# iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests

Ruixin Wang
ruixinwang@zju.edu.cn
Zhejiang University
China

Yang Chen
yangchen20@hust.edu.cn
Huazhong University of Science and Technology
China

Wing Lam
winglam@gmu.edu
George Mason University
USA

## ABSTRACT

Developers typically run tests after code changes. Flaky tests, which are tests that can nondeterministically pass and fail when run on the same version of code, can mislead developers about their recent changes. Much of the prior work on flaky tests is focused on Java projects. One prominent category of flaky tests that the prior work focused on is order-dependent (OD) tests, which are tests that pass or fail depending on the order in which tests are run. For example, our prior work proposed using other tests in the test suite to fix (or correctly set up) the state needed by Java OD tests to pass.

Unlike Java flaky tests, flaky tests in other programming languages have received less attention. To help with this problem, another piece of prior work recently studied flaky tests in Python projects and detected many OD tests. Unfortunately, the work did not identify the other tests in the test suites that can be used to fix the OD tests. To fill this gap, we propose iPFlakies, a framework for automatically detecting and fixing Python OD tests. Using iPFlakies, we extend the prior work's dataset to include (1) tests that can be used to reproduce and fix the OD tests and (2) patches for the OD tests. Our work to extend the dataset finds that reproducing passing and failing test results of flaky tests can be difficult and that iPFlakies is effective at detecting and fixing Python OD tests. To aid future research, we make our iPFlakies framework, dataset improvements, and experimental infrastructure publicly available.

## KEYWORDS

flaky tests, order-dependent test, Python, automated repair

**ACM Reference Format:**
Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516846

## 1 INTRODUCTION

To detect whether recent code changes introduce any bugs, developers typically perform regression testing after code changes. If all the tests pass, developers can safely merge their changes. However, if some tests fail, developers typically debug their changes to fix the failures. Unfortunately, flaky tests [12], which are tests that nondeterministically pass and fail when run on the same version of code, mislead developers about their changes, waste developers' time debugging failures in their changes, and reduce developers' trust in testing. In fact, many companies have published blogs (e.g., Fitbit [11], Gradle [17]) and research papers (e.g., Apple [5], Facebook [3], Google [13], Microsoft [6]) on flaky tests.

Much research has been done using Java projects in recent years to help with flaky tests. For example, Luo et al. [12] proposed the first taxonomy of flaky tests in Java projects. One prominent category of flaky tests identified in this taxonomy are *order-dependent (OD)* flaky tests, which are tests whose outcomes depend on the order in which tests are run. Since the taxonomy, much of the flaky-test work has focused on OD tests [1, 2, 7–9, 12, 15, 19].

To help with OD flaky tests, our prior work proposed iDFlakies [7], a framework for detecting and partially categorizing Java OD tests, and iFixFlakies [15], a framework for automatically fixing Java OD tests. On a high level, the two frameworks run tests in various orders to detect, categorize, and generate patches for OD tests. To detect OD tests, the frameworks run tests in random orders. To categorize OD tests for debugging and fixing, the frameworks run tests in certain orders to determine whether an OD test is a *victim* (a test that fails when run after another test, called a *polluter*, but passes when run before the polluter) or a *brittle* (a test that passes only when it is run after another test, called a *state-setter*, but fails when run before the state-setter).

In this paper, we focus our discussion of OD tests on victims as our prior work [15] found that 91% of OD tests are victims. Essentially, a victim fails because the victim reads some shared state that the polluter modifies. To generate patches, the frameworks first find the polluters and *cleaners* (tests that clean the polluted state shared by a victim and a polluter such that running the polluter, then cleaner, and finally the victim will result in the victim passing) for each victim. To find cleaners, our prior work [15] searches for such tests in the passing test suite runs of the victim. More recently, additional work has been proposed to generate cleaners [10]. An example of a victim, polluter, and cleaner is in Section 2. Once cleaners are identified, the frameworks then search the code within cleaners to propose a patch for the victim. To be brief, we refer to the tests (polluter, cleaner, state-setter) that are related to OD tests as *OD-related tests*.

