Preempting Flaky Tests via Non-Idempotent-Outcome Tests

Anjiang Wei
Stanford University
anjiang@stanford.edu

Pu Yi
Peking University
lukeyi@pku.edu.cn

Zhengxi Li
University of Illinois
zli89@illinois.edu

Tao Xie
Peking University
taoxie@pku.edu.cn

Darko Marinov
University of Illinois
marinov@illinois.edu

Wing Lam
George Mason University
winglam@gmu.edu

ABSTRACT

Regression testing can greatly help in software development, but it can be seriously undermined by flaky tests, which can both pass and fail, seemingly nondeterministically, on the same code commit. Flaky tests are an emerging topic in both research and industry. Prior work has identified multiple categories of flaky tests, developed techniques for detecting these flaky tests, and analyzed some detected flaky tests.

To proactively detect, i.e., *preempt*, flaky tests, we propose to detect *non-idempotent-outcome (NIO)* tests, a novel category related to flaky tests. In particular, we run each test twice in the same test execution environment, e.g., run each Java test twice in the same Java Virtual Machine. A test is NIO if it passes in the first run but fails in the second. Each NIO test has side effects and "self-pollutes" the state shared among test runs. We perform experiments on both Java and Python open-source projects, detecting 223 NIO Java tests and 138 NIO Python tests. We have inspected all 361 detected tests and opened pull requests that fix 268 tests, with 192 already accepted, only 6 rejected, and the remaining 70 pending.

ACM Reference Format:
Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510170

1 INTRODUCTION

Nondeterministic tests that can pass or fail for the same version of the code under test are known by multiple names. Practitioners and researchers most often call these tests "flaky" [37, 44, 65, 77, 91] but also call them "flappers" [33], "unreliable tests" [40], "brittle assertions" [48], "nondeterministic tests" [35], "erratic tests" [70], and more. In this paper, we use the term *flaky tests*. Flaky tests have been reported as an important problem in academic research (e.g., at least seven papers from 2021 analyze [27, 30, 39, 85] and detect [21, 62, 84] flaky tests) and in both "grey literature" (e.g., blogs from Gradle [94], Salesforce [33], and Thoughtworks [35]) and research papers by various companies (e.g., by Apple [55], Facebook [34, 44], Google [19, 38, 69, 71, 100], Huawei [49], Microsoft [46, 47, 56, 57], and Mozilla [83, 90]).

One well-studied category [23, 36, 40, 48, 58, 65, 76, 88, 99] of flaky tests are *order-dependent (OD)* tests [99], whose outcome depends on the order in which tests are run; OD tests occur in two major situations. First, testing frameworks, such as JUnit, do not mandate the order in which tests are run, and test suites that pass in one order can start failing when run in another order. A notorious example occurred when the Java standard library changed from Java 6 to Java 7: many test suites that used to pass started failing, resulting in many publicly reported complaints [52, 53, 66]. Second, tests can run in different orders due to the use of regression testing techniques [59], such as test prioritization [32, 64, 86], test selection [45, 96, 97], and test parallelization [26, 50, 54, 89].

The terminology on OD tests is somewhat confusing as prior papers [23, 36, 40, 48, 58, 65, 76, 88, 99] used the same term with different meanings or introduced new terms for same/similar concepts. We follow the most recently used terminology [39, 88]. Following Shi et al. [88], we call a test a *victim* for a given test suite (e.g., `t1` in Figure 1) if the test *fails* when run *after* another test, called a *polluter*, in the same test suite (e.g., `t2` in Figure 1) but *passes* when run *before* that other test. The victim fails because the tests share some state (`x` in Figure 1), and the polluter modifies (i.e., "pollutes" [40, 88]) the shared state. Huo and Clause [48] called the test assertions that depend on the shared state "brittle assertions". Each victim has at least one brittle assertion, but not all tests with a brittle assertion are victims (e.g., `t3` in Figure 1 has a brittle assertion, but no test pollutes `z`). We call a test a *latent-victim* if it has a brittle assertion but may or may not be a victim in the current test suite.

Note that a polluter is defined with respect to a given test suite, where the test suite has a victim. Gyori et al. [40] used the term "polluter" to refer to any test that changes some shared state even if it has no victim in the current test suite (e.g., `t4` in Figure 1 modifies `y` but no test fails because of that). To avoid confusion, we use *latent-polluter* to refer to a test that modifies the shared state but may or may not have a victim in the current test suite. Following Musuvathi et al. [74], a latent-polluter can be also called a "non-idempotent-state test", because the test definitely modifies the state, but running the test twice may or may not have a different behavior.

To reduce the risk that flaky tests fail at inopportune times, practitioners [44, 90] and researchers [40, 48, 61] have advocated for proactively detecting potential flaky tests, i.e., *preempting* them from becoming flaky. For example, to preempt OD-related tests, Huo and Clause [48] proposed using dynamic taint analysis to detect latent-victims, and Gyori et al. [40] proposed monitoring

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

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510170

```
// shared variables x, y, z, w are initialized to 0
void t1() { assert x == 0; } // victim
void t2() { x = 1; } // polluter
void t3() { assert z == 0; } // latent-victim
void t4() { y = 1; } // latent-polluter
void t5() { assert w == 0; w = 1; } // NIO
```

Figure 1: Example test suite containing different kinds of victims and polluters, including an NIO test.

the shared heap state and file system to detect latent-polluters. However, while detecting latent-victims and latent-polluters, the key is to balance detecting as many tests as possible with detecting tests that are worth fixing. For example, Gyori et al. [40, Figure 4] reported many false positives: they automatically found 575 latent-polluters, manually inspected all and filtered out 381 tests that cannot reasonably become polluters (e.g., modify state that cannot be observed via any public API but only through reflection), and did not fix any of the remaining 194 tests.

To detect latent-victims and latent-polluters that are worth fixing, we propose to focus on *non-idempotent-outcome (NIO)* tests, which are related to OD tests [99] and similar to "unrepeatable tests" [70]. A test is an NIO test if the test outcome (pass or fail) changes after repeated test runs, due to the changes of the state shared among runs of the NIO test (e.g., `t5` in Figure 1). For a test t to be an NIO test, t must write and read some shared state (`w` in Figure 1). Detecting t can be helpful because an OD test can emerge when t is run together with *another* test (from the current or future versions of the t's test suite), where that other test writes or reads a part of the state shared with t. If the other test writes to the same shared state, then t can become flaky/victim; if the other test reads the shared state, then t can become a polluter with the other test as flaky/victim. Figure 2 shows the relationship of multiple categories of OD-related tests.

NIO tests are important to detect because they may be more worthy fixing than other latent-victims and latent-polluters, considering that NIO tests are *both* latent-victims *and* latent-polluters at the same time. In contrast to many false positives (66%) that Gyori et al. [40] reported for latent-polluters, with no pull requests opened, we find that developers are receptive to our NIO test fixes. While debugging (NIO tests in) unfamiliar projects can be time-consuming, we have fixed many NIO tests that we have detected in open-source projects. Specifically, we have opened pull requests for 268 tests. Developers have accepted fixes for 192 of the tests with only 6 rejected, and the remaining 70 pending. (One project is an outlier as we have opened fixes for 120 NIO tests, and the developers have accepted our fixes for all 120 tests.)

To detect NIO tests, we use a simple idea: run each test twice in the same test execution environment to check whether the test passes in the first run but fails in the second run. As prior work on flaky tests has been mostly for Java [77] and recently for Python [39], we perform our evaluation on open-source Java and Python projects; the general principle easily extends to other languages. We run each test twice in the same execution environment, a Java Virtual Machine (JVM) or a Python interpreter for Java or Python tests, respectively. A test is categorized as NIO if the test outcome changes *deterministically* from "pass" to "fail". More formally, we require that there *exists* a test order $\langle \ldots, t, t, \ldots \rangle$ such that running the test t results in the outcomes "pass" and "fail".

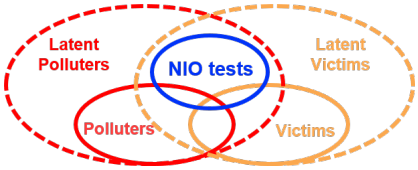

Figure 2: Relationship of polluters, victims, latent-polluters, latent-victims, and NIO tests.

To detect NIO tests for Java projects, we evaluate three modes that run tests in isolation or together. One mode reruns only *one test method* in the same execution environment, another mode reruns test methods from only *one class*, and the third mode reruns all test methods from the *entire test suite*. We modify the iDFlakies tool [58] to support running the same test twice *without* changing the test code. By default, JUnit does *not* run the same test *method* twice. Hence, prior tools for flaky tests (including DTDetector [99], ElectricTest [24], iDFlakies [58], iFixFlakies [88], and PraDet [36]) do not contain this feature.

