Flake It 'Till You Make It: Using Automated Repair to Induce and Fix Latent Test Flakiness

Owain Parry
University of Sheffield

Gregory M. Kapfhammer
Allegheny College

Michael Hilton
Carnegie Mellon University

Phil McMinn
University of Sheffield

ABSTRACT

Since flaky tests pass or fail nondeterministically, without any code changes, they are an unreliable indicator of program quality. Developers may quarantine or delete flaky tests because it is often too time consuming to repair them. Yet, since decommissioning too many tests may ultimately degrade a test suite's effectiveness, developers may eventually want to fix them, a process that is challenging because the nondeterminism may have been introduced previously. We contend that the best time to discover and repair a flaky test is when a developer first creates and best understands it. We refer to tests that are not currently flaky, but that could become so, as having *latent* flakiness. We further argue that efforts to expose and repair latent flakiness are valuable in ensuring the future-reliability of the test suite, and that the testing cost is greater if latent flakiness is left to manifest itself later. Using concrete examples from a real-world program, this paper posits that automated program repair techniques will prove useful for surfacing latent flakiness.

ACM Reference Format:
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 *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392177

1 INTRODUCTION

Flaky tests are software tests that appear to exhibit an element of randomness in their outcome despite covering code that has not changed [2, 3]. Developers report that such tests represent a frequent and non-negligible problem [3]. This paper proposes a technique, named FITTER (Flakiness Inducing TesT crEation and Repair), that uses Automated Program Repair (APR) techniques to expose and assist in the fixing of latent sources of test flakiness, i.e., those not currently manifest but that may later become so. Focused on surfacing two critical sources of latent flakiness — test order dependencies and test resource leaks [3, 10, 18] — the proposed technique aims to automatically generate a flakiness-inducing test (*FIT*) that reveals the latent flakiness within some developer-written test (*DWT*). We contend that APR techniques will be useful in achieving this goal. Our position is that by investigating the flakiness surfaced by the *FIT*, the developer will better understand the root cause and will thus be far better equipped to remove it during the subsequent improvement of the *DWT*. After repairing the source of test flakiness, the developer can discard the *FIT* or keep it as a support for resolving future test flakiness issues. Although we anticipate that the technique is most useful when writing new tests, we judge that it will also be an aid to test maintainers who must find and fix sources of test flakiness in existing test suites.

We propose to evaluate a FITTER implementation with programs created in Python since the 2019 StackOverflow Developer Survey called it "the fastest-growing major programming language today" [16] and Facebook reports that 21% of its infrastructure codebase, "comprising millions of lines of code, thousands of libraries and binaries", is in Python [4]. Since much of the current work on flaky tests focuses on Java programs (e.g., [1, 2, 5, 7, 10, 15]), targeting FITTER for Python will both help to avoid overfitting research in this field while also supporting experiments that can identify how a programming language influences test flakiness.

```python
@pytest.mark.parametrize(
    "config_dir", ["../hydra/test_utils/configs"],
)
@pytest.mark.parametrize(
    "config_file, overrides, expected",
    [
        (None, [], {}),
        (None, ["foo=bar"], {"foo": "bar"}),
        ("compose.yaml", [], {"foo": 10, "bar": 100}),
        ("compose.yaml", ["group1=file2"], {"foo": 20, "bar": 100}),
    ],
)
class TestCompose:
    def test_compose_decorator(
        self, hydra_global_context, config_dir, config_file, overrides, expected
    ):
        with hydra_global_context(config_dir=config_dir):
            ret = hydra.experimental.compose(config_file, overrides)
            assert ret == expected
```

Figure 1: An Excerpt from a Test for the Hydra Project

2 EXAMPLE OF STATE POLLUTING TESTS

As a motivating example this paper envisages how FITTER could exploit a latent source of flakiness in the form of an opportunity for state pollution. The Python code in Figure 1 is an excerpt of a test class from the open-source Facebook Research project called Hydra[1]. Hydra aims to provide a framework that simplifies the development of Python programs by giving developers the facilities to compose and override configurations. Hydra's test suite leverages the PyTest[2] framework that supports the parameterisation of an

[1]https://github.com/facebookresearch/hydra
[2]https://docs.pytest.org/en/latest/

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392177

```
1     overrides.append("fooooooooo=bar")
2     with hydra_global_context(config_dir=config_dir):
3         ret = hydra.experimental.compose(config_file, overrides)
4         assert ret == expected
```

Figure 2: A *FIT* Generated from *DWT* by Inserting Line 1[3]

individual test or a test suite. In this case, the class is parameterised such that a single argument is provided for `config_dir` and four sets of arguments are provided for `config_file`, `overrides`, and `expected`. This results in four parameterised runs for each test within the class. Since some of the arguments to these shared parameters are mutable objects (i.e., the lists for `overrides`), they establish a shared state accessible to all the tests in this class.