Although Java flaky tests have received much attention (e.g., frameworks [1, 7, 15] that automatically detect, categorize, and patch), flaky tests in other programming languages, such as Python,


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

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516846

have received less attention. To help with this problem, Gruber et al. [1] recently studied flaky tests in Python projects. Their work studied flaky tests in 22,352 Python projects using a total of 400 runs for each project's test suite, and they found OD tests to be the most common category of flaky tests, accounting for 59% of all flaky tests detected. Despite providing an invaluable dataset of Python OD tests, the dataset does not contain OD-related test information, which are useful to generate patches for the OD tests.

In this paper, we aim to fill this gap by proposing *iPFlakies*, a framework for automatically detecting and fixing Python OD tests and the use of the framework to find OD-related tests to generate patches for Python OD tests. On a high level, iPFlakies encapsulates the functionalities of iDFlakies and iFixFlakies but for Python tests. iPFlakies is also much easier to setup – instead of requiring several commands to setup two frameworks, iPFlakies is one framework that requires just one command (`pip install ipflakies`).

Using iPFlakies on Gruber et al.'s dataset [1], we identify two main findings: (1) reproducing passing and failing test results of flaky tests is difficult, and (2) iPFlakies is effective at detecting and fixing Python OD tests. When we run the test suites that Gruber et al. detected at least one OD test, we find that only 64% of the test suites can be run without errors (e.g., missing dependencies), even though we follow a similar experimental methodology as Gruber et al. To help future research, we run all test suites using Docker and share our experimental infrastructure [4]. For the test suites that we can run, we observe a passing and failing run for 57% of the OD tests detected in prior work when we run the test suites in 100 random orders and when we run every test before each of the OD test. We also find 30 OD tests that were not detected in prior work. Lastly, we find that 63% of OD tests contain cleaners or state-setters, and iPFlakies can automatically fix 33% of such tests.

Overall, this paper makes the following main contributions:
**iPFlakies Framework**: A publicly available [4] framework that unifies the automatic detection and fixing of Python OD tests.
**Dataset**: The first dataset of OD-related tests (e.g., for victims, we identify polluters and cleaners) and patches for Python OD tests. Our dataset also includes the Docker images used to detect the OD-related tests and generate patches [4].
**Study**: We perform a study on Python OD tests from Gruber et al.'s dataset. We find that reproducing both passing and failing test results of flaky tests is difficult and that iPFlakies is effective at detecting and fixing Python OD tests.

## 2 BACKGROUND

OD flaky tests pass and fail due to the order in which the tests are run. They consistently pass in one order and consistently fail in another order, and they are categorized as either brittles or victims.

A brittle is an OD test that consistently fails when run in isolation. There must also be a certain test order containing another test, referred to as state-setter, such that running the state-setter before the brittle results in the brittle passing. A state-setter can be one or more test. However, our prior work [15] reported that it is rare for an OD test to require multiple tests to pass or fail.

A victim is an OD test that consistently passes when run in isolation but can fail if polluters run before the victim. If a *cleaner* is run in between polluter and victim, then the victim passes. A

```
1  def test_register_multiple(): # victim (V)
2    con.register_many(Ex, [Ex1, Ex2])
3    inst = con.get(List[Ex])
4    ... # check some properties of inst
5    assert type(inst[0]) is Ex1
6    assert type(inst[1]) is Ex2
7  def test_multiple_list_..._wrap(): # polluter (P)
8    con._clear_all()
9    con.register_many(Ex, [Ex2, Ex3])
10   con.register_many(Ex1, [Ex3, Ex4])
11   inst = con.get(MainWrapper)
12   ... # check some properties of inst
13 def test_class_mapping(): # cleaner (C)
14   con.initialize()
15   con.register_class(Ex, Ex2)
16   inst = con.get(Ex)
17   assert type(inst) is Ex2
```

**Figure 1: Example victim, polluter, and cleaner from `omer-saraf/IOCynergy` on GitHub.**

victim can pass when a cleaner is run in between a polluter and the victim, because cleaners clean the state polluted by a polluter. Similar to state-setters, each polluter and cleaner can be one or more test, but our prior work [15] has found that these categories of tests are rarely multiple tests. This prior work also found that 91% of OD tests are victims. Therefore, we focus our discussion of OD tests on victims. We next show an example of a victim OD test.