Our evaluation on 127 test suites from open-source Java projects detects 223 NIO tests in 34 of the test suites. The three modes detect only slightly different tests, but rerunning all tests from the entire test suite runs the fastest. We apply that mode on 1006 Python projects and detect 138 NIO tests in 90 projects. Of the 361 detected NIO tests, 42 are not NIO in the latest version (i.e., already fixed or deleted). We have inspected in detail all remaining 319 tests and opened pull requests for 84% (268/319) of the tests. Each test requires building the project, running the test, and debugging the shared state—usually requiring at least an hour from multiple authors. Section 4.4 discusses our experience of fixing NIO tests and Section 5 presents multiple real cases of our fixes.

In this paper, we make the following main contributions:

- **NIO Tests.** We define NIO tests as tests that deterministically change outcome from "pass" to "fail" when run twice in the same execution environment; NIO tests are in the intersection of latent-polluters and latent-victims.
- **Effective Detection.** We propose three modes to detect NIO tests by repeatedly running tests, either in isolation or together, in the same execution environment.
- **Empirical Evaluation.** We evaluate all modes on 127 Java test suites and detect 223 NIO tests. We also evaluate the most effective mode on 1006 Python projects and detect additional 138 NIO tests.
- **Real Cases.** We present multiple real cases of NIO tests to illustrate the specific causes that make them NIO and discuss our experience in fixing NIO tests.
- **Well-Accepted Fixes.** We have opened pull requests that fix 268 tests, with 192 of them accepted, only 6 rejected, and the remaining 70 pending.

Our dataset and scripts are publicly available [18].

2 BACKGROUND, MODES, AND EXAMPLES

NIO tests are related to OD tests, which can pass or fail based on the order of the tests in the test suite. Section 1 has introduced the most common kinds of OD-related tests. We introduce one less common kind here. OD tests deterministically fail when run in some pre-states. (A test that fails for all pre-states is a broken, not

flaky, test.) Beyond victims and polluters (defined in Section 1), Shi et al. [88] defined a *brittle* as a test that fails even when run in the starting execution environment state (i.e., a brittle b fails in the order $\langle b \rangle$) and has at least one test s in the test suite such that s sets the state for the brittle to pass (i.e., b passes in the order $\langle s, b \rangle$). In contrast, victims pass in the starting state (i.e., a victim v passes in the order $\langle v \rangle$) but fail after a polluter p (i.e., v fails in the order $\langle p, v \rangle$); victims are much more common than brittles. In all prior work [77], various tests in the same order were all *different* tests because each test was always run only once in a test order.

In contrast, NIO tests stem from repeating the *same* test. In the simplest case, only one test t is run twice, i.e., the order is $\langle t, t \rangle$. If the first run fails, the test is a brittle. However, if the first run passes and the second run fails, the test is NIO. Note that each NIO test is, by definition, *both* a latent-victim and a latent-polluter—it "self-pollutes" the state on which it depends. Moreover, some NIO tests may be victims or polluters in their test suite, but our evaluation for Java tests shows that most of the NIO tests, 87.4% (195/223), are *neither* polluters *nor* victims in their test suite.

2.1 Three Modes for Detecting NIO Tests

We evaluate three *modes* to detect NIO tests. Each mode repeatedly runs (1) just a particular test method from a test class (*isolated-method*), (2) all the test methods from a test class (*isolated-class*), or (3) all the test methods from a test suite (*entire-suite*). For example, consider a test suite with two test classes, C and D, and three tests[1] $C.t$, $C.u$, and $D.v$ with their explicitly listed classes: isolated-method repeatedly runs each test in its own VM[2] twice, e.g., $\langle C.t, C.t \rangle$; isolated-class runs all tests from each test class in one VM, e.g., $\langle C.t, C.t, C.u, C.u \rangle$; entire-suite runs all tests from the test suite in one VM, e.g., $\langle C.t, C.t, C.u, C.u, D.v, D.v \rangle$.

Different modes for detecting NIO tests could have trade-offs in terms of the tests that they detect or miss, and how fast they run. Compared to isolated-class and entire-suite, isolated-method would miss detecting a test t as NIO if another test, t', sets/pollutes the state so that running $\langle t', t, t \rangle$ makes the second t fail, while running just $\langle t, t \rangle$ makes both runs pass. In our experiments (Section 4), we find that isolated-method does *not* miss detecting any NIO test that isolated-class and entire-suite detect. However, isolated-method needs to create a new VM for every test, so this mode can run substantially slower than the other modes. Some prior projects [22, 75] did compare running Java tests isolated in JVM vs. all together in one JVM, but those projects were *not* repeating tests trying to detect NIO tests.

On the other hand, isolated-class and entire-suite would also miss detecting a test t as NIO if some other test, t'', sets/cleans the state so that running $\langle t'', t, t \rangle$ makes both t pass or both fail, while running just $\langle t, t \rangle$ makes the first t pass and the second t fail. In our experiments, we find that entire-suite misses 13 (of 223) NIO tests detected by isolated-method. Of the 13 tests, 10 are not detected because they fail in both runs, which should prompt developers to inspect them already. The remaining 3 tests are not detected because they pass in both runs. Section 4.2 presents more details.

[1]We use the term "test" to refer to a test method, following the JUnit terminology and how the test code is organized into test classes that contain test methods.
[2]We use the term "VM" to refer to an execution environment, such as a Java Virtual Machine or a Python interpreter.

```
1  static AtomicInteger counter = new AtomicInteger();
2  class Command...Exception {
3    public void execute(...) {
4      counter.incrementAndGet();
5      throw new ActivitiException("");
6    }
7  }
8  @Test
9  public void testRetryInterceptor() {
10   ... // setup retryInterceptor and processEngine
11   try {
12     processEngine.getManagementService().
         executeCommand(new Command...Exception());
13     Assert.fail("...");
14   } catch (ActivitiException e) {
15     ... // assert what number of retries failed
16   }
17   Assert.assertEquals(retryInterceptor.
       getNumOfRetries() + 1, counter.get());
18 }
19 @After
20 public void shutdownProcessEngine() {
21   processEngine.close();
22 + counter.set(0);
23 }
```

Figure 3: Our fix for an example NIO test detected by all modes in `activiti` [20].

To illustrate the differences and similarities, we next show two examples of real NIO tests that we detect in open-source Java projects: (1) a test detected by all three modes and (2) a test detected by only isolated-method. These examples come from popular Java projects, showing that even mature, well-tested projects can have NIO tests. Section 5 discusses more examples of NIO tests.

2.1.1 NIO Test Detected by All Modes. Figure 3 shows an NIO test detected by all three modes. This test is from the project `Activiti` [20], which is a light-weight workflow and Business Process Management platform.

The `testRetryInterceptor` test starts by setting up a `retryInterceptor`, which is used to set up a `processEngine` (Line 10). The test then runs a command with the `processEngine` (Line 12) before asserting that some number of retries are performed (Lines 15 and 17). By default, the `retryInterceptor` is set to retry a command three times if it fails. Specifically, the command object used by the test is `Command...Exception`, which simply increments the *shared* `counter` before throwing an exception (Lines 4–5).

This test is NIO because of the shared `counter` value. The test asserts (Line 17) that the number of retries recorded by `retryInterceptor` is the same as the value of the `counter`. In the first test run, the `retryInterceptor` ensures that the command (Line 12) is retried three times (+1 for the first try), and the test passes as the `execute` method (Line 3) will have run four times, setting `counter` to four. However, in the second test run in the same JVM, the `retryInterceptor` is reinitialized and starts with zero retry, while the counter is *not* reinitialized and will already be four from the first run of this test. Indeed, the exception for the test failures in the second run is that `retryInterceptor.getNumOfRetries()+1` is four, while `counter.get()` is eight from the two runs of the test.

We prepare a fix by resetting the `counter` to 0 in the `@After` method of the class (Line 22). Our pull request [1] for this fix has been accepted by the developers. (An alternative fix would have

```
1  @Test
2  public void testSize() throws InterruptedException {
3    ... // create an object in InternalThreadLocal
4    Assert.assertTrue("size method is wrong!",
       InternalThreadLocal.size() == 1);
5    ... // create an object in InternalThreadLocal
6    Assert.assertTrue("size method is wrong!",
       InternalThreadLocal.size() == 2);
7  + InternalThreadLocal.removeAll();
8  }
9  @Test
10 public void testSetAndGet() {
11   ... // setup testVal and internalThreadLocal
12   ... // create an object in InternalThreadLocal
13   Assert.assertTrue("set is not equals get", Objects.
       equals(testVal, internalThreadLocal.get()));
14 }
```

Figure 4: Our fix for an example NIO test (`testSize`) detected by only the isolated-method mode in `dubbo` [28].

added `counter = new AtomicInteger();` at the start of the test.) The `testRetryInterceptor` test is *neither* a victim nor a polluter.

