Do Automatic Test Generation Tools Generate Flaky Tests?

Martin Gruber*
BMW Group, University of Passau
Munich, Germany
martin.gr.gruber@bmw

Muhammad Firhard Roslan*
University of Sheffield
Sheffield, United Kingdom
mfroslan2@sheffield.ac.uk

Owain Parry
University of Sheffield
Sheffield, United Kingdom
oparry1@sheffield.ac.uk

Fabian Scharnböck
University of Passau
Passau, Germany
scharn05@ads.uni-passau.de

Phil McMinn
University of Sheffield
Sheffield, United Kingdom
p.mcminn@sheffield.ac.uk

Gordon Fraser
University of Passau
Passau, Germany
gordon.fraser@uni-passau.de

ABSTRACT

Non-deterministic test behavior, or flakiness, is common and dreaded among developers. Researchers have studied the issue and proposed approaches to mitigate it. However, the vast majority of previous work has only considered developer-written tests. The prevalence and nature of flaky tests produced by test generation tools remain largely unknown. We ask whether such tools also produce flaky tests and how these differ from developer-written ones. Furthermore, we evaluate mechanisms that suppress flaky test generation. We sample 6 356 projects written in Java or Python. For each project, we generate tests using EvoSuite (Java) and Pynguin (Python), and execute each test 200 times, looking for inconsistent outcomes. Our results show that flakiness is at least as common in generated tests as in developer-written tests. Nevertheless, existing flakiness suppression mechanisms implemented in EvoSuite are effective in alleviating this issue (71.7 % fewer flaky tests). Compared to developer-written flaky tests, the causes of generated flaky tests are distributed differently. Their non-deterministic behavior is more frequently caused by randomness, rather than by networking and concurrency. Using flakiness suppression, the remaining flaky tests differ significantly from any flakiness previously reported, where most are attributable to runtime optimizations and EvoSuite-internal resource thresholds. These insights, with the accompanying dataset, can help maintainers to improve test generation tools, give recommendations for developers using these tools, and serve as a foundation for future research in test flakiness or test generation.

CCS CONCEPTS

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

KEYWORDS

Test Generation, Flaky Tests, Empirical Study

*Both authors contributed equally to this research.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3608138

ACM Reference Format:
Martin Gruber, Muhammad Firhard Roslan, Owain Parry, Fabian Scharnböck, Phil McMinn, and Gordon Fraser. 2024. Do Automatic Test Generation Tools Generate Flaky Tests?. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3608138

1 INTRODUCTION

A *flaky test* is a test case that produces inconsistent results, meaning that the same test can pass or fail for no apparent reason, even when the system being tested has not changed [51]. They are a major problem for software developers because they limit the efficiency of testing, complicate continuous integration, and reduce productivity [19, 39, 48]. The negative effects of flaky tests are ubiquitous, experienced by large companies such as Google, Microsoft, and Facebook, as well as the developers of smaller open-source projects [19, 36, 45, 47]. Indeed, recent surveys found that a majority of developers observe flaky tests on at least a monthly basis [29, 52]. As well as being a burden on developers, flaky tests are also a persistent problem in research, limiting the deployment of several state-of-the-art techniques for test selection and prioritization [45, 54, 68].

Increasing research interest in the area of flaky tests has produced a range of empirical studies regarding the causes, origins, and impacts of developer-written flaky tests [21, 37, 44, 64]. However, far less attention has been paid to flaky tests produced by automatic test generation tools [53, 60]. This research gap is problematic for several reasons. Firstly, there is minimal guidance for developers regarding the sorts of flaky tests they might expect to receive from test generation tools and more crucially how to avoid them. This threatens to detract from the positive benefits of such tools on the software development lifecycle as previously established [61]. Similarly, the maintainers of test generation tools have only limited information on the prevalence of automatically generated flaky tests and what causes them. These insights are crucial for maintainers to prevent their tools from producing flaky tests. Furthermore, researchers in the field of flaky tests would benefit from an analysis of how the root causes of developer-written flaky tests compare to those that are automatically generated. Such an investigation would inform researchers on whether generated flaky tests are representative of their developer-written counterparts. This would be useful for augmenting existing datasets of developer-written flaky tests with generated flaky tests [35].

In this study, we used the popular search-based test generation tools EvoSuite [25] and Pynguin [42] to generate test suites for 1 902

Java projects and 4 454 Python projects respectively. We repeatedly executed both the developer-written and automatically generated test suites of all the projects, consisting of nearly a million individual test cases, 200 times each to detect flaky tests. We compared the prevalence of flakiness between both types of test suites and went one step further by comparing root causes, following our manual analysis on a random sample of 481 non-order-dependent flaky tests. Furthermore, we performed the first scientific evaluation on the effectiveness of EvoSuite's built-in flaky test suppression feature. Chiefly among our findings, we found that flaky tests are at least as common in generated tests as they are in developer-written tests, that EvoSuite's flaky test suppression feature can reduce the number of generated flaky tests by 71.7 %, and that the distribution of the root causes of generated flaky tests differs to that of developer-written tests.

The main contributions of this study are as follows:

Contribution 1: Empirical study: Our empirical study involving 6 356 open-source projects is the largest among all previous studies on both flaky tests and search-based test generation. Our study is also the first to analyze the root causes of automatically generated flaky tests. See Section 3 for more information.

Contribution 2: Recommendations: The results of our study have important implications for software developers, maintainers of test generation tools, and researchers in the area of flaky tests. From these, we are able to offer insights and recommendations that are actionable by these stakeholders. See Sections 4 and 5 for more information.

Contribution 3: Dataset: The dataset we collected for this study is the first publicly available dataset of flaky tests that contains automatically generated flaky tests and features a large manually annotated sample. See our replication package for more information [11].

2 BACKGROUND

2.1 Flaky Tests

Luo et al. [44] performed one of the earliest empirical studies of test flakiness. They categorized the cause of the flaky tests repaired by developers in 201 commits across 51 open-source projects using the following ten categories:

1. Async. Wait. Test makes an asynchronous call but does not properly wait for it to finish, leading to intermittent failures.

2. Concurrency. Test spawns multiple threads that behave in an unsafe or unanticipated manner, such as a race condition.

3. Floating Point. Test uses floating points and is flaky due to unexpected results such as non-associative addition.

4. Input/Output (I/O). Test uses the filesystem and is flaky due to intermittent issues such as storage space limitations.

5. Network. Test depends on the availability of a network and is flaky when the network is unavailable or busy.

6. Order Dependency. Test depends on a shared value or resource that is modified by other test cases as a side effect.

7. Randomness. Test involves random number generators and is flaky due to not setting seeds, for example.

8. Resource Leak. Test does not release acquired resources (e.g. database connection) inducing flaky failures for itself or for other tests that require the same resources.

9. Time. Test relies on measurements of date and/or time. Flakiness is caused by, for instance, discrepancies in precision and representation of time across libraries and platforms.

10. Unordered Collection. Test assumes a deterministic iteration order for an unordered collection-type object, such as a `set`, leading to intermittent failures.

Eck et al. [21] asked Mozilla developers to categorize the causes of 200 flaky tests they had previously repaired. The developers used the categories introduced by Luo et al. but with the option to create new categories if needed. Following this, Eck et al. identified the following additional four categories:

11. Too Restrictive Range. Test includes a range-based assertion that excludes a portion of the valid output values.

12. Test Case Timeout. Test intermittently exceeds a pre-defined upper limit on its execution time.

13. Platform Dependency. Test outcome varies across the project's target platforms.

14. Test Suite Timeout. Test intermittently exceeds a pre-defined upper limit on the execution time of the test suite.

While studying flakiness in Python tests, Gruber et al. [32] identified another root cause:

15. Infrastructure. Test fails intermittently due to issues outside the project code, but inside the execution environment (the container or the local host), for example, permission errors or lack of disk space.