### 2.1 Victim OD Test Example

Figure 1 shows an example of a victim, polluter, and cleaner. The victim is `test_register_multiple()` (or `V` for short), the polluter is `test_multiple_list_..._wrap()` (or `P` for short), and the cleaner is `test_class_mapping()` (or `C` for short). The victim is from Gruber et al.'s dataset [1] and the polluter and cleaner are found by iPFlakies. All three tests change or read the `con` variable shared among the tests. Specifically, in `P`, Line 9 sets a list containing `Ex2` and `Ex3` for key `Ex` in `con`. Therefore, when `V` is run after `P`, the `con` variable would already contain the key `Ex` when `V` starts to run. This shared state is problematic because at Line 2 of `V`, `Ex` already exists in the map, and it cannot be mapped to a list containing `Ex1` and `Ex2`, which lines 5 and 6 are checking for. If `C` is run in between `P` and `V`, then Line 14 initializes the `con` variable and removes the `Ex` key along with its mapped value. Therefore, even if `V` is run after `P`, as long as Line 14 of `C` is run before `V`, then `V` will pass. When we apply iPFlakies to `V`, iPFlakies is able to find the polluter and cleaner for this victim and generate a patch (Line 14) in just 12.5 seconds.

## 3 IPFLAKIES

iPFlakies encapsulates the functionalities of iDFlakies and iFixFlakies. Figure 2 shows an overview of iPFlakies. We refer to the iDFlakies and iFixFlakies that are parts of iPFlakies with iDFlakies$_p$ and iFixFlakies$_p$, respectively, while we refer to the original iDFlakies [7] and iFixFlakies [15] from our prior work using their original names.

### 3.1 iDFlakies$_p$

On a high level, iDFlakies$_p$ detects and categorizes Python OD tests using two main components: Randomizer and Analyzer.
**Detecting flaky tests.** The goal of Randomizer is to run a test suite multiple times in random orders. To accomplish this goal,