2.1.2 NIO Test Detected by One Mode. Figure 4 shows a test detected as NIO in only the isolated-method mode. This test is from the project `Dubbo` [28], which is a high-performance remote procedure call framework.

The `testSize` test checks whether the `size()` method defined in `InternalThreadLocal` correctly returns the total number of local `InternalThreadLocal` objects bound to the current thread. The test first creates a thread local object and asserts that `size()` returns 1 (Line 4). The test then creates another thread local object and asserts that `size()` returns 2 (Line 6).

This test first passes and then fails when run twice in the isolated-method mode, because the test does not remove the two created objects between the two test runs. Specifically, during the second run of this test in the same JVM, three objects are bound to the current thread (two objects from the previous run of the test and one new object from the current run of the test), while during the first run only one object is bound. Therefore, in the second run, the test fails the first assertion (Line 4).

In contrast, `testSize` fails in both runs of the isolated-class and entire-suite modes and is, thus, not reported as NIO. In these two modes, `testSize` runs after `testSetAndGet` (Line 10), which also creates a thread local object (Line 12) and does not remove it. Essentially, `testSize` is not only NIO when run in the isolated-method mode but also a victim with `testSetAndGet` being the polluter.

Although `testSize` is not reported as NIO in the isolated-class and entire-suite modes, the test does fail in both modes, and developers would ideally fix all failing and NIO tests. Section 4.2 describes an interesting case where a test passes in both isolated-class and entire-suite modes, but is detected as NIO in isolated-method.

Our fix simply adds `InternalThreadLocal.removeAll();` at the end of the test (Line 7). Our pull request [2] for this fix has been accepted by the developers.

3 RESEARCH QUESTIONS AND SETUP

To improve the understanding of NIO tests, we investigate the following research questions (RQs):

RQ1: How prevalent are NIO tests in projects with flaky tests?
RQ2: How do different running modes affect NIO test detection?
RQ3: How do the runtimes of detection modes differ?
RQ4: How do developers respond to proposed fixes for NIO tests?
RQ5: How do NIO tests compare to other OD-related tests?

We empirically address these RQs on Java and Python projects. We first describe how we select the projects for our evaluation. We use Java projects for all five RQs but Python projects for only RQ1 and RQs4-5. We do not use Python projects for RQs2-3 to reduce the machine costs; our evaluation on Java projects finds the entire-suite mode to be the best trade-off. We next describe how we use/modify some testing tools for our evaluation. We finally describe how we confirm the detected NIO tests.

3.1 Projects

For Java, we use the projects from our recent studies [61, 92] on flaky tests. The studies found at least one flaky test in 55 open-source Java projects obtained from GitHub. For each project, we use the same Git commit as the studies. We use the same projects and commits because the studies detected victims and polluters in the specific project commits, thereby allowing us to compare NIO tests that we detect to previously detected tests (Section 4.5). However, these project commits are somewhat older, so some detected NIO tests may be already fixed or deleted in the latest project version.

All selected projects use the Maven build system [67], so each test suite in our study, as in prior studies [61, 92], is a Maven module. Maven-based Java projects are organized in a set of modules that can each have their own code under test and a test suite. Our study uses 127 of the 130 modules from the recent studies. We omit 3 modules because we have trouble running their tests. All projects use JUnit, the most popular testing framework for Java. Most Maven plugins run the same operation on each module in the project. Surefire [68] is the default Maven plugin for running tests; executing `mvn test` at the top level of a Maven-based project runs Surefire for each module in the project. Surefire then finds all test classes in the module and passes them to JUnit, which for each test class finds all test methods and runs them *without* repetition. As described in Section 3.2, we adapt the iDFlakies tool [58] to enable running various modes *with* test repetition.

For Python, we use the dataset from a recent study by Gruber et al. [39]. The dataset has 1006 projects, each of which is reported to have at least one flaky test. To detect NIO tests, we run each project on the commit in which the dataset reports at least one flaky test. Building Python projects can be difficult due to dependency-related errors [73]. We use FlaPy [39], the infrastructure released with the dataset for building the projects and running the tests.

3.2 Tools for Detecting NIO Tests

For Java, we modify a research testing tool, i.e., iDFlakies [58], to enable repeating tests in one execution without any changes to the test code. We choose iDFlakies because we are familiar with the tool, and it works for the Java projects that we select [58]. Our extension is relatively simple as iDFlakies already treats the input as a *list* of tests and allows repetition. However, the output that iDFlakies produced for multiple test runs would overwrite the results of earlier runs with the later runs because the test name was

used as a key to the run result. To support running test(s) multiple times in one JVM, we change how iDFlakies reports results and have added our change to iDFlakies.

For Python, the dataset that we use comes with an infrastructure to run tests using `pytest` [80], the most popular testing framework in Python. To run each test multiple times in one Python interpreter, we use a `pytest` plugin called `pytest-repeat` [81]. We invoke `pytest-repeat` with `--count=2`, with no modifications to the test code. From the output, we exclude parameterized unit tests (whose name includes a square bracket) and *UnitTest style* tests (whose class extends `unittest.TestCase`) because `pytest-repeat` cannot run them correctly [81]. If `pytest-repeat` did not have these limitations, we could have detected even more NIO tests in the projects that we evaluate. To ensure that our experiments finish in a reasonable amount of time, we set a 10-second timeout for each test. Only 486 of 34,446 tests time out.

We run setups and teardowns during NIO detection for both Java and Python. We do not clean the disk state between runs because the cleaning would be too expensive.

3.3 Confirming Detected NIO Tests

Running a test twice and observing the first run pass and the second run fail does *not* guarantee that the test is NIO, because some tests are non-deterministic [60], i.e., they can pass and fail in repeated runs even for the same test order. Moreover, considering that iDFlakies or `pytest-repeat` may have bugs, we want to check whether all NIO tests detected by these two specialized tools can also be detected using the tools that developers more commonly use to run tests, such as Surefire and JUnit for Java and `pytest` for Python.

For both Java and Python, we confirm whether the detected tests are NIO by adding a copy of the test and running together the original and the copy, thus effectively running the test twice. For example, for a Java test `@Test public void testOriginal() {/*body*/}`, we can add `@Test public void copy() {testOriginal();}`. Running a test twice by adding a copy is fairly reliable and robust.[3] Using a copy to confirm the detected NIO tests, we remove the tests that are brittles [88] or non-deterministic. Beyond confirming every test from Table 2 by running a copy, we manually inspect the code to identify the shared state. Our manual inspection and running of the original and copy of the NIO tests confirm that *every* test in our evaluation is indeed NIO (no false positives).

Another benefit of adding a test copy is to show that one need not use specialized tools, such as iDFlakies and `pytest-repeat`, to detect the NIO tests. For Java, we also consider confirming the NIO tests with the `@Rule` and `@ClassRule` annotation from JUnit. We find these annotations to be worse than adding a copy, because they do not work for some older versions of JUnit or when the test class uses some specialized test runners.

4 RESULTS

4.1 RQ1: Prevalence of NIO Tests

For Java, we detect a total of 223 NIO tests in 34 modules. We apply our three detection modes to 127 modules and detect at least one NIO test in 26% of modules. These 127 modules have a total of 40,019 tests, so NIO tests are over 0.5% of all tests in these modules. Module M20, with 122 NIO tests, is an outlier. Even ignoring this module, we still find the ratio of NIO tests to be non-negligible, over 0.2% (101 out of 39,536) of all tests, in the remaining 126 modules.

[3]Running a test copy, however, can mask the confirmation of NIO tests when the test name affects the test behavior, e.g., the test name matches some external resource in the file system. We do not observe such a problem in our additions of copy.

```
public class SearchQueryTest extends ... {
  @Override public void setUp() {
    super.setUp();
    createUser("Bob", ...);
    createUser("Barbara", ...);
    createUser("Anton", ...);
    createUser("Robert", ...);
    createUser("John", ...);
    Session session = getSession();
    session.flush();
    session.getTransaction().commit();
    session.beginTransaction();
  }
  @Test
  public void no_where() {
    assertEquals(5, query().fetch().size());
  }
}
```

Figure 5: NIO test (`no_where`) in `querydsl` [82].

