to exist, so iFixFlakies may not be able to repair all order-dependent tests. ODRepair first identifies the polluted state and then guides an automatic test-generation tool to specifically generate cleaners that likely reset the polluted state. We then leverage iFixFlakies on these generated cleaners.

Gyori et al. previously proposed PolDet [28] as a means to proactively identify potential polluters, even without a order-dependent test that would fail due to its pollution. PolDet captures the heap-state reachable from static fields before and after each individual test run and compares the captured states. If the states differ, then PolDet reports the test as a potential polluter that leaves behind some polluted state. We use the similar state-capturing logic by serializing the state into an XML representation first and then comparing the XML representations. However, we do not aim to simply detect whether pollution happened, because we are already aware of an existing polluter and order-dependent test. We instead compare the states captured right before the order-dependent test is run in both the passing and failing test orders as to identify potential polluted static fields. We later deserialize the XML representations for these static fields back into objects to be loaded in during a failing test order right before the order-dependent test is run as to check if changing that part of the heap-state to what it was in the passing test order makes the order-dependent test pass.

Wei et al. recently studied non-idempotent-outcome tests that fail when run twice in the same JVM [45]. Their goal is similar to that of Gyori et al. with PolDet, to proactively detect polluters before a potential order-dependent test failure. In fact, these non-idempotent-outcome tests clearly demonstrate a potential problem with polluted state that results in a test failure (since the test itself fails!). We also encounter non-idempotent-outcome tests when we attempt to run the same order-dependent test twice to capture the correct state before the test runs. We encounter relatively few cases (33/327), and we mitigate the issue by only collecting the loaded classes and static fields and then eagerly loading the classes.

Automatic unit test generation [19, 24, 25, 39] aims to generate test inputs and method-call sequences that can execute code under test to find bugs in existing code. Randoop [39] performs random unit test generation. It randomly calls methods from the code under test, where the method-call sequences can result in objects that it can further use as inputs for other methods that it randomly calls. EvoSuite [24–26] leverages evolutionary algorithms to evolve and search for method-call sequences that optimize for a fitness function, such as branch coverage. In this work, we use automatic test generation for its ability to generate method-call sequences as opposed to its goal of achieving high coverage or finding bugs. We use Randoop, and we focus it on generating method-call sequences for potential reset-methods that may reset the polluted state. We use Randoop because prior work showed its random algorithm is both fast and effective at generating method-call sequences, and we do not have any effective fitness function to guide the search.

Jaygarl et al. [29] proposed an object capture-based automated testing technique, which uses dynamically captured object instances from program executions to improve code coverage of existing test-generation tools. Zhang et al. [49] used dynamic analysis to extract information for guiding legal method-call sequence generation and used static analysis to improve the diversity of the generated tests. Babić et al. [16] used dynamic analysis to revolve around a visibly pushdown automaton (VPA) for representing the programs global control flow structure and used static analysis to assist symbolic execution to explore vulnerabilities based on the shortest paths and loop pattern heuristics. Ma et al. [38] proposed GRT, a guided random testing technique that uses static analysis to extract information on program types, data, and dependencies to guide test generation. Kechagia et al. [30] proposed Catcher, a tool that utilizes static exception propagation analysis to guide search-based test generation, effectively pinpointing crash-prone API misuses. Derakhshanfar et al. [20] proposed behavioural model seeding, which synthesizes a behavioural model using the class usage information from the system under test and existing test cases; the model is used to guide search-based test generation. In our work, ODRepair provides guidance to test-generation tool Randoop to focus on previously identified reset-method that are likely to modify and reset polluted state. Our static analysis to mine literals was inspired by previous work [14, 38, 40] with the insight that methods may use these literals, especially those in test code, to reset polluted state.

8 CONCLUSION

Flaky tests mislead developers concerning the results of their regression testing, and order-dependent tests are a common kind of flaky tests. Prior work iFixFlakies can automatically repair order-dependent tests, but it is limited by requiring developers to have already written tests that reset the polluted state for an order-dependent test. We propose ODRepair to automatically repair order-dependent tests by first identifying the exact state that becomes polluted, leading to a failing order-dependent test, and then generating method-call sequences that call reset-methods to reset that polluted state by leveraging a test-generation tool. Our evaluation on 327 order-dependent tests from a public dataset of order-dependent tests shows that we can automatically identify the polluted state for 181 order-dependent tests and generate patches for 141 of them. Compared against iFixFlakies, we can generate a patch for 24 order-dependent tests that iFixFlakies cannot generate a patch for. We believe the two techniques can be complementary; if iFixFlakies does not apply, a developer can then use ODRepair. In the future, we plan to improve ODRepair further through more efficient search of heap-state for pollution as well as generating the reset-methods from scratch instead of searching for existing ones in the codebase.