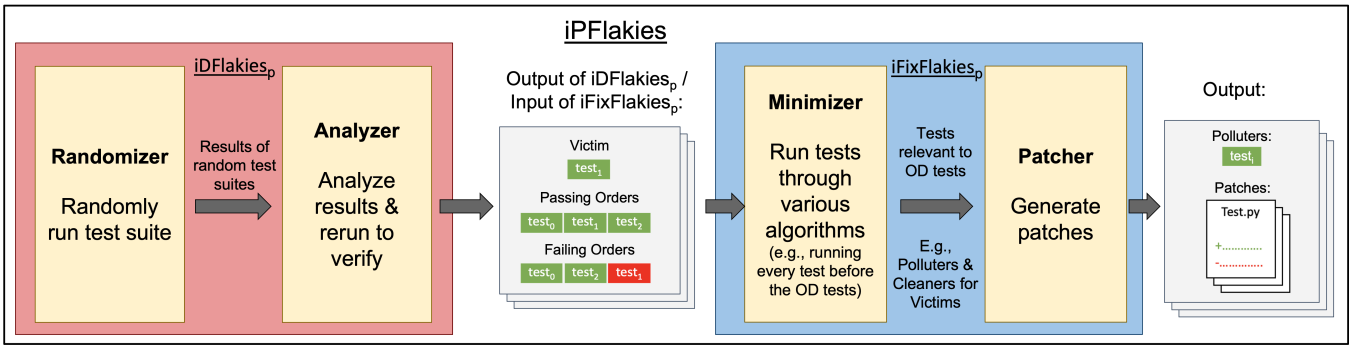


**Figure 2: Overview of iPFlakies.**

Randomizer accepts two main parameters: `-seed` for generating deterministic random orders and `-num_iterations` for controlling the number of random orders to run. To run tests, Randomizer relies on the `-random-order-seed` parameter from `PyTest`. `PyTest` uses this parameter to run tests in one random order. By default, `PyTest` cannot run a test suite in a random order multiple times. Therefore, to run test suites multiple times, Randomizer first uses a seed to generate other seeds (i.e., one seed for each random order to run) and then invoke `PyTest` multiple times with the generated seeds. The output of Randomizer is the execution result of tests from all of the random orders that were run.

**Categorizing flaky tests.** The goal of Analyzer is to analyze the execution results of tests from different runs to identify and categorize flaky tests. To accomplish this goal, Analyzer reruns passing and failing orders for each test that contains at least one passing and at least one failing order. On a high level, Analyzer accepts the output of Randomizer and two main parameters: `-num_orders` for controlling the number of passing and failing orders to rerun and `-num_rerun` for controlling the number of reruns for each order. If a flaky test does not consistently pass in its passing orders and does not consistently fail in its failing orders, then the test is categorized as *nondeterministic (NO)*. If a test is not NO, then Analyzer runs the test in isolation. If the test consistently passes in isolation, then it is a victim. If the test consistently fails, then it is a brittle. Lastly, if the test inconsistently passes and fails, then the test is actually NO. The output of Analyzer is a set of categorized flaky tests along with passing and failing orders for each OD test. Compared to the original iDFlakies, iDFlakies$_p$ is meant for Python instead of Java tests and iDFlakies$_p$ runs detected OD tests in isolation to further categorize the tests as victims or brittles.

## 3.2 iFixFlakies$_p$

On a high level, iFixFlakies$_p$ detects OD-related tests (e.g., polluters and cleaners for victims) and generates patches using two main components: Minimizer and Patcher.

**Finding OD-related tests.** The goal of Minimizer is to identify all OD-related tests for a given OD test. To identify OD-related tests, Minimizer takes an OD test and its passing and failing test orders as input, and then runs the tests through various algorithms. The parameter `-minimizer_mode` enables one to choose from two algorithms for detecting polluters and state-setters: (1) run each test in the test suite before the OD test or (2) use delta debugging [18] to minimize the failing and passing orders to find polluters and state-setters, respectively. Once polluters are identified for victims, the Minimizer then uses the algorithms again to find cleaners (e.g., run every test in between polluter and victim).

**Generating patches.** The goal of Patcher is to generate patches for OD tests using *helpers* (cleaners for victims and state-setters for brittles) from the output of Minimizer. When generating patches for an OD test, Patcher first copies all `Import` statements of a helper to the Python file containing the OD test. Next, Patcher copies the method bodies of the `setup` and `teardown` functions of the helper along with the method body of the helper itself directly into the beginning of the OD test. Patcher then checks whether the added code can make the OD test pass (e.g., victim passes when it is run after the polluter). If so, Patcher will further delta debug the added code to get the minimal patch. Note that for victims, iFixFlakies$_p$ can be configured to insert the patches at the beginning of the victims or at the end of the polluters. The parameter `patch_mode` allows the user to decide whether to get all possible patches, or simply the first patch that is able to fix the victim for all of its polluters. Compared to the original iFixFlakies, iFixFlakies$_p$ is meant for Python tests instead of Java tests and these additional parameters (e.g., first patch that fixes all polluters for a victim) are new. In the future, we plan to continue improving both the Java and Python variants.

# 4 EVALUATION

To evaluate iPFlakies, we consider the following RQs:

**RQ1:** How effective is iPFlakies at detecting OD tests?

**RQ2:** How effective is iPFlakies at generating patches for OD tests?

Before we describe the results of the RQs, we first describe the projects and the methodology we use for our evaluation.