For Python, we detect a total of 138 NIO tests in 90 projects. We apply the entire-suite mode to 1006 projects and detect at least one NIO test in about 9% of projects. This mode runs a total of 34,446 tests in these 1006 projects, so NIO tests are over 0.4% of all tests.

Table 1 shows the statistics of the NIO tests detected in 34 Java modules. Due to limited space, we omit a detailed breakdown for 90 Python projects. For each Java module in which our experiments detect at least one NIO test, we tabulate the GitHub slug (username/project) and module name, the number of NIO tests that are detected for isolated-method (IM), isolated-class (IC), and entire-suite (ES) modes, the time to run our experiments for the three different modes, and the time ratios. In Section 4.5, we compare NIO tests to other kinds of OD-related tests in these Java modules and Python projects.

A1: NIO tests are currently prevalent enough that every project should run NIO detection at least once.

4.2 RQ2: NIO Tests Detected in Different Modes

Section 2 has introduced our three modes to detect NIO tests. In our experiments, the majority (210) of NIO tests are detected by all three modes. All tests detected by entire-suite are detected by isolated-class, and all tests detected by isolated-class are detected by isolated-method. In contrast, isolated-method detects 11 and 13 tests that are not detected by isolated-class and entire-suite, respectively. Our inspection finds that 8 (resp. 10) tests are not detected by isolated-class (resp. entire-suite), because they have polluters in the test class (resp. test suite). These polluters run before the NIO tests, making them fail twice, in the isolated-class or entire-suite mode. Section 2.1.2 presents an example of one of these tests where a polluter makes the example test fail in both runs of the isolated-class and entire-suite modes.

The remaining 3 out of 13 tests, which pass in the first run and fail in the second run in isolated-method, interestingly *pass* in both runs in the isolated-class and entire-suite modes. All three tests are from the module M22 (`querydsl-hibernate-search`), and

Table 1: Statistics about the NIO tests detected and the time taken in various modes (IM: isolated-method; IC: isolated-class; ES: entire-suite). Overhead shows the ratio of runtime for various modes.

		# NIO Tests			Time to Run [s]			Overhead	
ID	**Project User/Name - Module**	**IM**	**IC**	**ES**	**IM**	**IC**	**ES**	**IM/IC**	**IM/ES**
M1	activiti/activiti - activiti-engine	2	2	2	15729	4694	604	3.4	26.0
M2	activiti/activiti - activiti-spring-boot-starter	7	6	6	726	369	229	2.0	3.2
M3	apache/hadoop - hadoop-hdfs-httpfs	3	3	2	2981	904	596	3.3	5.0
M4	apache/hadoop - hadoop-hdfs-nfs	1	0	0	1290	880	749	1.5	1.7
M5	apache/hadoop - hadoop-mapreduce-client-app	3	3	3	6087	2258	1400	2.7	4.3
M6	apache/hadoop - hadoop-mapreduce-client-core	3	1	1	3689	1617	841	2.3	4.4
M7	apache/hadoop - hadoop-mapreduce-client-jobclient	5	5	5	29354	19702	7200	1.5	4.1
M8	apache/hbase - hbase-server	13	13	13	180778	55207	10320	3.3	17.5
M9	apache/incubator-dubbo[4] - dubbo-cluster	3	3	3	927	450	221	2.1	4.2
M10	apache/incubator-dubbo - dubbo-common	6	5	5	3341	751	292	4.4	11.4
M11	apache/incubator-dubbo - dubbo-config-api	2	2	1	2184	411	215	5.3	10.2
M12	apache/incubator-dubbo - dubbo-monitor-default	1	1	1	258	249	255	1.0	1.0
M13	apache/incubator-dubbo - dubbo-remoting-netty	1	1	1	374	356	310	1.1	1.2
M14	apache/incubator-dubbo - dubbo-rpc-api	4	4	4	577	382	236	1.5	2.4
M15	apache/incubator-dubbo - dubbo-rpc-rest	2	2	2	315	256	216	1.2	1.5
M16	eclipse-ee4j/tyrus - non-deployable	1	1	1	433	201	112	2.2	3.9
M17	elasticjob/elastic-job-lite - elastic-job-lite-core	4	4	4	2890	964	188	3.0	15.4
M18	looly/hutool - hutool-core	1	1	1	2967	651	85	4.6	34.9
M19	orbit/orbit - actor-tests	1	1	1	8518	752	278	11.3	30.6
M20	pholser/junit-quickcheck - core	122	122	122	2687	671	101	4.0	26.6
M21	pholser/junit-quickcheck - generators	4	4	4	6255	2294	156	2.7	40.1
M22	querydsl/querydsl - querydsl-hibernate-search	3	0	0	386	372	316	1.0	1.2
M23	spring-projects/spring-boot - spring-boot	2	2	2	148990	12299	384	12.1	388.0
M24	spring-projects/spring-boot - spring-boot-actuator	12	11	11	21821	5416	368	4.0	59.3
M25	spring-projects/spring-boot - spring-boot-actuator-autoconfigure	2	2	2	21866	9357	429	2.3	51.0
M26	spring-projects/spring-boot - spring-boot-test	1	1	1	23505	5623	242	4.2	97.1
M27	spring-projects/spring-boot - spring-boot-test-autoconfigure	4	4	4	6468	4358	538	1.5	12.0
M28	spring-projects/spring-ws - spring-ws-core	1	1	1	4529	1247	223	3.6	20.3
M29	undertow-io/undertow - servlet	1	0	0	2289	1133	297	2.0	7.7
M30	vmware/admiral - kubernetes	1	1	1	684	308	182	2.2	3.8
M31	vmware/admiral - common	4	4	4	668	285	142	2.3	4.7
M32	vmware/admiral - request	1	0	0	3995	1833	995	2.2	4.0
M33	wildfly/wildfly - server-integration	1	1	1	690	457	271	1.5	2.5
M34	zalando/riptide - riptide-spring-boot-starter	1	1	1	494	321	134	1.5	3.7
Sum × 3 | **Arith. Mean** × 3 | **Geo. Mean** × 2		**223**	**212**	**210**	**14963**	**4030**	**857**	**2.5**	**8.4**

all three tests have the same root cause. One of the three tests (`SearchQueryTest.no_where`) is shown in Figure 5.

Unlike the two examples described in Section 2, the polluted state that causes `no_where` to fail is not on the heap but in a database stored in the file system. Specifically, the test adds five entries to a database and then checks whether the database contains five entries (Line 16). The entries added to the database are saved to the file system only after all of the tests finish and the JVM exits. Therefore, in the isolated-class and entire-suite modes, even when this test is run multiple times, and the `setUp()` method is run multiple times, all of the runs use a new (empty) database and all of the runs pass. However, we find `no_where` to pass in the first run and fail in the second run in the isolated-method mode because we already run (twice) another test in the `SearchQueryTest` class, saving the database to the file system before `no_where` runs. The first run of `no_where` in the isolated-method mode passes even with a polluted database because the database is loaded asynchronously, and if the database is not ready by the first run, then it simply uses a new database from memory. On the other hand, the second run typically uses the polluted database from the file system and consequently fails. Section 5.1 describes more details about this test.

A2: All three modes detect similar tests, but isolated-method detects slightly more than isolated-class, which detects slightly more than entire-suite.

4.3 RQ3: Runtime of Different Modes

Section 2 has discussed how the three different modes can vary in the runtime due to the number of JVMs that the modes need to run. In all three modes, the total number of *test runs* is exactly the same, but the number of *JVM runs* differs greatly. Each JVM run has to start a JVM and load the required classes, taking nontrivial time. The number of JVM runs needed is the same as the number

[4] `apache/incubator-dubbo` now redirects to `apache/dubbo`, but we keep the old name to be consistent with prior work.

of tests, the number of test classes, and one in the isolated-method, isolated-class, and entire-suite modes, respectively.

Table 1 presents the time to run the three detection modes. As expected, the isolated-method mode is the slowest among the three modes. Specifically, the (geo. mean) overhead of the isolated-method mode is 8.4x and 2.5x over the entire-suite and isolated-class modes, respectively. These numbers confirm that the overhead for each JVM run is nontrivial [22, 75]. Our results also show that the M23 module has a much higher overhead for isolated-method/entire-suite than other modules. We find that the higher overhead is because M23 has the highest number of tests (2,108) of all the modules, and consequently, isolated-method runs 2,108 JVMs, while entire-suite runs only 1 JVM. As the entire-suite mode runs substantially faster than the other two modes and yet misses detecting only 5.8% (13/223) of NIO tests, we recommend that developers regularly run the entire-suite mode and only rarely run the isolated-method mode to detect NIO tests that the entire-suite mode may miss. Following our own recommendation, we run only the entire-suite mode for Python projects but confirm the detected Python NIO tests by running each test twice in isolation.