2.2 Automatic Test Generation

EvoSuite is an automatic test generation tool for Java projects [25, 26]. It uses a search-based approach to generate test suites that cover as much of the code under test as possible. The tool is based on evolutionary algorithms, which means that it uses principles from biological evolution to search for test cases [46]. The search is guided by a fitness function that can be configured to optimize test suite generation for high line, branch, or mutation coverage. The tool applies techniques from mutation testing to minimize the number of assertions in the generated tests [33]. This ensures that they are more easily understandable to developers and not overly brittle. A large-scale empirical study demonstrated that EvoSuite can achieve an average of 71% branch coverage per class [27]. EvoSuite also provides ways to suppress "unstable" (flaky) tests, addressing this issue both during the evolutionary search process and after its completion. EvoSuite detects unstable tests by controlling the environmental dependencies using bytecode instrumentation, resetting the state of static variables before executing each test, using mocks to replace non-deterministic calls, and compiling and executing the generated tests, removing failing tests [14].

Pynguin is an automatic test generation tool for Python projects [41–43]. Target languages aside, Pynguin and EvoSuite share several similarities. Both tools use a search-based approach to generate test cases and both apply mutation testing to generate assertions. Pynguin faces the additional challenge that Python is a dynamically typed language, meaning that generating test inputs of the appropriate type is much harder. Therefore, Pynguin relies on existing type hints in function and class definitions in the code under test. From these, Pynguin can apply type inference to attempt to determine the types of variables without hints. A previous empirical evaluation of

Pynguin found that it was able to achieve a mean branch coverage of 71.6 % on 163 Python modules from 20 open-source projects [43].

3 METHODOLOGY

With our study, we aim to answer the following research questions:

RQ1 (Prevalence): How prevalent is flakiness in tests that were generated without flakiness suppression mechanisms?

RQ2 (Flakiness Suppression): How many flaky tests can EvoSuite's flakiness suppression mechanism prevent?

RQ3 (Root Causes): How do the root causes of generated flaky tests differ from those of developer-written tests?

Fig. 1 depicts an overview of our study setup.

3.1 Project Sampling

To collect subjects for our empirical study, we randomly sampled open-source projects written in Java and Python. These are two of the most popular programming languages, which have also been the main targets for both test flakiness and test generation research.

3.1.1 Java. To collect Java projects, we used the index of the Maven Central Repository [4], one of the official software repositories for Java. The index is updated weekly for newly added projects or patches. It consists of roughly 520 000 unique packages (as of 2022–10–26). We iterated over the entire index and each project's Project Object Model (POM) to fetch the URL to the project repository. We limited our project sampling to only include projects whose source code is available on GitHub and which use Maven as a build tool. Since the project's POM in the Maven Central Repository does not include details of the build automation tool it is using—which means that the index includes projects built from Gradle or Ant—we crawled through the GitHub URLs to filter for projects that include a `pom.xml` file in the root of their repository. This is to confirm that the project is using Maven as its build automation tool. In total, we found 38 841 Maven projects that include a link to the project repository on GitHub. While some projects may lack developer-written tests, we did not exclude them during this crawling process (but we did so later, during the test outcome analysis). We decided against sampling projects from existing flakiness databases, such as IDoFT [35], because we did not want to limit our study to projects that already contain developer-written flaky tests.

3.1.2 Python. To collect Python projects, we used the dataset from Gruber et al. [32], who studied flakiness in Python. It consists of 22 352 projects that were randomly sampled from the Python Package Index (PyPI) [10], the official third-party software repository of Python. Each project contains at least one test that could be executed using pytest [8] and its source code is available on GitHub. Unlike IDoFT [35], these projects do not all contain flaky tests: The original study found 7 571 flaky tests among 1 006 projects.

3.2 Test Generation

To generate tests for the sampled projects, we use state-of-the-art test generation tools for the respective language. For Java, we use EvoSuite [25], an automated test generation tool that utilizes metaheuristic techniques to generate JUnit test suites. For Python, we use Pynguin [41–43], a test generation tool that produces unit tests for Python programs. The tests are generated from scratch, not relying on any existing developer-written tests as input [56].

Table 1: Updated parameters to deactivate flakiness suppression mechanisms (EvoSuite$_{FSOff}$)

Parameter	Description	New Value
Test Scaffolding	Generate separate scaffolding file to execute the tests	false
No Runtime Dependency	Avoid runtime dependencies in JUnit test	true
JUnit Check	Compile and run the resulting JUnit test suite	false
Sandbox	Executing tests in an independent testing environment	false
Virtual FS	Using virtual file system for all File I/O operations	false
Virtual Net	Using virtual network for all network communications	false
Replace Calls	Replacing non-deterministic calls	false
Replace System In	Replacing the InputStream mechanism with a mock	false
Replace GUI	Replacing the GUI calls with a mock	false
Reset Static Fields	Call static constructors only after static field was changed	false
Reset Static Field Gets	Call static constructors only after static field was read	false
Reset Static Final Fields	Remove the static modified in target fields	false

3.2.1 Java. We use the latest release of EvoSuite at the time (v1.2.0) to generate tests for each Java project. Since EvoSuite applies multiple techniques to avoid creating flaky tests, and we want to measure the impact of these flakiness suppression mechanisms, we apply EvoSuite under two configurations: *with* and *without* Flakiness Suppression, which we refer to as EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$. The flakiness suppression parameters are turned on by default, which means that for EvoSuite$_{FSOn}$, we do not change any of the EvoSuite parameters. To generate tests without the flakiness suppression mechanisms (EvoSuite$_{FSOff}$), we update several parameters of EvoSuite that we extracted from previous studies [14, 24, 26], and which we confirmed via in-depth discussions with one of the EvoSuite maintainers on our team. Table 1 shows the parameters that we changed to deactivate EvoSuite's flakiness suppression. These parameters consist of actions carried out during the test generation process to mitigate non-determinism. They address factors related to managing environmental dependencies, establishing a virtual file system, mocking non-deterministic output such as Random [2] and Calendar [1] classes, and resetting the state of static and final fields to avoid creating dependencies on other tests. For each of the Java projects, both EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$ generate tests for every testable class in the system under test (SUT). A class is considered testable if it has at least one public method. EvoSuite aims to generate a test suite that covers all public methods. We set the search budget for generating tests to two minutes per class for both EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$, which is what previous tool competitions have used [58, 65].