## 4.1 Evaluation Projects

For our evaluation of iPFlakies, we use the same projects and commits used in a prior study of Python OD tests [1]. We use the same project commits because the prior work detected flaky tests and categorized them as OD or NO in the specific project commits, thereby allowing us to compare flaky tests that we detect to previously detected tests (Section 4.3.1).

To setup the test suites in Gruber et al.'s dataset [1], we obtain the same 610 projects and the same version of the projects in which Gruber et al. detected at least one OD test. Similar to Gruber et al.'s work, we then search for files that developers are likely to keep dependency requirements of their projects in. Specifically,

we search for `Pipfile` and `*requirements*.txt` files to install the dependencies of each project.

Once we identify the requirements-related files, we run `pip install -r` with the files to setup the dependencies of the projects. Regardless of whether requirements-related files are found, we then proceed to run the project's test suite by running `pytest`. If no tests are run from the `pytest` command, then we consider the test suite to be incorrectly setup. Conversely, if at least one test is run, we consider the test suite to be correctly setup. All of our experiments are run inside a Python `3.8.12` Docker image, which we make publicly available [4]. Our Docker image is first obtained from running `docker pull python:3.8.12` and then improved to contain the necessary dependencies required by our experimental scripts and iPFlakies.

The OD tests in Gruber et al.'s dataset are from 610 projects. Following our described project setup, we correctly run 64% (392) of the projects. The other 36% (218) of projects we could not correctly run due to missing dependencies or the projects require specific Python versions that differs from the one we used (3.8.12). To help with some of these issues, we plan to try approaches that fix Python builds caused by dependency errors [14] in the future.

Of the 392 projects we are able to run, 26 projects timed out following our experimental methodology described in Section 4.2 and one project's tests deleted files on the filesystem. We remove these 27 projects from our evaluation and list why they are removed on our website [4]. In the end, we are able to run 1978 OD tests in 365 projects from Gruber et al.'s dataset. To aid future research, we include the code and dependencies of the 365 projects as part of our dataset [4].

## 4.2 Methodology

*4.2.1 RQ1.* In this RQ, we study how many tests that were detected to be OD in Gruber et al.'s dataset can be detected as flaky (OD or NO) with iPFlakies and how many flaky tests iPFlakies can detect that were not detected by Gruber et al. To avoid confusion, we refer to the OD tests detected by Gruber et al. as *potentially-OD* (*potent-OD* for short) tests and the ones we detect as OD tests. To evaluate this RQ, we perform the following steps.

**Step 1: Run each test suite in 100 random orders and categorize flaky tests.** We first run each test suite in 100 random orders. Using the results of the 100 runs, we then categorize detected flaky tests as NO or as likely OD. A test is categorized as NO if the test nondeterministically passes and fails in any given prefix of tests (i.e., tests running before the NO test). A test is categorized as likely OD if the test passes and fails in 100 runs but the test deterministically passes or fails in any given prefix of tests.

To verify whether each likely OD test is OD, we rerun up to three passing and up to three failing orders for each test. Each order is rerun three times. If a likely OD test has a test order in which it nondeterministically passes and fails, it is categorized as NO. Otherwise, the test is categorized as OD.

In summary, each OD test detected in our work is run 100 times in random test suite orders plus at least six and at most 18 times to verify the test. The main difference between our methodology to detect OD tests and the methodology from prior work [1] is that our methodology includes a step to verify the detected OD tests. Details for how this difference affects the OD tests that we detect are presented in Section 4.3.1.

**Step 2: Run each potent-OD test detected by Gruber et al. and OD test from Step 1 in isolation 10 times.** For each time, we run one test in its own Python interpreter. If any test passes and fails in the 10 runs, then the test is categorized as NO. This step also categorizes OD tests as either victims (always passed in isolation) or brittles (always failed in isolation).