A3: The most cost-beneficial mode is entire-suite; we suggest running entire-suite periodically and isolated-method for only newly-added or directly-modified tests [61].

4.4 RQ4: Experience with Fixing NIO Tests

We fix and open pull requests for 268 NIO tests. 192 of them are accepted, only 6 rejected, and the remaining 70 pending. Table 2 shows the statistics about our pull requests. The rows "Java" and "Python" show the sum for all Java modules and Python projects, respectively. To illustrate diversity, we show details for each Java module. Due to space limit, we show only the sum for Python.

Our experiments use an older version of projects (to compare with victims and polluters; Section 4.5), but tests are worth fixing only in the latest version. The table marks as "N/A" 42 tests that are not NIO in the latest version, i.e., fixed, deleted, ignored (e.g., annotated with `@Ignore` in Java) or archived.

We inspect *all* 361 NIO test that we detect, even "N/A". For each NIO test, we inspect for at least one hour before giving up and proceeding to the next test. Each NIO test that we do not fix in our first iteration is reinspected later. In the end, we do not fix 51 tests (17 Java and 34 Python) for three reasons: (1) we cannot localize the pollution (for 3 Java and 16 Python tests); (2) we localize the pollution but it is difficult to clean (for 13 Java and 13 Python tests); and (3) we do not fix tests that are specified to run in specific orders (for 1 Java and 5 Python tests, e.g., annotated with `@TestMethodOrder` in Java) because developers are likely already aware of the state pollution, thus unlikely to want to clean the state pollution that makes other tests pass (in other words, the NIO test sets state for some brittle test [88]).

An example test that we do not fix because we cannot localize the state pollution is `AuthUtilsTest.testValidateSessionData` from M31. The test tries to create a new user each time, checks several functionalities of the session, and then clears the session. In the second run, the test fails the assertion `assertEquals(authCtxUser, getOp.getAuthorizationContext())`, because the call `getAuthorizationContext()` returns `null` instead of the expected object. We find that `null` is returned because at the end of the first run, the regular user "logs out" via `AuthUtils.cleanupSessionData(getOp)`. However, after careful inspection, we find that seemingly all the variables involved in the test code are newly initialized, and therefore, we cannot easily identify the exact global variable that is shared between the two runs. We envision that future research can apply program analysis techniques to help developers localize the shared state for NIO tests.

An example test that we do not fix because we cannot clean the pollution is `WebMvcMetricsFilterTests.regexBasedRequestMapping` from M24. In the second run, the test fails the assertion `assertThat([...].timer().count()).isEqualTo(1L)` because `count()` returns `2L` instead of the expected `1L`. It is obvious that we need to reset the timer (or the object that stores the timer) to clean the

Table 2: Statistics about our pull requests (PRs) for NIO tests; "N/A" marks tests not available in the latest project version.

	# NIO Tests		# NIO Tests in Our PRs		
ID	detected	N/A	opened	accepted	rejected
M1	2	0	2	2	0
M2	7	7	N/A	0	0
M3	3	0	3	0	1
M4	1	0	1	1	0
M5	3	0	3	1	0
M6	3	0	3	1	0
M7	5	0	5	1	0
M8	13	3	7	1	0
M9	3	0	2	2	0
M10	6	0	6	6	0
M11	2	1	1	1	0
M12	1	0	0	0	0
M13	1	1	N/A	0	0
M14	4	0	4	4	0
M15	2	2	N/A	0	0
M16	1	0	1	1	0
M17	4	0	4	4	0
M18	1	1	N/A	0	0
M19	1	1	N/A	0	0
M20	122	2	120	120	0
M21	4	0	4	4	0
M22	3	0	3	3	0
M23	2	1	1	0	0
M24	12	0	2	0	0
M25	2	0	2	0	0
M26	1	0	1	0	1
M27	4	1	3	0	3
M28	1	0	1	0	0
M29	1	0	1	0	0
M30	1	0	1	0	0
M31	4	0	3	0	0
M32	1	0	1	0	0
M33	1	0	0	0	0
M34	1	0	1	1	0
Java	223	20	186	153	5
Python	138	22	82	39	1
Total	**361**	**42**	**268**	**192**	**6**

pollution. However, after extensive code search, we cannot find a reasonable cleaning method. Moreover, the receiver object for `timer()` is created by a class imported from a library dependency, preventing us from easily implementing the functionality to reset the timer's state in the module M24 itself.

Overall, we fix 268 (of the available 319) tests by revising/adding code to clean the shared state of the NIO tests, thereby making them idempotent. The number of opened pull requests (PRs) is "N/A" if all NIO tests detected in a module are no longer applicable in the latest version. Our fixes cover at least one test in 82 out of 109 components—27/29 Java modules and 55/80 Python projects—that have at least one NIO test remaining in the latest version.

Based on the acceptance rate and developers' comments, most of our fixes have been appreciated by developers. Ignoring the outlier M20, of all the accepted tests, 40% for Java (19% for Python) were accepted without comments, while 60% (81%) were accepted with developer compliments (e.g., "nice catch", "thanks", "lgtm"); only 2 for Java (5 for Python) tests had some discussions with developers (e.g., asking us to modify '@Before'). Out of the 268 fixed tests, only 6 (5 for Java and 1 for Python) had our PRs rejected. For the rejected tests in M3 and M26, the developers believed that the polluted state would affect only the test itself and not any other test, and thus claimed that no fixes were needed [3, 4]. For the three rejected tests in M27, the developers believed that our fix was potentially "masking" the problem [5]. For the rejected Python test, the developer appreciated our finding by confirming that the NIO test is "certainly a valid issue", but rejected our PR because the developer wanted to completely refactor the code instead [6].

The module M20 is a big outlier with 120 tests. We point out that the fixes for these 120 tests do have some diversity, e.g., modifying eight different sets of data structures. Moreover, each test requires a *different* line to be fixed, so fixing all the tests is not simply changing one line that is shared across all the tests.

Reflecting on our PRs and our limited discussions with the developers, the lessons learned are that providing (1) steps to reproduce test failures and (2) explanations of why fixing NIO is beneficial can improve the likelihood that PRs get accepted. For example, for one PR [7], we had initially reported the failure message from running the test twice without providing (1) or (2). The developer commented "I'm not sure how this would solve the [failure], and I've never come across it" and closed our PR. After we provided (1) and (2), including "clean state pollution so that some other tests won't fail in the future", the developer promptly reopened and merged our PR, replying "Thanks for the explanation. The PR makes sense". Our recent PRs include both (1) and (2).

A4: Developers are generally positive about fixes for NIO tests; providing reproducing steps and explaining the motivation help.

4.5 RQ5: NIO – Victim – Polluter Comparison

Every NIO test is both latent-victim and latent-polluter, and some NIO tests may be victims or polluters in their test suite. To understand how the NIO tests that we detect relate to OD-related tests detected in prior work, we intentionally use the same projects/modules and commits as prior studies for Java [61, 92] and Python [39] that have reported some victims or polluters. Detecting victims is expensive and typically requires running many random orders of test suites or sophisticated analyses of test dependencies [36, 58, 92, 99]. Detecting (all) polluters is even more expensive as the detection can require checking all test pairs [36, 88, 92]. In fact, the Python dataset [39] that we use reports only victims but does not report polluters.

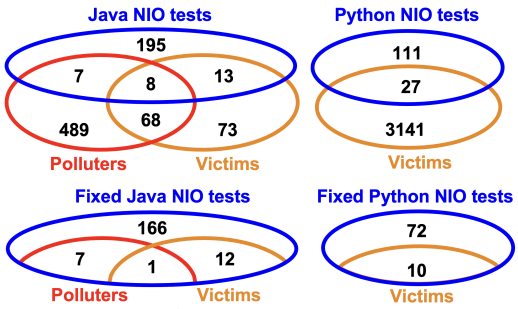

Figure 6: Number of (fixed) Java and Python NIO tests and their overlap with victim and (for Java only) polluter tests.

Figure 6 (top row) shows Venn diagrams relating the number of NIO, victim, and (for Java only) polluter tests in these datasets. Of 223 Java NIO tests, only 13 are victims but not polluters, 7 are polluters but not victims, and 8 are both victims and polluters. Of 138 Python NIO tests, only 27 are victims. We find that a large number of victims and polluters detected in prior work are not NIO tests. On the other hand, despite the relation of victims and polluters to NIO tests, the majority of the NIO tests that we detect were *not* detected before.