3.2.2 Python. To generate tests for the Python projects, we use version 0.27.0 of Pynguin, the latest release at the time (2022–11-14). Pynguin operates on module-level, and we apply it to each module contained in each of our sample projects. Like Lukasczyk et al. [43], we use a maximum search budget of ten minutes. Unlike EvoSuite, Pynguin does not offer optional parameters for flakiness

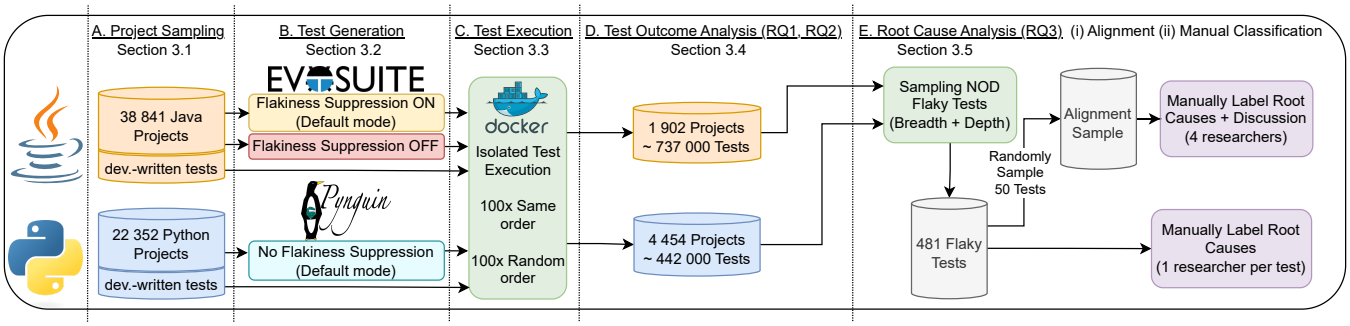


Figure 1: Study setup

suppression. Instead, it applies a re-execute-once strategy to "filter out trivially flaky assertions, e.g., strings that include memory locations" [7]: After generating a test that contains passing assertions, the test is executed again. Any assertion that does not hold in this execution is excluded, as it was made on an apparently flaky value. This behavior is inspired by EvoSuite's `JUnit Check`, however, since it is not optional and Pynguin has no further flakiness-relevant parameter, we execute Pynguin using only one configuration: the default settings.

Both EvoSuite and Pynguin generate tests non-deterministically, meaning that they generate different test suites every time the tool is used. Most of the previous studies investigating automatic test generation generated more than one test suite per project to take the random nature of the evolutionary search [42, 60] into account. Unlike these, we generate only one test suite per class/module, since our study does not draw any conclusions by comparing individual projects or components. Instead, we accommodate for the randomness in the test generation by sampling a large corpus of projects, which also contributes to the generalizability of our findings.

3.3 Test Execution

To detect flaky tests, we execute all generated tests and all developer-written tests 100 times in the same order and 100 times in random orders. This procedure follows other studies [32, 38] and allows us to distinguish order-dependent (OD) from non-order-dependent (NOD) flaky tests. The test executions are conducted either directly through or inspired by FlaPy [31], a tool that allows researchers to mine flaky tests from a given set of projects by repeatedly executing their test suites. FlaPy ensures a fresh and isolated environment for each test execution and handles dependency installation. Furthermore, it splits the runs into iterations, where each iteration is executed in a separate Docker container, which helps to avoid timeouts and detect environment issues, such as infrastructure flakiness [32]. In our case, we split the 200 runs into at least five iterations per project and test type. Generated and developer-written tests are not executed together, but in separate iterations to avoid side-effects.

To install third-party dependencies of the project under evaluation, we use language-specific pipelines: For Java this can easily be accomplished by using `'mvn dependency:copy-dependencies'` to make a copy of all the dependencies from the repository on our local machine. We then update the environment variables to include all the dependencies that the project needs when executing the tests. For Python this process is more complicated since the general landscape of build systems is more heterogeneous. As we cannot rely on a standardized solution, we use FlaPy's built-in dependency installation heuristic which searches for `requirements.txt` (or similarly named) files and runs them against `pip`. To execute the projects' test suites we use the JUnit Runner [3] for Java and pytest [8] for Python. When conducting test executions in random orders, we shuffle the tests on class-level, which randomly sorts first the classes and then the tests within each class. For Python, this can easily be accomplished using pytest's random-order plugin [9]. For Java, we had to create a custom test runner since Maven's Surefire plugin [5] currently does not support this form of shuffling.

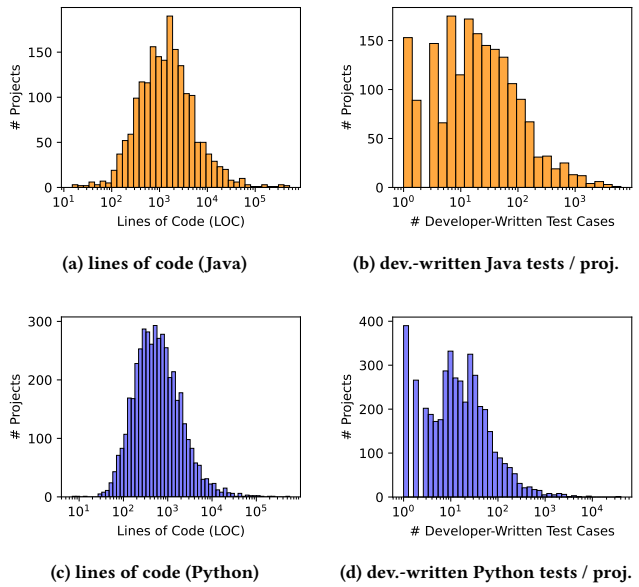
Table 2: Size of our dataset

Language	Test Gen. Framework	# Projects	# Tests (*FS* = Flakiness Suppression)		
			Developer -written	Generated Without *FS*	Generated With *FS*
Java	EvoSuite	1 902	163 305	264 000	310 193
Python	Pynguin	4 454	303 711	138 627	

3.4 Test Outcome Analysis

Table 2 depicts the number of projects for which we were able to successfully execute the developer-written tests, and successfully generate tests using EvoSuite or Pynguin. We consider the execution of the developer-written test suite to be successful if at least one test case was executed without producing an *Error* or *Skip* outcome. We consider the test generation to be successful if at least one executable test was generated. Since we use the same generic setup processes for all projects of the same language, we were not able to successfully execute the developer-written test suite and the test generation for each sampled project. Reasons for erroring test executions or test generation include project-specific requirements that go beyond standard third-party dependencies, such as setting global variables or installing system software.

To assert if we still derived a sufficiently large and diverse set of projects, we inspect two quantitative metrics: the lines of source code (SLOC) of the projects measured via CLOC [16], and the number of developer-written tests they possess (Fig. 2). To assess if we applied the test generation tools properly, we look at the coverage



(a) lines of code (Java) (b) dev.-written Java tests / proj.

(c) lines of code (Python) (d) dev.-written Python tests / proj.

Figure 2: Dataset statistics

that the generated tests achieved and compare it with the coverage reported by previous studies applying these tools.

3.4.1 Java. On average (mean) the Java projects possess 4 948 lines of source code (median 1 395). Only 1.5 % of all projects contain less than 100 SLOC, whereas the largest project (`cosmos-sdk-java`[1]) has more than 500 000 lines of Java code. The total number of SLOC is around 9.4 million for all projects combined. Fig. 2a shows the histogram of the SLOC distribution for the Java projects. Fig. 2b shows the number of developer-written test cases per Java project. Multiple parametrizations of the same test case are treated as separate test cases, following a previous study [32]. In total, the projects possess 163 305 developer-written tests and each project contains between 1 and 6 315 test cases. The median number of tests per project is 18, and the mean is 85.9. As these figures about SLOC and test cases show, we have indeed derived a large and diverse sample of Java projects. Both EvoSuite$_{FSOn}$ and EvoSuite$_{FSOff}$ generated test suites with high branch- (over 81 % mean) and line- (over 84 % mean) coverage (Fig. 3). This is similar to the reported code coverage of previous studies on EvoSuite [27, 28].