**Step 3: Run each test in the test suite before each potent-OD test and before each OD test from Step 2.** Running each test before potent-OD tests has been shown to be effective for confirming OD tests in prior work [9, 15] and helps find OD-related tests (polluters for victims and state-setters for brittles).

To limit the cost of our experiments, we set each test run (i.e., a test suite or just a pair of tests) to time out after 864 seconds (i.e., Step 2 takes at most 24 hours for 100 runs).

*4.2.2 RQ2.* In this RQ, we study the effectiveness of iPFlakies to generate patches for the OD tests we detect. To generate patches, iPFlakies relies on state-setters for brittles and cleaners for victims. Therefore, for all of the OD tests that we detect, we attempt to find all of the state-setters and cleaners. We obtain the state-setters for brittles from running each test in the test suite before the brittle (Step 3 from Section 4.2.1). For cleaners, we follow a similar methodology to Step 3 – for every polluter and victim pair, we run every test in between the polluter and victim. If for any given test, the victim passes, then we rerun the three tests three times to verify that the victim passes in all three runs. Using the state-setters and cleaners, we use iPFlakies to generate one patch that can help brittles pass in isolation and victims pass when they are run after any of their polluters. We find only four victims where we could generate a patch for some polluters but not for all of its polluters.

## 4.3 Results

*4.3.1 RQ1.* Of the 1978 potent-OD tests that we can run, we find that 803 are victims, 318 are brittles, 225 are NO, and 632 are not flaky (453 always passed and 179 always failed). Overall, our results suggest that at least 11% of potent-OD tests are actually NO and that up to 32% of the potent-OD tests may not be flaky. The high percentage of NO tests is likely because of methodological differences that we and Gruber et al. used to categorize tests. Namely, Gruber et al. ran test suites in (1) one order 200 times and (2) 200 random orders (running each order just once). Tests whose execution result changed in (1) are categorized as NO, while tests that changed in (2) are categorized as OD. In contrast to Gruber et al., we do not assume that all changes from random orders imply that an OD test is detected (such changes can still be due to NO tests). In fact, our prior work [8] found that many NO tests are nondeterministic, order-dependent tests, which are tests that have statistically significantly different failure rates in different test orders. Finer-grained categorization of NO tests continues to be an important future research topic. Lastly, the high percentage of tests that may not be flaky is likely because we run fewer random orders than Gruber et al. and because of differences in our experimental infrastructure.

Our experiments with iPFlakies also detect 30 OD tests not detected in Gruber et al.'s dataset. Specifically, our experiments detect 23 victims and 7 brittles that were not detected in Gruber et al.'s

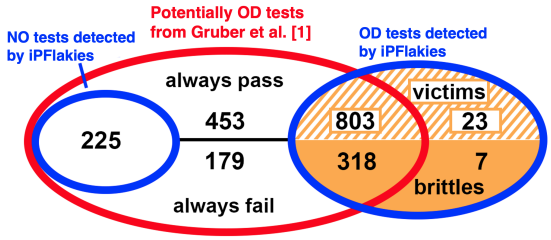


**Figure 3: Number of NO and OD tests detected by iPFlakies and their overlap with tests detected by Gruber et al. [1].**

dataset. Figure 3 shows a detailed breakdown of how the flaky tests detected by iPFlakies overlap or not with the potent-OD tests detected by Gruber et al. Our prior work [15] found that only 3% of Java OD tests required more than one test to exhibit flakiness (i.e., brittles needing more than one test as a state-setter or victims needing more than one test as a polluter). Compared to our prior work [15], we find that 26% of OD tests (31% for victims and 12% for brittles) require more than one test to exhibit flakiness.

**A1:** Using iPFlakies on the same set of project commits as prior work [1], we detect 57% of the OD tests detected in prior work and detect an additional 30 OD tests not detected in prior work.