Since any test may modify this globally shared state, there is the potential for latent test flakiness whenever test cases inadvertently depend on the existence of any mutable data created by a previously run test. As in the example furnished by Figure 2, a generated *FIT* may exploit this insight by transplanting a single line of code from later in the test class into the existing *DWT* [3]. Here, *FIT* modifies `overrides` and, since the variable is shared amongst the other tests, this creates a discrepancy with the expected value, thereby causing the *DWT* to fail when the *FIT* is executed before it. In order to detect and take advantage of such instances of improper state-sharing, FITTER will take inspiration from previous approaches applied to Java programs [7]. This process highlights to the developer the potential problem of this particular shared variable, for which a fix, in this case, is to replace the list type with an immutable tuple.

3 EXPOSING TEST FLAKINESS WITH FITTER

Without modifying the code of either the existing tests or the program, FITTER attempts to generate a *FIT* that causes the *DWT* to execute in a different, and potentially flaky, fashion. It will construct a *FIT* from program statements similar to those used in the *DWT*, the rest of the tests, and the program, in a style similar to current work in the area of APR [11, 12] — but rather with the goal of revealing latent flakiness so as to highlight repairs.

Attempting to induce flaky behaviour, FITTER will perform three types of fitness evaluations upon a candidate *FIT* with respect to some targeted *DWT*, following an approach in the *generate-and-validate* style [12]. The first fitness evaluation considers the number of modifying operations performed by the *FIT* upon mutable objects that are also referenced by the *DWT*. A *FIT* that maximises this property is more likely to pollute the shared state, an action commonly associated with test-order-related flakiness [5]. We envision the further extension of this evaluation to also handle, for instance, files and web sockets. The second fitness evaluation considers how "close" assertions in the tests are to being evaluated differently (e.g., how close an assertion, that usually passes as *true*, is to being rendered as *false*), and the third how "close" the path through the code under test is to being different from the default. The latter of these two would be calculated from runs before and after introducing the candidate *FIT* before some targeted *DWT* in the test suite, the basis of these being the traditional branch distance functions used in search-based testing [14]. *FIT*s that maximise these two metrics ought to induce different behaviour during execution of the *DWT* and are more likely to produce a different outcome [8].

[3]Link to the source code line in the GitHub repository: https://bit.ly/2TK5DWt

In contrast to previous methods (e.g., [1, 2, 5, 7, 18]), FITTER automatically generates tests, using APR techniques, that act as witnesses to latent flakiness. While prior work highlighted the variability of test coverage when executing nondeterministic tests [6, 8, 13, 15, 17], FITTER will exploit these differences to generate a *FIT* that gives developers actionable insights. Although FITTER will be evaluated using Python, the technique is generalisable to other languages and environments, unlike previous work (e.g., [1, 2, 5]), which carry platform-dependencies by requiring access to the components in a run-time environment like the JVM heap. Although prior work showed that it is time-consuming to manifest test flakiness by repeatedly running tests [1, 18], FITTER will not be hampered by this issue since it normally runs small portions of the test suite containing the *DWT* and a *FIT*. Since it uses test outcomes and coverage information to drive a search-based repair-like process responsible for generating a *FIT*, it will not mislead developers by reporting false positives about flakiness. Since we propose that the evaluation of genetic material, including the code under test, is to be performed, the scope of repairs is not limited to program statements from existing tests (as with [15]), which may be insufficient for resolving flakiness. Rephrasing a well-known aphorism to "flake it 'till you make it", we argue that the technique proposed in this paper represents a novel and practical approach — designed with developer preferences in mind [9] — that will help to expose and fix latent flakiness with methods inspired by APR. Simulating test suite evolution by generating new tests from existing material will offer developers a way to address flakiness before it manifests.

REFERENCES

[1] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *ESEC/FSE 2015*.
[2] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *ICSE 2018*.
[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *ESEC/FSE 2019*.
[4] Facebook. 2016. Python in production engineering, https://code.fb.com/production-engineering/python-in-production-engineering.
[5] A. Gambi, J. Bell, and A. Zeller. 2018. Practical Test Dependency Detection. In *ICST 2018*.
[6] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. 2015. Making System User Interactive Tests Repeatable: When and What Should we Control?. In *ICSE 2015*.
[7] A. Gyori, A. Shi, F. Hariri, and D. Marinov. 2015. Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency. In *ISSTA 2015*.
[8] M. Hilton, J. Bell, and D. Marinov. 2018. A Large-Scale Study of Test Coverage Evolution. In *ASE 2018*.
[9] P. S. Kochhar, X. Xia, and D. Lo. 2019. Practitioners' Views on Good Software Testing Practices. In *ICSE-SEIP 2019*.
[10] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. IDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST 2019*.
[11] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *ICSE 2012*.
[12] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *ISSTA 2019*.
[13] P. Marinescu, P. Hosek, and C. Cadar. 2014. Covrig: A Framework for the Analysis of Code, Test, and Coverage Evolution in Real Software. In *ISSTA 2014*.
[14] P. McMinn. 2004. Search-Based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* (2004).
[15] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-dependent Flaky Tests. In *ESEC/FSE 2019*.
[16] StackOverflow. 2019. Developer survey results, https://insights.stackoverflow.com/survey/2019.
[17] H. Zhai, C. Casalnuovo, and P. Devanbu. 2019. Test Coverage in Python Programs. In *MSR 2019*.
[18] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA 2014*.