3.4.2 Python. Fig. 2c depicts the size of the 4 454 Python projects in terms of SLOC. The average (mean) project has 1 755 SLOC (median 549.5). The largest project (`kuber`[2]) has more than 500 000 lines of source code and only 5.2 % of projects contain less than 100 SLOC. Combined, the projects feature 7.8 million lines of Python code. Fig. 2d shows the size of the Python projects in terms of the number of developer-written tests they contain. Like for the Java projects, the mean number of test cases per project (68.2) is substantially greater than the median (14), which is caused by a small number of very large projects. In total, the Python projects contain 303 711 developer-written test cases. After inspecting the projects both

[1]https://github.com/cloverzrg/cosmos-sdk-java
[2]https://github.com/sernst/kuber

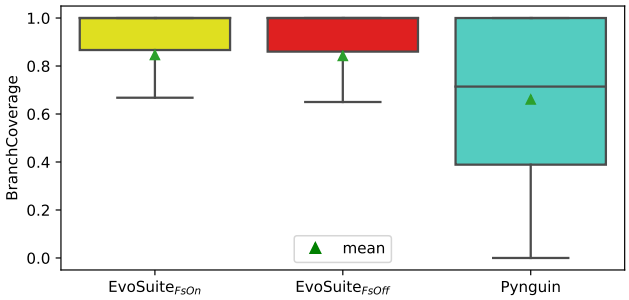


Figure 3: Branch coverage of generated tests

in terms of number of SLOC and their number of test cases, we find no obvious bias towards overly small or large projects and conclude that we have derived a large and diverse sample of Python projects. The Python tests generated by Pynguin yielded a mean branch coverage of 66.0 % (Fig. 3) with a standard deviation of 31.7. This performance is very similar to the one reached by Lukasczyk et al. [43], the creators of Pynguin, who achieved a mean branch coverage of 71.6 % with a standard deviation of 30.5.

3.4.3 RQ1 (Prevalence). To study the prevalence of flakiness in generated tests, we compare the number of flaky tests that were created by the test generation tools—without using flakiness suppression—to the number of flaky tests found in the developer-written tests of the respective language. We regard a test as flaky if it yielded at least one passed and one failed or errored outcome [17, 32]. Tests that switch between failing and erroring verdicts are therefore not considered as flaky, since both lead to a build failure and the test therefore does not contribute to the typical developer experience caused by flakiness (sporadically failing builds). Furthermore, we look at the ratio between order-dependent flaky tests (which only show flaky behavior when run in random orders) and non-order-dependent flaky tests (which also show flaky behavior when run in the same order). Lastly, we also compare the projects containing at least one developer-written or generated flaky test to investigate if generated and developer-written flakiness tends to appear in the same projects.

3.4.4 RQ2 (Flakiness Suppression). To assess the effectiveness of EvoSuite's flakiness suppression, we compare the number of flaky tests generated by EvoSuite$_{FSOn}$ to those generated by EvoSuite$_{FSOff}$, and to developer-written tests. For each Java project and each of the three test types, we compute the ratio of flaky to non-flaky tests and use a Wilcoxon signed-rank test [67]—which is a commonly used, non-parametric paired difference test—to check for statistically relevant differences. We refrain from using a parametric test, as we found our data to be not normally distributed according to a Shapiro-Wilk test [62]. Since Pynguin does not offer any optional flakiness suppression mechanisms, we limit our comparison to EvoSuite.

3.5 Root Cause Analysis

In our last research question, we investigate the similarity of developer-written and generated flaky tests regarding their root causes,

which is seen as a core property of flakiness [44, 51]. Like other studies [32, 44], we categorize the flaky tests' root causes by labeling them manually along established categories. Namely, we use the amalgamation of root causes collected by Parry et al. in 2021 [51] as our initial set of pre-defined categories (items 1. to 14. in Section 2.1). Since we can automatically detect order-dependency (OD) through test executions in random orders, we only consider non-order-dependent (NOD) flaky tests for this step.

3.5.1 Sampling. As we found a total of 1 740 NOD flaky tests, we have to take a representative sample to keep the labeling feasible. To avoid creating a bias towards projects with only a few flaky tests (e.g., by randomly selecting projects), or tests from only a few large projects (e.g., by randomly selecting flaky tests), we combine two sampling strategies: First, we randomly select one NOD flaky test from each affected project, regardless of the test type (generated or developer-written), resulting in the *breadth sample*. Second, we randomly choose 21 Java and 9 Python projects and sample all their flaky tests (*depth sample*). These projects are evenly distributed regarding the type—or combination—of flaky tests they contain (Java developer-written, EvoSuite$_{FSOn}$, EvoSuite$_{FSOff}$, Python developer-written, Pynguin). Using this technique, we sampled a total of 481 flaky tests (340 Java, 141 Python): 329 from the breadth sample, 122 from the depth sample, and 30 selected by both strategies.

3.5.2 Labeling. The manual labeling itself is carried out by four of the authors using the project code, the test code, as well as the test failures (stack trace and error message). To create a common understanding about what constitutes a certain root cause and to assess if existing root cause categories are applicable to generated flaky tests, we precede the actual labeling with an alignment step: We randomly choose 50 flaky tests from our sample, which are then classified by all four researchers. According to Fleiss' Kappa [23], the four labeling authors have reached an inter-rater reliability of 0.41, which is considered 'good' according to Regier et al. [55]. For cases in which the authors disagree, discussions are held, which have resulted in the following adjustments to the set of root causes:

- Broaden the category *unordered collection* to also include *unspecified behavior* in general.
- Broaden the category *resource leak* to also include *resource unavailability*.
- Add category *performance*, which describes tests that fail intermittently due to varying durations of (sequential) processes (example: Fig. 5).
- Add category *non-idempotent-outcome* (NIO), which covers self-polluting and self-state-setting tests. This was described by Wei et al. [66] shortly after the literature survey on which we based our root cause categories [51].

After finishing the alignment, the remaining flaky tests in the sample are labeled each by one of the four researchers.

3.5.3 RQ3 (Root Causes). We use the labeled root causes to make three main comparisons: First, the root causes we found in developer-written tests against those found by previous studies. Second, the root causes of flaky tests generated without flakiness suppression (EvoSuite$_{FSOff}$, Pynguin) against those of developer-written tests of the respective language. Third, the root causes of tests generated with and without flakiness suppression (EvoSuite$_{FSOff}$ vs. EvoSuite$_{FSOn}$).

3.6 Threats to Validity

3.6.1 External Validity. To sample projects for our study, we relied on the Maven Central Repository [4] and PyPI [10], which are the largest official software repositories of Java and Python. However, we had to make certain assumptions to keep our setup feasible: For Java, we only considered projects using Maven and we excluded projects using JUnit 3 due to compatibility issues with more recent versions. While these design decisions might potentially influence our results, we tried to mitigate this threat by assuming the usage of the predominantly used build automation and testing technologies (Maven and JUnit), which also other studies on test flakiness rely on [13, 38, 63]. For Python, we used an existing dataset of Python projects [32], which was also used by other researchers to evaluate flakiness detection and debugging techniques [12, 30, 57, 66]. Nevertheless, we inherit any potentially existing bias in this dataset.