Figure 6 (bottom row) shows Venn diagrams focusing on the 268 NIO tests that we fix. Of 186 fixed Java NIO tests, only 12 are also victims, 7 are also polluters, and 1 is both victim and polluter. Of 82 fixed Python NIO tests, only 10 are victims. The Venn diagrams show some of the diversity of the NIO tests that we fix.

A5: NIO tests are related to but not subsumed by OD tests; detecting NIO tests can be an effective way to preempt OD tests.

5 CASE STUDIES

We next discuss some interesting examples of NIO tests that we inspect or fix. While all NIO tests pollute some part of the shared state, different tests pollute different parts. These examples illustrate various parts of the shared state that cause NIO tests.

5.1 Java – Database

Three tests from `querydsl` [82] (Section 4.2) are NIO because of the state polluted through a database stored on the disk. Specifically, the test fails an assertion (Line 16 in Figure 5) with the message `expected:<5> but was:<10>`. The test class `SearchQueryTest` extends `AbstractQueryTest` whose `@Before public void setUp()` method inserts five new users into a database (Line 8). Whenever JUnit runs the test class, it invokes the `setup` method that inserts five more users into the database. Indeed, if we run the test in two JVMs, the failure in the second JVM is due to every user being added twice.

Our inspection reveals that each JVM run adds some files in the directory `querydsl-hibernate-search/target/lucene/indexes/com.querydsl.hibernate.search.User/`. These files are database-related, persist across different JVMs, and make the test NIO. Interestingly, these files are only persisted *after* the JVM finishes. Therefore,

```
1  @Test
2  public void testAsync() {
3    RpcContext rpcContext = RpcContext.getContext();
4    Assert.assertFalse(rpcContext.isAsyncStarted());
5    rpcContext.setAsyncContext(new AsyncContext...);
6    ... // checks no asyncContext has started
7    RpcContext.startAsync();
8    Assert.assertTrue(rpcContext.isAsyncStarted());
9    asyncContext.write(new Object());
10   ... // assert something was done by asyncContext
11   rpcContext.stopAsync();
12   Assert.assertTrue(rpcContext.isAsyncStarted());
13 + RpcContext.removeContext();
14 }
```

Figure 7: Developer fix for NIO test in dubbo [28].

running the same test twice or even more times in the same JVM, from a clean disk state, does *not* lead to any test failure.

The fix for these tests is simple, just correcting a typo: changing `FileUtils.delete(new File("target/lucene3"))` to `FileUtils.delete(new File("target/lucene"))` in the setup method. This project uses Travis CI for continuous integration, but this issue is *not* detected in CI because it runs the entire test suite (without repeated tests) in one JVM always from a clean disk state. In contrast, running the test suite multiple times on the same machine (e.g., a developer's laptop) would have detected the issue. We opened a PR [8] with our fix, and the developers accepted it and replied "Thanks! 👍".

5.2 Java – File System

The test `TestViewfsWithNfs3.testNfsRenameSingleNN` from `hadoop` checks whether it can rename a file represented by an `HdfsFile` object. This test is NIO because of disk updates. The test first gets the `HdfsFile` that it tries to rename and checks the status of this `HdfsFile`. The test then renames that `HdfsFile` and checks its status after renaming. In the second run on the same JVM, this test raises a `NullPointerException`, specifically from invoking `statusBeforeRename.isDirectory()`. Before renaming, the test checks that the `HdfsFile` is not a directory. The problem is that the test gets the `HdfsFileStatus` object for `statusBeforeRename` based on the file name, but the name has been changed in the first run, so `statusBeforeRename` becomes `null` and causes the exception. Our proposed fix [9] renames the `HdfsFile` again back to its original name at the end of the test, and was accepted with "Merged. Thank you".

In the same project, the test `TestTaskProgressReporter.testBytesWrittenRespectingLimit` writes some bytes to the local file system. It also increments some counters that are written to the file system. However, after the test finishes, the counters are not reset, making one assertion fail when the test runs for the second time. Our fix [10] invokes `FileSystem.clearStatistics()` to reset the counters at the end of the test, and was also accepted.

5.3 Java – Heap Reachable from Static Fields

The most common case for NIO tests is heap "pollution": either the static fields themselves or the objects reachable from the static fields are polluted. Figure 7 shows an example NIO test from `Dubbo` [28]. The test `testAsync` starts by getting a Remote Procedure Call context (Line 3). The test aims to check whether the context properly exercises some task asynchronously. In doing so, the test also checks

```
1  @Test
2  public void testSigTermedFunctionality() throws ... {
3    AppContext mockContext = Mockito.mock(AppContext.
       class);
4    JHEventHandlerForSigtermTest jheh =
5      new JHEventHandlerForSigtermTest(mockContext, 0);
6    // adds some jobId to the static fileMap
7    jheh.stop();
8    // adds some jobId to the static fileMap
9    jheh.stop();
10   // assertions at the end of the test
11 }
12 // a method to execute jheh.stop()
13 @Override
14 protected void serviceStop() throws Exception {
15   // log the info
16   for (Map.Entry<JobId,MetaInfo> jobIt : fileMap.
       entrySet()) {
17     JobId toClose = jobIt.getKey();
18     // log the info
19     final Job job = context.getJob(toClose);
20     int successfulMaps = job.getCompletedMaps()
21       - job.getFailedMaps() - job.getKilledMaps();
22     // NullPointerException raised in the second run
23     // stop the job
24   }
25 ...
26 // helper class for testSigTermedFunctionality
27 class JHEventHandlerForSigtermTest extends
       JobHistoryEventHandler {
28   public JHEventHandlerForSigtermTest(AppContext
       context, int startCount) {
29     super(context, startCount);
30 +   JobHistoryEventHandler.fileMap.clear();
31   }
32 }
```

Figure 8: Our fix for NIO test in hadoop [43].

four times (Lines 4, 6, 8, and 12) whether the async task in `rpcContext` has started. The first two checks before the `rpcContext` starts the async task (Line 7) are expected to be `false`, while the later two checks are expected to be `true`. The test is NIO because the `rpcContext` (from Line 3) is *shared* in all runs of this test. Therefore, the first check (Line 4) in the second run fails, because the async task has already started during the first run. Note that even though Line 11 stops the async task, the check on Line 12 still passes in the first run (while Line 4 fails in the second run) because `rpcContext.isAsyncStarted()` is simply checking whether the async task has started before, and not whether it is still ongoing. In the latest version, the developers have cleaned the state pollution by adding `RpcContext.removeContext();` at the end (Line 13), so the test is "N/A" in Table 2.

`TestJobHistoryEventHandler.testSigTermedFunctionality` from `hadoop` is NIO. Figure 8 shows the relevant code snippet. The root cause of the failure is that this test adds some entries to the static field `JobHistoryEventHandler.fileMap`, which is shared among tests, and does not remove the entries from the map. `JobHistoryEventHandler.fileMap` has as keys the ids of the jobs that have been created. The call `jheh.stop()` (Line 7) in the test calls another method `serviceStop` (Line 14), which iterates over the `JobHistoryEventHandler.fileMap` to get the job ids, then gets each `Job` object by its id, and finally stops it. After the first test run, the created `Job` objects are cleaned, but their stale ids remain in `JobHistoryEventHandler.fileMap`. Therefore, in the second run, the test gets `null` from these stale ids (Line 19) and throws `NullPointerException` (Line 22). Our fix is to clear `JobHistoryEventHandler.fileMap` at the

```
def cmd_mock():
    def _cmd_mock(name: str):
        cmd.__overrides__[name] = ['/bin/true']
    yield _cmd_mock
-   cmd.__overrides__ = []
+   cmd.__overrides__ = {}
```

Figure 9: Our fix for NIO test in `benchbuild` [25].

end of the constructor of class `JHEventHandlerForSigtermTest`, a helper class of this test (Line 30). In fact, the constructor of another helper class, `JHEvenHandlerForTest`, in the same file clears `JobHistoryEventHandler.fileMap`, so the same clean up needs to take place in the constructor of `JHEventHandlerForSigtermTest`. Our PR [11] for this test has been merged by the developers.

`MetricsTest.shouldRecordCircuitBreakers` from `riptide` is another test that is NIO due to the pollution of the shared static field. This test adds new `timers` to the `SimpleMeterRegistry` but does not clear them. In the second test run, the assertion `assertEquals(4, timer.count())` at the end fails because the `timer.count()` has been incremented to 8 after two runs. Interestingly, this state pollution not only makes the test fail in the second run but also causes another test, `shouldRecordRequests` in the same test class, to fail. In other words, `shouldRecordCircuitBreakers` is a polluter for the victim `shouldRecordRequests`. Because this issue affects multiple tests, we do not fix just one test body but add to the class a `teardown()` method, which cleans `SimpleMeterRegistry` after each test run. Our PR [12] for this test was promptly accepted (within 10 minutes), and the developers gave a thumb up and thanked for our contribution.