9 DATA AVAILABILITY

Our evaluation data and scripts are available at https://github.com/UT-SE-Research/ODRepair.

ACKNOWLEDGEMENTS

We would like to acknowledge NSF Grant No. CCF-1718903 for supporting Chenguang Zhu and Wenxi Wang.

REFERENCES

[1] 2012. JUnit and Java 7. http://intellijava.blogspot.com/2012/05/junit-and-java-7.html.
[2] 2013. Maintaining the order of JUnit3 tests with JDK 1.7. https://coderanch.com/t/600985/engineering/Maintaining-order-JUnit-tests-JDK.
[3] 2016. NonDex. https://github.com/TestingResearchIllinois/NonDex.
[4] 2019. iFixFlakies Framework. https://github.com/idflakies/iDFlakies.
[5] 2020. Apache Commons BCEL. https://commons.apache.org/proper/commons-bcel/.
[6] 2021. IDoFT. http://mir.cs.illinois.edu/flakytests.
[7] 2021. JNDI Overview. https://docs.oracle.com/javase/jndi/tutorial/getStarted/overview/index.html.
[8] 2021. Mockito framework site. https://site.mockito.org.
[9] 2021. Pull Request "Including setup method to clean state between tests". https://github.com/apache/dubbo/pull/9265.
[10] 2021. Randoop: Automatic unit test generation for Java. https://randoop.github.io/randoop/.
[11] 2021. Randoop literals file. https://github.com/randoop/randoop/blob/master/src/systemTest/resources/literalsfile.txt.
[12] 2021. XMLUnit. https://www.xmlunit.org/.
[13] 2021. XStream. https://x-stream.github.io/.
[14] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability* 16, 3 (2006).
[15] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated unit test generation for classes with environment dependencies. In *International Conference on Automated Software Engineering*.
[16] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. 2011. Statically-directed dynamic automated test generation. In *International Symposium on Software Testing and Analysis*.
[17] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *International Conference on Software Engineering*.
[18] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *International Symposium on Foundations of Software Engineering*.
[19] Yanping Chen, Robert L. Probert, and Hasan Ural. 2007. Model-based regression test suite generation using dependence analysis. In *Proceedings of the 3rd international workshop on Advances in model-based testing*.
[20] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie van Deursen. 2020. Search-based crash reproduction using behavioural model seeding. *Software Testing, Verification and Reliability* 30, 3 (2020).
[21] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *International Symposium on Software Testing and Analysis*.
[22] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. FLEX: Fixing flaky tests in machine-learning projects by updating assertion bounds. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.
[23] 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*.
[24] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *International Symposium on Foundations of Software Engineering*.
[25] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012).
[26] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *International Symposium on Software Testing and Analysis*.
[27] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *International Conference on Software Testing, Verification, and Validation*.
[28] 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*.
[29] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. 2010. OCAT: Object capture-based automated testing. In *International Symposium on Software Testing and Analysis*.
[30] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and efficient API misuse detection via exception propagation and search-based testing. In *International Symposium on Software Testing and Analysis*.
[31] Biju Kunjummen. 2013. JUnit test method ordering. http://www.java-allandsundry.com/2013/01/junit-test-method-ordering.html.
[32] 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*.
[33] 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*.
[34] 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*.
[35] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020).
[36] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/html/.
[37] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *International Symposium on Foundations of Software Engineering*.
[38] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-analysis-guided random testing (T). In *International Conference on Automated Software Engineering*.
[39] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *International Conference on Software Engineering*.
[40] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016).
[41] August Shi, Jonathan Bell, , and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *International Symposium on Software Testing and Analysis*.
[42] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *International Conference on Software Testing, Verification, and Validation*.
[43] 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*.
[44] 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*.
[45] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting flaky tests via non-idempotent-outcome tests. In *International Conference on Software Engineering*.
[46] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002).
[47] Peilun Zhang, Yanjie Jiang, Anjiang Wei, Victoria Stodden, Darko Marinov, and August Shi. 2021. Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications. In *International Conference on Software Engineering*.
[48] 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*.
[49] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. 2011. Combined static and dynamic automated test generation. In *International Symposium on Software Testing and Analysis*.