3.6.2 Construct Validity. To detect test flakiness, we executed each test 100 times in a fixed order and 100 times in shuffled orders. However, some flaky tests have very low failure rates, which might have caused us to underestimate the number of flaky tests in our dataset. Another potential threat to the construct validity of our study is the search budget used for test generation. We gave two minutes per Java class for EvoSuite and ten minutes per Python module for Pynguin. However, allowing more time might have yielded different results. Choosing a meaningful search budget is a non-trivial issue, especially when setting up experiments that include thousands of projects with various different sizes [15]. To achieve a balance between feasibility, resembling a practical use case, and giving sufficient resources to the tools, we chose our search budgets according to the most commonly used configurations in tool competitions [58, 65] or evaluations by the maintainer [43]. We also measured the coverage of the generated tests and found that they yielded a high branch coverage, which indicates that our search budgets were sufficient.

3.6.3 Internal Validity. As we found almost 1 800 NOD flaky tests, we had to take a sample before manually labeling their root cause, which might pose a potential threat to the validity of our findings. To avoid favoring overly large or small projects, we applied a two-fold sampling strategy (see Section 3.5.1). Each flaky test was then manually labeled by one of four authors. This might pose a potential threat, as the authors have different backgrounds and experiences when it comes to root-causing flaky tests. To mitigate this issue, we created an alignment sample of 50 flaky tests that were labeled by all four researchers, and we held discussions about cases in which we disagreed. In our alignment sample, we reached a 'good' inter-rater reliability, meaning that we were aligned in most of the verdicts given even before starting the alignment. Furthermore, the root causes we found for developer-written flaky tests match previous studies [21, 44], which increases our confidence in the validity of our other findings.

Table 3: Number of flaky tests found in developer-written and automatically generated tests

Language	Test Type	NOD		OD		Flaky (NOD + OD)		All	
		# Tests	# Projects	# Tests	# Projects	# Tests	# Projects	# Tests	# Projects
Java	Developer-Written	698 (0.43 %)	105 (5.52 %)	830 (0.51 %)	104 (5.46 %)	1 528 (0.94 %)	161 (8.46 %)	163 305	1 902
	EvoSuite$_{FSOn}$	175 (0.06 %)	43 (2.26 %)	1 110 (0.35 %)	109 (5.73 %)	1 285 (0.41 %)	133 (6.99 %)	310 193	
	EvoSuite$_{FSOff}$	597 (0.22 %)	111 (5.84 %)	3 235 (1.23 %)	163 (8.57 %)	3 832 (1.45 %)	228 (11.9 %)	264 000	
Python	Developer-Written	182 (0.06 %)	88 (1.98 %)	1 728 (0.57 %)	270 (6.06 %)	1 910 (0.63 %)	341 (7.65 %)	303 711	4 454
	Pynguin	88 (0.06 %)	49 (1.10 %)	925 (0.67 %)	183 (4.11 %)	1 013 (0.73 %)	224 (5.03 %)	138 627	

4 RESULTS

4.1 RQ1: Prevalence

Table 3 depicts the number of flaky tests we discovered for each language and configuration. Overall we executed almost 1.2 million tests and discovered 9 568 flaky tests, roughly two-thirds of them generated ones. For developer-written tests, we found roughly 0.5 % to 1 % of all tests to be flaky, which is similar to previous studies on flakiness in Java [38] and Python [32]. Like them, we also found the ratio between OD and NOD flaky tests to be almost even for Java projects, whereas it strongly tilts towards order-dependency for Python projects.

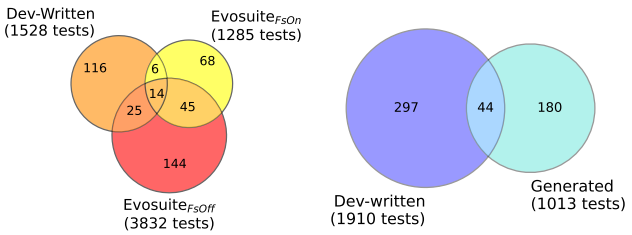
Looking at the flaky tests generated by EvoSuite$_{FSOff}$ (3 832) and Pynguin (1 013), we see that for both languages/tools, flakiness is more prevalent in generated than in developer-written tests, relative to the total number of generated/developer-written tests. In the case of Java, we even see an increase of 54 % (0.94 % to 1.45 %). When distinguishing between order- and non-order-dependent flakiness, we see a strong tendency towards OD flaky tests (91 % of flaky Pynguin tests are OD, 84 % of flaky EvoSuite$_{FSOff}$ tests are OD).

To check if generated flaky tests tend to appear more frequently in projects that already contain developer-written flaky tests, we looked at the sets of projects containing at least one flaky test. We found 224 Python projects containing generated flaky tests and 341 projects having developer-written ones. For Java, EvoSuite$_{FSOff}$ produced flaky tests for 228 projects, while 161 projects contained developer-written flaky tests. Fig. 4 depicts the overlap between these sets, which is notably small: Only 17.1 % of Java and 19.6 % of Python projects that contain generated flaky tests also contain developer-written flaky tests.

Summary *(RQ1: Prevalence)* For both Java and Python projects, flakiness is at least as common in generated tests as in developer-written tests. However, it does not appear in the same projects. Similar to developer-written tests in Python (but unlike Java tests), the ratio between order-dependent and non-order-dependent flaky tests is leaning strongly towards order-dependency for generated tests.

4.2 RQ2: Flakiness Suppression

The second row in Table 3 (Java, EvoSuite$_{FSOn}$) depicts the amount of flakiness we found among tests generated while using flakiness suppression. We observe a significant (p-value of Wilcoxon test < 0.001) reduction in flakiness of 71.7 % (1.45 % to 0.41 %) compared to EvoSuite$_{FSOff}$ and 56.4 % compared to the developer-written tests. Among the remaining flaky tests, order-dependency is again far more common than non-order-dependent causes (86.4 % of EvoSuite$_{FSOn}$ flaky tests are OD). Looking at projects containing flaky tests (Fig. 4a), we see only a minor overlap between EvoSuite$_{FSOn}$ (133 projects) and the developer-written tests (161 projects), as we found for EvoSuite$_{FSOff}$ (228 projects). When comparing the two EvoSuite configurations, we are surprised to also find only a moderate overlap. To investigate this observation more deeply, we look at the flaky tests' root causes in Section 4.3.



(a) Java **(b) Python**

Figure 4: Projects containing flaky tests

Summary *(RQ2: Flakiness Suppression)* EvoSuite's flakiness suppression mechanism is effective: It reduced the number of flaky tests by 71.7 %, which is considerably lower than the relative number of developer-written flaky tests (56.4 % fewer flaky tests). The ratio of NOD and OD flaky tests remains strongly leaning towards OD.

4.3 RQ3: Root Causes

Table 4 depicts the distribution of flakiness root causes that we identified in our sample via manual labeling (Section 3.5).

For Java projects, we found *asynchronous waiting* to be a major cause (21.2 %) for flakiness in developer-written tests, which corroborates previous studies [21, 44]. However, we also found many flaky tests to be caused by brittle assumptions about the *Performance* (i.e., duration) of sequential processes (19.4 %), a root cause that has not previously been described. Fig. 5 shows an example of such a test. The assertion on line 113 is flaky as it assumes that the execution time (line 106) is within a certain range, which is not guaranteed. For Python projects, the main causes for developer-written flakiness are networking (30.1 %) and randomness (17.2 %), which was also found by the study from which we sampled our projects [32].