5.4 Java – System Property

`SecurityUtilsTest.testEnsureTrustStoreSettings` in `admiral` tests whether it can properly set some system properties. For example, for `SECURITY_PROPERTIES`, the test starts by getting the value of the system property with `System.getProperty(SECURITY_PROPERTIES)`. Then it checks that this property has not been set, by comparing the value to `null` and an empty string. This check passes in the first test run. The test then runs `System.setProperty(SECURITY_PROPERTIES, ...);` and `assertEquals(...);` to set another value to that system property and to assert that it has been properly set. The test is NIO because it does not reset this system property at the end. In the second run of the test on the same JVM, the first assertion (checking that this property has not been set) fails.

The fix for this test is to clean the polluted system properties, e.g., we add `System.clearProperty(SECURITY_PROPERTIES);`. Our PR [13] was pending review before the project got archived.

5.5 Python – Buggy Cleaning

The test `test_cli_slurm.test_slurm_command` from `benchbuild` [25] is NIO due to an interesting state pollution: developers have code to clean the state but mistakenly pollute the type of a variable so that the test fails on the second run. This type mistake is more likely to appear in a dynamically typed language. Figure 9 shows the relevant code and our fix. The test calls the function `cmd_mock` that itself returns a function `_cmd_mock` that can add the corresponding value of the key `name` to a dictionary called `cmd.__overrides__`. Note that `cmd.__overrides__` is a global variable shared among test

```
def to_zero(tvd, northing, easting, surface_northing,
  surface_easting):
    # perform some checking
-   northing -= surface_northing
-   easting -= surface_easting
+   northing = northing - surface_northing
+   easting = easting - surface_easting
    return tvd, northing, easting

# initialization for global variables g1, g2, ..., g5
g1 = ...
def test_zero():
    # global variables passed in as arguments
    v1, v2, v3 = to_zero(g1, g2, g3, g4, g5)
    np.testing.assert_equal(...) # assertion
```

Figure 10: Our fix for NIO test in `wellpathpy` [93].

```
def test_celery_integration():
    server_address = ("", 8080)
    server = HTTPServer(server_address, Handler)
    # perform some assertions
+   server.socket.close()
```

Figure 11: Our fix for NIO test in `pybrake` [79].

runs. In the second run, the test fails reporting `TypeError: list indices must be integers or slices, not str`. The root cause is `cmd.__overrides__ = []` that sets the global variable to be an empty *list*. While developers had thought to clean the state, they mistakenly wrote the wrong cleaning code. Our fix changes the empty list to the empty dictionary. The developers accepted our PR [14] and said "Thanks, good catch!".

5.6 Python – Function Side Effect

The test `test_location.test_zero` from `wellpathpy` [93] is NIO due to state pollution stemming from the side effects in the function under test. Figure 10 shows the test and the function `to_zero`. The test calls `to_zero` by passing 5 global variables (of type numpy arrays) initialized outside `test_zero`. The second run of the test fails, causing `AssertionError` when executing Line 14. The root cause is that `to_zero` modifies the data in the numpy arrays passed in (namely `g2` and `g3`). Within `to_zero`, `northing` and `easting` point to the same numpy arrays as `g2` and `g3`, respectively. A discussion of such aliasing for numpy arrays is on StackOverflow [15]. Our fix replaces the operator `-=` that modifies array data *in place* with an assignment that creates new arrays and does not modify the arrays passed in. The developers merged our PR [16] and commented "This is a good change".

5.7 Python – Network Related

The test `test_celery_integration.test_celery_integration` from `pybrake` [79] is NIO due to state pollution related to network. Figure 11 shows the relevant code and our fix. The second test run throws `OSError: [Errno 98] Address already in use`. The reason is that the test does not release the network resource at the end of the execution, and therefore, the second run cannot initialize the server using the same address. Our fix is to close the server to make the address reusable after the test execution. The developers merged our PR [17].

6 DISCUSSION

Motivation and cost for rerunning a passing test. One could question why rerun tests twice when it is not usually done and takes time. The cost of rerunning tests twice to detect NIO tests has the benefit to *preempt* order-dependent (OD) tests, a prominent category of flaky tests [31, 58, 65, 99]. As a cost-effective approach to proactively detect OD-related tests, instead of rerunning all tests twice all the time, developers could rerun (1) only sometimes: all tests periodically (e.g., every weekend) or (2) only some tests: newly-added or recently-modified tests for all regression runs. We previously proposed these two options for other flaky-test-detection approaches [61] but not for rerunning tests twice.

Evidence of NIO tests becoming polluters or victims. We describe the history of two examples from `hadoop`. An example NIO test that became a polluter is `TestJobHistoryEventHandler.testSigTermedFunctionality`, which one developer had added (in `5f52156a`) and later became a polluter when another developer added (in `64e4fb98`) three victims ten months later. For this NIO test, the developers accepted our fix [11] with compliments. Had the first developer used our approach to detect NIO tests, the test could have been fixed before the victims were added. An example NIO test that became a victim is `TestTaskProgressReporter.testTaskProgress`, which one developer had added first (in `7e6f384d`) and then became a victim when another developer added (in `cb26cd4b`) a polluter about seven months later. Yet again, using our approach could have prevented the later polluter-victim pair.

7 THREATS TO VALIDITY

Some key threats to validity are the runtime ratios and whether the tests that we detect are really NIO and worth fixing. Our comparison of the runtime for different modes (Section 4.3) can be affected by the noise in the measurement of time. To mitigate this threat, we run the experiments on isolated Azure machines, and claim only a general trend of the overhead of different modes.

As our evaluation for NIO tests involves rerunning the tests, our results could be affected by flakiness itself. For example, a test may appear NIO (first run passes, second run fails), although the test is actually idempotent and happens to exhibit NIO-like results because of some nondeterminism. To mitigate this threat, we rerun the NIO tests with various tools and additionally manually inspect all detected NIO tests to obtain higher confidence in the tests that we study. In fact, the projects under evaluation likely have more NIO tests that are not detected because of tool limitations (e.g., `pytest-repeat` does not run certain kinds of Python tests).

We also use many existing tools and modify some to detect NIO tests. In principle, many of these tools, such as Maven Surefire [68], JUnit [51], `pytest` [80], or `pytest-repeat` [81], could have bugs that impact our results. We mitigate this threat by choosing some of the most widely used build systems and testing frameworks. Our own modifications to iDFlakies [58] are more likely to have bugs. We mitigate this threat by having multiple authors check iDFlakies modifications and manually inspect various results. Finally, the best way to alleviate concerns about usefulness of NIO tests is to provide fixes that developers largely accept: paraphrasing the saying "the proof is in the pudding", we could say "the proof is in PRing", i.e., opening pull requests that get accepted.

8 RELATED WORK

A recent survey [77] reviews many papers that have studied various causes and categories of flaky tests [21, 24, 29, 31, 36, 39, 42, 48, 57, 58, 60, 65, 72, 78, 85, 87, 88, 99]. These papers focus on tests that can pass or fail when running each test in a test suite only once. In contrast, we are the first to investigate NIO tests, which pass and fail when run twice in the same VM.

To help with the problem of flaky tests, various tools have also been proposed to help detect these tests [24, 36, 48, 58, 78, 99]. Most tools require running the tests and observing whether a test can pass in some runs and fail in other runs. Similar to these tools, our detection of NIO tests is based on whether a test can pass and fail in various runs. Unlike these prior tools, we run the tests twice in the same VM. Our findings for NIO tests are particularly important to the topic of flaky-test detection, because NIO tests can pollute the state used by other tests *and* they can fail themselves depending on the state set by other tests. Much of prior work on flaky-test detection has been on order-dependent tests, which pass or fail due to the pollution of *other* tests.

Beyond detecting flaky tests, related work has also proposed automatically fixing flaky tests [30, 63, 88, 98], tolerating flaky tests by resetting state [23], accommodating test dependencies [24], generating order-dependent flaky tests using mutations [41], and accommodating order-dependent tests in regression testing [59]. The ideas from these projects could help automatically fix or accommodate NIO tests in the future.

NIO tests are both latent-victim and latent-polluter tests. OracleFlakies [48] and PolDet [40] use sophisticated techniques to detect latent-victim and latent-polluter tests, respectively. However, they report many more tests that developers consider false positives. More recently, additional tools have been proposed to detect polluters [88, 95]. We report the *important intersection* of latent-victim and latent-polluter tests. Our approach is *simple* but effective at preempting polluters and victims. Our approach is also portable, e.g., we evaluate on both Java and Python tests.