*4.3.2 RQ2.* Of the 1151 OD tests (1121 tests from Gruber et al. and iPFlakies + 30 tests from iPFlakies), we find cleaners and state-setters (referred to as *helpers* [15]) for 537 tests (251 victims + 286 brittles). Note that some brittles may not have state-setters because their state-setters are composed of more than one test. We detect these brittles in steps 1 and 2 described in Section 4.2.1 but are unable to find their state-setters in Step 3. Of the 537 tests, iPFlakies can generate at least one patch for 33% (175) of tests. For all tests, except for four tests, the generated patch can help victims pass for all of their polluters and for brittles to pass in isolation.

For the projects that contain at least one test that iPFlakies could not fix, we inspect at least one test from each of these projects. We find that iPFlakies failed to fix some tests because (1) the OD test or OD-related tests require specific parameters (e.g., a cleaner only cleans when run with a specific parameter) or (2) some global variables or methods are needed by the helper but are missing when we copy the helper code into the OD test, which often led to runtime errors. In the future, we plan to improve iPFlakies to deal with these issues and to consider other ways to find helpers [10].

**A2:** Using iPFlakies on 537 OD tests, we find that iPFlakies can generate patches for 33% of the OD tests with the overwhelming majority of the patches enabling victims to run after any polluter.

## 5 THREATS OF VALIDITY

One threat to validity is that we use projects from Gruber et al.'s dataset [1]. The projects in this dataset may not be representative of all Python projects. We also could not setup and run all projects in the dataset due to missing dependencies for some projects and a small number of projects timed out. Our methodology to resolve dependency issues includes broadly searching for two different commonly used files to specify dependencies for Python projects. However, such efforts may nonetheless be inadequate for some projects. Nevertheless, we attempt to mitigate this threat by still using a substantial number of projects in our evaluation of iPFlakies.

As this work is on flaky tests, it is likely that test results can be different if they are run in different environments or run more times. To mitigate this threat, we run our experiments inside Docker containers and make available as much of our dataset and infrastructure [4] as possible. We also run each test suite in our evaluation projects in 100 random orders. For OD tests, the lowest failure rate reported in prior work [16] is 4.5%. With 100 random orders, the likelihood of not observing a failure for such a test is merely 1%.

## 6 CONCLUSION

In this paper, we proposed iPFlakies, a framework for detecting and fixing Python OD tests. Using iPFlakies, we extended an existing dataset of Python OD tests to include (1) OD-related tests that can be used to reproduce and fix OD tests and (2) patches for the OD tests. To aid future research, we make our iPFlakies framework, dataset improvements, and experimental infrastructure publicly available [4]. In the future, we plan to submit the generated patches to developers and improve iPFlakies to work on more projects (e.g., support more testing frameworks).

## ACKNOWLEDGMENTS

We thank Marcelo d'Amorim, Darko Marinov, August Shi, Anjiang Wei, and Pu Yi for their feedback on this work.

## REFERENCES

[1] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An empirical study of flaky tests in Python. In *ICST*.
[2] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*.
[3] 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*.
[4] iPFlakies website 2021. https://sites.google.com/view/ipflakies
[5] Emily Kowalczyk, Karan Nair, Zebao Gao, Leo Silberstein, Teng Long, and Atif Memon. 2020. Modeling and ranking flaky tests at Apple. In *ICSE SEIP*.
[6] Wing Lam, Kivanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *ICSE*.
[7] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*.
[8] 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*.
[9] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. In *OOPSLA*.
[10] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing order-dependent flaky tests via test generation. In *ICSE*.
[11] Serban Liviu. 2019. A machine learning solution for detecting and mitigating flaky tests. https://eng.fitbit.com/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests
[12] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *FSE*.
[13] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE SEIP*.
[14] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *ISSTA*.
[15] 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*.
[16] 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*.
[17] Eric Wendelin. 2021. Introducing flaky test mitigation tools. https://blog.gradle.org/gradle-flaky-test-retry-plugin
[18] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *TSE* (2002).
[19] 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*.