[3] https://github.com/krka/mockachino/tree/9bcdda05

Table 4: Root causes for NOD flaky tests. Cells: number of tests (number of projects)

Language	Java			Python	
Test Type	Dev.-Written	EvoSuite$_{FSOn}$	EvoSuite$_{FSOff}$	Dev.-Written	Pynguin
Total	170 (106)	57 (42)	113 (102)	93 (88)	48 (47)
Async Wait	36 (23)	3 (2)	5 (4)	11 (7)	1 (1)
Concurrency	15 (14)	0 (0)	5 (3)	3 (3)	0 (0)
Floating Point	0 (0)	0 (0)	0 (0)	2 (2)	0 (0)
I/O	9 (9)	0 (0)	0 (0)	6 (6)	3 (3)
Network	36 (9)	1 (1)	13 (13)	28 (27)	6 (6)
NIO	0 (0)	0 (0)	0 (0)	3 (3)	5 (5)
OTHER	0 (0)	34 (22)	0 (0)	0 (0)	3 (3)
Performance	33 (11)	0 (0)	7 (7)	4 (4)	4 (4)
Randomness	13 (12)	2 (2)	23 (22)	16 (16)	11 (11)
Resource Leak / Resource Unavailability	7 (7)	2 (2)	7 (6)	0 (0)	0 (0)
Test Case Timeout	2 (2)	4 (4)	23 (18)	0 (0)	0 (0)
Time	6 (6)	2 (1)	12 (12)	4 (4)	0 (0)
Too Restrictive Range	2 (2)	0 (0)	0 (0)	7 (7)	0 (0)
UNKNOWN	8 (8)	2 (2)	4 (4)	7 (7)	6 (5)
Unordered Collection / Unspecified Behavior	3 (3)	7 (6)	14 (13)	2 (2)	9 (9)

```
73 @Test
74 public void testTimeoutFailExactly() {
75     final List mock = Mockachino.mock(ArrayList.class);
76     mock.size();
77     mock.size();
78     runTimeoutTest(Mockachino.verifyExactly(2),200,220,200,500,
         mock,() -> mock.size());
79 }
   // ... omitted ...

98 private void runTimeoutTest(VerifyRangeStart type,int min,int
       max,int waitTme,int timeout,List mock,Runnable runnable) {
99
100    long t0 = System.currentTimeMillis();
101    Executors.newSingleThreadScheduledExecutor().schedule(
         runnable,waitTime,TimeUnit.MILLISECONDS);
102    long t1 = System.currentTimeMillis();
103    long margin = t1 - t0;
       // ... omitted ...
106    type.withTimeout(timeout).on(mock).size();
       // ... omitted ...
111    long t2 = System.currentTimeMillis();
112    long time = t2 - t1;
113    assertTrue(time + " expected at most " + max, time <= max +
         margin);
114 }

   <error message="273 expected at most 220" type="junit.framework
       .AssertionFailedError">
```

Figure 5: Developer-written flaky test with root cause *Performance* (project `krka-mockachino`[3])

```
1 @Test(timeout = 4000)
2 public void test00()  throws Throwable  {
3     RandomJava randomJava0 = new RandomJava();
4     randomJava0.gaussian((double) 1057);
5     double double0 = randomJava0.gaussian();
6     assertEquals(0.8241080392646101, double0, 0.01);
7 }

  Expected:<0.8241080392646101> but was:<-0.08836772391958745>
```

Figure 6: Randomness-related flaky test generated by EvoSuite$_{FSOff}$ (project `mitchelltech5-jmatharray`)

For flaky tests that were generated without using flakiness suppression (EvoSuite$_{FSOff}$ and Pynguin), their root causes fit our predefined categories (only few *OTHER* cases), however, the distribution differs: Generated flaky tests tend to be more commonly caused by *Randomness* (~20 %) and *Unspecified Behavior* (EvoSuite$_{FSOff}$: 12.4 %, Pynguin: 18.7 %). Fig. 6 is an example of a test generated by EvoSuite$_{FSOff}$, which is flaky due to randomness. The test makes an assertion against the value of a random variable that it sampled from a Gaussian distribution using a Box-Muller transform. Fig. 7 shows an example of a randomness-related flaky test that was generated by Pynguin for the `usolitaire` project, which is a terminal solitaire application. The `game_0` object shuffles the deck of cards randomly which causes the `move_tableau_pile(int_0, bool_0)` method to arbitrarily pass or fail raising an `InvalidMove` error. Fig. 8 shows another test that was generated by Pynguin, which is flaky due to the non-deterministic order within a `frozensets` in Python.

For EvoSuite$_{FSOff}$ we also found *Test Case Timeouts* happening frequently (20.4 %), which are caused by the 4000ms timeout EvoSuite sets for each test it generates. Python tests generated by Pynguin, on the other hand, do not exhibit such issues, as it does not set a test case timeout. One type of flakiness that was generated by Pynguin, but not by EvoSuite, is non-idempotent-outcome (NIO) (10.4 %). Fig. 9 shows a NIO test produced by Pynguin from project `bl` (BlackEarth core library). The statement in line 7 tries to create a file, which succeeds for the first run in each iteration (i.e. container), but raises a `FileExistsError` for every further run. During the test generation, the `config_0.write(str_0)` operation was most likely executed multiple times, which caused Pynguin to assume that the exception is meant to be thrown, so it created an assertion based on it (line 6). The test therefore has an inverted failure pattern, where its first execution fails and the following executions pass. EvoSuite$_{FSOn}$ doesn't experience such issues, since it uses a virtual file system, however, we also did not find such cases for EvoSuite$_{FSOff}$, where we deactivated this mechanism.

When looking at tests generated while using flakiness suppression (EvoSuite$_{FSOn}$), the picture changes drastically: On one hand, the flakiness suppression vastly reduced the amount of flakiness caused by any known root cause. On the other hand, we found that the majority (59.6 %) of the remaining flaky tests do not fit any known root cause category (*OTHER*). We inspected these cases in greater detail and found them to be attributable to two causes: *Verifying Expected Exceptions* (18/34) and *StackOverflowErrors* (16/34).

```
def test_case_33():
    int_0 = 0
    game_0 = module_0.Game()
    # omitted
    bool_0 = True
    with pytest.raises(module_0.InvalidMove):
        game_0.move_tableau_pile(bool_0, int_0)
    var_0 = game_0.move_tableau_pile(int_0, bool_0)

E usolitaire.game.InvalidMove
```

Figure 7: Randomness-related flaky test generated by Pynguin (project `usolitaire`)

```
def test_case_54():
  str_0 = ''
  query_0 = module_0.where(str_0)
  # omitted
  query_instance_0 = query_0.all(str_0)
  # omitted
  query_instance_1 = query_0.all(query_instance_0)
  bool_1 = query_instance_1.__call__(dict_0)
  # omitted
  query_instance_2 = query_instance_1.__or__(query_instance_0)
  var_0 = query_instance_2.__repr__()
  assert var_0 == "QueryImpl('or', frozenset({('all', (''),),
      QueryImpl('all', (''),), '')), ('all', (''),), '')}))"
```

Figure 8: Unordered collection flakiness generated by Pynguin (project `TinyDB`)

```
def test_case_9():
    none_type_0 = None
    str_0 = '/'
    config_0 = module_0.Config(none_type_0)
    # omitted
    with pytest.raises(FileExistsError):
        config_0.write(str_0)

E IsADirectoryError: [Errno 21] Is a directory: '/'
```

Figure 9: Non-idempotent-outcome flakiness generated by Pynguin (project `bl`)