9 CONCLUSION

This paper has focused on *NIO tests*, which pass in the first run but fail in the second run in the same VM. We have proposed three modes to detect NIO tests; these modes detect 223 NIO Java tests, and the most practical mode detects 138 NIO Python tests. These NIO tests are mostly new and have *not* been detected by prior research on flaky tests. We have opened pull requests for 268 NIO tests and developers have accepted many of them. We hope that our promising results on NIO tests and our publicly available dataset [18] can spur more research on this topic.

ACKNOWLEDGMENTS

We thank Yang Chen, Ruixin Wang, Satvik Eltepu, Reed Oei, and Jonathan Stein for their help. This work was partially supported by US NSF grants CCF-1763788 and CCF-1956374, and by Natural Science Foundation of China (Grant No. 62161146003), and the XPLORER PRIZE. Tao Xie (the corresponding author) is also with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China. We acknowledge support for research on flaky tests from Facebook and Google.

REFERENCES

[1] 2022. https://github.com/Activiti/Activiti/pull/3488
[2] 2022. https://github.com/apache/dubbo/pull/6936
[3] 2022. https://github.com/apache/hadoop/pull/2482
[4] 2022. https://github.com/spring-projects/spring-boot/pull/25435
[5] 2022. https://github.com/spring-projects/spring-boot/pull/27664
[6] 2022. https://github.com/josiest/geom/pull/1
[7] 2022. https://github.com/mtik00/yamicache/pull/10
[8] 2022. https://github.com/querydsl/querydsl/pull/2658
[9] 2022. https://github.com/apache/hadoop/pull/2724
[10] 2022. https://github.com/apache/hadoop/pull/2500
[11] 2022. https://github.com/apache/hadoop/pull/2499
[12] 2022. https://github.com/zalando/riptide/pull/1020
[13] 2022. https://github.com/vmware/admiral/pull/319
[14] 2022. https://github.com/PolyJIT/benchbuild/pull/425
[15] 2022. https://stackoverflow.com/questions/11585793/are-numpy-arrays-passed-by-reference
[16] 2022. https://github.com/Zabamund/wellpathpy/pull/50
[17] 2022. https://github.com/airbrake/pybrake/pull/163
[18] 2022. NIO Tests. https://sites.google.com/view/nio-tests
[19] 2022. TotT: Avoiding flakey tests. http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html
[20] Activiti 2022. https://github.com/activiti/activiti
[21] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *ICSE*.
[22] Jonathan Bell. 2014. Detecting, isolating, and enforcing dependencies among and within test cases. In *FSE Doctoral Symposium*.
[23] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*.
[24] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*.
[25] BenchBuild 2022. https://github.com/PolyJIT/benchbuild
[26] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. 2017. Test suite parallelization in open-source projects: A study on its usage and impact. In *ASE*.
[27] Zhen Dong, Abhishek Tiwari, Xiao Liang Yu, and Abhik Roychoudhury. 2021. Flaky test detection in Android via event order exploration. In *ESEC/FSE*.
[28] Dubbo 2022. https://github.com/apache/dubbo
[29] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *ISSTA*.
[30] Saikat Dutta, August Shi, and Sasa Misailovic. 2021. FLEX: Fixing flaky tests in machine learning projects by updating assertion bounds. In *ESEC/FSE*.
[31] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *ESEC/FSE*.
[32] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2000. Prioritizing test cases for regression testing. In *ISSTA*.
[33] Lamyaa Eloussi. 2016. Flaky tests (and how to avoid them). https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60
[34] Facebook testing and verification request for proposals 2019. https://research.fb.com/programs/research-awards/proposals/facebook-testing-and-verification-request-for-proposals-2019
[35] Martin Fowler. 2011. Eradicating non-determinism in tests. https://martinfowler.com/articles/nonDeterminism.html
[36] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*.
[37] 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*.
[38] Google. 2008. Avoiding flakey tests. http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html
[39] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An empirical study of flaky tests in Python. In *ICST*.
[40] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*.
[41] Sarra Habchi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2021. On the use of mutation in injecting test order-dependency. In *MSR*.
[42] Sarra Habchi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2021. A replication study on the usability of code vocabulary in predicting flaky tests. In *MSR*.
[43] Hadoop 2022. https://github.com/apache/hadoop
[44] 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*.
[45] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software. In *OOPSLA*.
[46] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. 2015. The art of testing less without sacrificing quality. In *ICSE*.
[47] Kim Herzig and Nachiappan Nagappan. 2015. Empirically detecting false test alarms using association rules. In *ICSE*.
[48] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*.
[49] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing. In *ICSE*.
[50] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *ICSE*.
[51] JUnit 2022. https://junit.org
[52] JUnit and Java 7 2012. http://intellijava.blogspot.com/2012/05/junit-and-java-7.html
[53] JUnit test method ordering 2022. http://www.java-allandsundry.com/2013/01/junit-test-method-ordering.html
[54] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. 2013. Optimizing unit test execution in large software programs using dependency analysis. In *APSys*.
[55] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *ICSE SEIP*.
[56] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *ISSTA*.
[57] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *ICSE*.
[58] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*.
[59] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-test-aware regression testing techniques. In *ISSTA*.
[60] 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 *ISSRE*.
[61] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. In *OOPSLA*.
[62] Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. 2021. When life gives you oranges: Detecting and diagnosing intermittent job failures at Mozilla. In *ESEC/FSE*.
[63] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing order-dependent flaky tests via test generation. In *ICSE*.
[64] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: Continuous prioritization for continuous integration. In *ICSE*.
[65] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.
[66] Maintaining the order of JUnit3 tests with JDK 1.7. 2013. https://coderanch.com/t/600985/engineering/Maintaining-order-JUnit-tests-JDK
[67] Maven 2022. https://maven.apache.org
[68] Maven Surefire plugin 2022. https://maven.apache.org/surefire/maven-surefire-plugin
[69] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE SEIP*.
[70] Gerard Meszaros. 2007. *xUnit Test Patterns: Refactoring Test Code*.
[71] John Micco. 2017. The state of continuous integration testing at Google. In *ICST*. https://bit.ly/2OohAip
[72] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. 2021. Verifying determinism in sequential programs. In *ICSE*.
[73] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *ISSTA*.
[74] Madan Musuvathi, Shaz Qadeer, and Thomas Ball. 2007. *CHESS: A systematic testing tool for concurrent software*. Technical Report MSR-TR-2007-149.
[75] Pengyu Nie, Ahmet Celik, Matthew Coley, Aleksandar Milicevic, Jonathan Bell, and Milos Gligoric. 2020. Debugging the performance of Maven's test isolation: Experience report. In *ISSTA*.
[76] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2020. Flake it 'till you make it: Using automated repair to induce and fix latent test flakiness. In *ICSE (Workshops)*.
[77] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A survey of flaky tests. *TOSEM* (2021).
[78] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests?. In *MSR*.
[79] pybrake 2022. https://github.com/airbrake/pybrake
[80] pytest 2022. https://docs.pytest.org/en/6.2.x
[81] pytest-repeat 2022. https://pypi.org/project/pytest-repeat
[82] Querydsl 2022. https://github.com/querydsl/querydsl
[83] Md Tajmilur Rahman and Peter C. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE*.
[84] Maaz Hafeez Ur Rehman and Peter C. Rigby. 2021. Quantifying no-fault-found test failures to prioritize inspection of flaky tests at Ericsson. In *ESEC/FSE*.

[85] Alan Romano, Zihe Song, Sampath Grandhi, Wei Yang, and Weihang Wang. 2021. An empirical analysis of UI-based flaky tests. In *ICSE*.
[86] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *TSE* (2001).
[87] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*.
[88] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*.
[89] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*.
[90] Test verification 2022. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification
[91] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An empirical study of flaky tests in Android apps. In *ICSME*.
[92] 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 *TACAS*.
[93] wellpathpy 2022. https://github.com/Zabamund/wellpathpy
[94] Eric Wendelin. 2022. Introducing flaky test mitigation tools. https://blog.gradle.org/gradle-flaky-test-retry-plugin
[95] Pu Yi, Anjiang Wei, Wing Lam, Tao Xie, and Darko Marinov. 2021. Finding polluter tests using Java PathFinder. *SEN* (2021).
[96] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *STVR* (2012).
[97] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. 2012. Regression mutation testing. In *ISSTA*.
[98] 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 *ICSE*.
[99] 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*.
[100] 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*.