Verifying Expected Exceptions describes issues happening when a test case expects a certain exception to be thrown and makes assertions about where (i.e., by which class) the exception was thrown. In other words, the test case asserts that the top of the stack trace of an expected exception has a certain value. Such tests can be flaky since a stack trace can change intermittently, even for the same exception. This is caused by optimizations, namely the just-in-time (JIT) compilation, that might decide at any point during the program execution to compile a frequently executed area in the class to native code, which causes it to no longer appear in the stack trace [34, 50]. Fig. 10a shows an example of such a case: The test is expecting an `IndexOutOfBoundsException` thrown by `java.nio.Buffer`. Sometimes, however, this exception is instead thrown by `java.nio.HeapByteBuffer`. This test is flaky due to the default way that the JVM decides to optimize the compilation, where the JIT compilation will compile certain parts of the `java.nio.Buffer` class to native code, causing it to no longer appear on top of the stack trace (as shown in Fig. 10b). Such optimization-based flakiness does not happen in tests generated by EvoSuite$_{FSOff}$ as we updated the `'No Runtime Dependency'` parameter to *true*, which prevents EvoSuite from generating tests

```
@Test(timeout = 4000)
public void test17()  throws Throwable  {
    Name name0 = new Name();
    int[] intArray0 = new int[1];
    intArray0[0] = (-3152);
    Blob blob0 = new Blob(intArray0);
    SafeBag safeBag0 = null;
    try {
        safeBag0 = new SafeBag(name0, blob0, blob0);
        fail("Expecting exception:IndexOutOfBoundsException");
    } catch(IndexOutOfBoundsException e) {
        verifyException("java.nio.Buffer", e);
    }
}
Exception was not thrown in java.nio.Buffer but in java.base/
      java.nio.HeapByteBuffer.get(HeapByteBuffer.java:169): java
      .lang.IndexOutOfBoundsException: 1
```

(a) Flaky test generated by EvoSuite$_{FSOn}$

```
Exception in thread "main" java.
lang.IndexOutOfBoundsException
 at java.nio.Buffer.checkIndex
  (Buffer.java:743)
 at java.nio.HeapByteBuffer.get
  (HeapByteBuffer.java:169)
 ...
 at SafeBag_ESTest.test17
  (SafeBag_ESTest.java:330)
 ...
```

Before JIT compilation

```
Exception in thread "main" java.
lang.IndexOutOfBoundsException
 at java.nio.HeapByteBuffer.get
  (HeapByteBuffer.java:169)
 ...
 at SafeBag_ESTest.test17
  (SafeBag_ESTest.java:330)
 ...
```

After JIT compilation

(b) Stack traces

Figure 10: Flakiness due to *Verifying Expected Exceptions* (project `named-data-jndn`)

that verify thrown exceptions. Although keeping it *false* (default) decreases the number of flaky tests of other causes, it also generates new flaky tests that try to verify the class that throws an exception.

Secondly, EvoSuite$_{FSOn}$ generates flaky tests that produce intermittent *StackOverflowErrors*. This was also discovered by a previous study [22]. The errors occur consistently when the flakiness suppression is turned off. EvoSuite$_{FSOn}$ includes an internal resource threshold—limiting the stack size—to prevent a test case from a *StackOverflowError*, however, the resource checking is nondeterministic and some errors manage to slip through. Such issues do not occur in EvoSuite$_{FSOff}$ because we have disabled the generation of test scaffolding files (first row Table 1), which include a check to prevent infinite loops in recursive methods. However, not generating scaffolding files for test classes makes the generated tests more susceptible to traditional causes of flaky tests [24].

Summary *(RQ3: Root Causes)* Generated tests are flaky for the same reasons as developer-written ones, however, the distribution among those reasons differs: While developer-written flaky tests are often caused by concurrency and networking operations, generated flaky tests tend to be the result of randomness and unspecified behavior. When using flakiness suppression, the picture changes vastly, as the majority of remaining flaky tests do not fit any previously described category of flakiness. Instead, they are caused by runtime optimizations and EvoSuite-internal resource threshold. Notably, both only take effect when certain flakiness suppression mechanisms are activated!

5 RECOMMENDATIONS

5.1 Maintainers of Test Generation Tools

We found EvoSuite's flakiness suppression mechanisms to be very effective and can recommend them to other tools, such as Pynguin, whose rate of flaky tests is currently still higher than the rate of flaky tests in developer-written tests. Nevertheless, EvoSuite still produces flaky tests, which are mainly caused by (1) *Verifying Expected Exceptions* related to the `'No Runtime Dependency'` option, and (2) *StackOverflowErrors* caused by scaffolding. Most notably, both mechanisms are meant—and also accomplish—to prevent traditional causes of flakiness. We therefore recommend revisiting EvoSuite's implementation of the `'No Runtime Dependency'` option and the scaffolding mechanisms to eradicate the flakiness these tend to introduce. Furthermore, we recommend studying and addressing order-dependency in generated tests, as we found high numbers of OD flaky tests for both EvoSuite and Pynguin.

Our dataset [11] should provide the maintainers of both EvoSuite and Pynguin with a large number of real-world examples that can help to reproduce flakiness in generated tests and serve as an evaluation sample for improved versions.

5.2 Developers Using Test Generation

For developers using EvoSuite, we can highly recommend its flakiness suppression options, which we found to be very effective. The remaining flaky tests are mostly caused by *Verifying Expected Exceptions* and *StackOverflowErrors*. The former can be mitigated by disabling the tiered compilation (`-XX:-TieredCompilation`) flag when executing the tests. This will prevent the JVM from compiling frequently executed parts of the bytecode into native code (**JIT** compilation), which however will adversely affect the performance and execution time of other tests. The flaky *StackOverflowErrors* can be mitigated by removing the `NonFunctionalRequirementRule` in the test scaffolding file, which however causes the test to fail consistently. For developers using Pynguin, we recommend setting seeds for random number generators. This should eliminate randomness-related flakiness, which we found to be the most common individual root cause for flaky Pynguin tests. However, there is known criticism about using seeding to avoid flakiness [20]. Developers should also consider order-dependencies between generated tests as a possibility. In our evaluation, about 4 % to 6 % of projects were affected by generated order-dependent flaky tests.

5.3 Researchers Studying Flaky Tests

Since generating tests automatically can be done quickly and efficiently, there is a potential for using tools such as EvoSuite and Pynguin to help in the research on flaky tests. For example, test generation tools could be used to create training data for machine learning models or to systematically expose non-determinisms in individual target projects. While we found generated flaky tests to have similar root causes compared to developer-written flaky tests as long as flakiness-suppression mechanisms are turned off (Table 1), we also found several differences: First, the root cause distribution differs, as flakiness in generated tests is less likely the cause of concurrency or networking issues, and more often the result of randomness and unspecified behavior (Table 4). Second, we found that projects containing developer-written flaky tests are not particularly likely to also produce generated flaky tests and vice versa (Fig. 4).

6 RELATED WORK

Shamshiri et al. [60] studied the effectiveness of automatically generated test suites in Java projects. They applied three automatic test generation tools, Randoop, EvoSuite, and AgitarOne, to a dataset of over 300 faults across five open-source projects, assessing how many bugs the automatically generated tests could detect. Through this process, they also identified the number of flaky tests that were generated by each of the three tools. Of the tests generated by Randoop, which uses feedback-directed random test generation, an average of 21% exhibited non-determinism in their outcomes. EvoSuite produced flaky tests at an average rate of 3%. Only 1% of the tests generated by the commercial, proprietary tool AgitarOne were flaky. While our study and theirs both demonstrate that automatic test generation tools are capable of producing flaky tests, there major are differences. The main objective of our study was to investigate the prevalence and root causes of flaky tests generated by automatic test generation tools. However, the main objective of the study performed by Shamshiri et al. was to assess the bug-finding capability of automatically generated tests. As such, we analyzed the prevalence of generated flaky tests in much more detail (see Section 4.1) and went on to categorize their root causes (see Section 4.3). Furthermore, our subject set of 1 902 Java projects and 4 454 Python projects is significantly larger than the five Java projects used by Shamshiri et al. in their empirical evaluation.

Paydar et al. [53] examined the prevalence of flaky tests, and other types of problematic tests, generated specifically by Randoop. They took between 11 and 20 versions of five open-source Java projects and used Randoop to generate regression test suites, which were the main objects of analysis. Overall, they found that 5% of the automatically generated test classes were flaky, and on average, 54% of the test cases within each of these were flaky. As before, since Paydar et al. were not solely investigating automatically generated flaky tests, they did not examine them to the same level of detail as in our study (for example, they did not consider root causes). Furthermore, while they are all automatic test generation tools, Randoop is significantly different from EvoSuite and Pynguin in that Randoop is based entirely on random search.

Li et al. [40] applied automatic test generation to the repair of order-dependent flaky tests. Their work builds upon the iFixFlakies tool introduced by Shi et al. [63], which uses the statements of existing "cleaner" tests to remove the state-pollution left behind by "polluter" tests that induce order-dependency in the "victim" tests that the tool aims to repair. A weakness of iFixFlakies is that if no such cleaner test exists in the test suite, the tool will be unable to repair the victim. The technique introduced by Li et al. aims to address this weakness by applying automatic test generation to generate cleaners such that the victim may be repaired. Beyond the intersection of flaky tests and automatic test generation, there are no significant similarities between their study and ours.

There have been several previous studies, in which the authors have manually classified the root causes of flaky tests. Luo et al. [44] categorized the causes of the flaky tests repaired by developers in

201 commits across 51 (mostly Java) projects of the Apache Software Foundation. Vahabzadeh et al. [64] categorized the causes of 443 bugs in test code (as opposed to in code-under-test), which they mined from the bug repository of the Apache Software Foundation. They found 51% of these to be flaky tests, which they went on to subcategorize based on their root cause. Eck at al. [21] asked Mozilla developers to categorize the causes of 200 flaky tests they had previously repaired. Lam et al. [37] categorized the type of flakiness repaired in 134 pull requests regarding flaky tests in six subject projects that were internal to the Microsoft Corporation. In spite of much previous work in this area, ours is the first study to categorize the root causes of automatically generated flaky tests.

Flakiness is also an issue in fuzzing, a discipline closely related to automatic test generation: Nourry et al. [49] surveyed 106 fuzzing practitioners and found that developers are often struggling to reproduce build failures or bugs detected by fuzzing. Ding et al. [18] analyzed 23 907 bugs discovered by OSS-Fuzz [59], of which 13 % were flaky. Like our study, they found that timeouts and resource thresholds set by the tool itself (25s and 2.5 GB RAM for OSS-Fuzz [6]) are major causes of flakiness.

7 CONCLUSIONS

Flaky tests are a common and troublesome phenomenon in software testing. Through this study, we were able to show that flakiness does not only affect developer-written tests, but also automatically generated tests: We sampled 6 356 open-source Java and Python projects and generated tests for them using two state-of-the-art test generation frameworks (EvoSuite and Pynguin). After executing the resulting test suites repeatedly, we found flakiness to be even more common among generated tests than among developer-written ones. The root causes of this flakiness are similar, however, its distribution differs: Developer-written flaky tests tend to be caused by concurrency and networking operations, while generated flaky tests are more frequently the result of randomness and unspecified behavior. For both developer-written and generated flaky tests, order-dependency is a frequent cause, which could be a possible direction for future work. While flakiness suppression mechanisms are effective in reducing the flakiness rate among generated tests, they also cause other, previously unseen, forms of flakiness. We hope that our work inspires researchers working on test flakiness and that our insights help maintainers and users of test-generation tools to avoid flakiness in generated tests. We make all data available [11].

ACKNOWLEDGEMENTS

We thank Stephan Lukasczyk, the creator and maintainer of Pynguin, for his support and advice. Phil McMinn and Owain Parry are supported by the EPSRC grant "Test FLARE" (EP/X024539/1) and a Meta Testing and Verification award. Owain Parry received additional support from the EPSRC Doctoral Training Partnership with the University of Sheffield (EP/R513313/1). Gordon Fraser is supported by the DFG project "STUNT" (FR2955/4-1) and by the BMWK project "ANUKI" (50RM2100B).

REFERENCES

[1] [n. d.]. Class Calendar. https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html
[2] [n. d.]. Class Random. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html
[3] [n. d.]. JUnit 4. https://junit.org/junit4/
[4] [n. d.]. Maven Central Repository. https://repo.maven.apache.org/maven2/
[5] [n. d.]. Maven Surefire plugin. https://maven.apache.org/surefire/maven-surefire-plugin/
[6] [n. d.]. OSS-Fuzz: How do you handle timeouts and OOMs? https://google.github.io/oss-fuzz/faq/#how-do-you-handle-timeouts-and-ooms
[7] [n. d.]. Pynguin documentation: Generating Assertions. https://pynguin.readthedocs.io/en/latest/user/assertions.html#simple
[8] [n. d.]. pytest. https://docs.pytest.org/en/7.2.x/
[9] [n. d.]. pytest-random-order: a pytest plugin that randomises the order of tests. https://pypi.org/project/pytest-random-order/
[10] [n. d.]. Python Package Index (PyPI). https://pypi.org/
[11] 2023. Do Automatic Test Generation Tools Generate Flaky Tests? [Dataset]. https://doi.org/10.6084/m9.figshare.22344706
[12] Azeem Ahmad, Erik Norrestam Held, Ola Leifler, and Kristian Sandahl. 2022. Identifying Randomness related Flaky Tests through Divergence and Execution Tracing. In *International Conference on Software Testing, Verification and Validation Workshops (ICST-Workshops)*. 293–300.
[13] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting Flakiness Without Rerunning Tests. In *International Conference on Software Engineering (ICSE)*. 1572–1584.
[14] Andrea Arcuri, Gordon Fraser, and Juan Pablo Galeotti. 2014. Automated Unit Test Generation for Classes with Environment Dependencies. In *International Conference on Automated Software Engineering (ASE)*. 79–89.
[15] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In *International Conference on Automated Software Engineering (ASE)*. 55–66.
[16] Albert Danial. 2021. *cloc: v1.92.* https://doi.org/10.5281/zenodo.5760077
[17] Jens Dietrich, Shawn Rasheed, and Amjed Tahir. 2022. Flaky Test Sanitisation via On-the-Fly Assumption Inference for Tests with Network Dependencies. In *IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*. 264–275.
[18] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *International Conference on Mining Software Repositories (MSR)*. 131–142.
[19] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *International Conference on Mining Software Repositories (MSR)*. 254–264.
[20] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting Flaky Tests in Probabilistic and Machine Learning Applications. In *International Symposium on Software Testing and Analysis (ISSTA)*. 211–224.
[21] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The Developer's Perspective. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 830–840.
[22] Zhiyu Fan. 2019. A systematic evaluation of problematic tests generated by EvoSuite. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. 165–167.
[23] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* (1971), 378.
[24] Gordon Fraser. 2018. A tutorial on using and extending the EvoSuite search-based test generator. In *International Symposium on Search Based Software Engineering (SSBSE)*. 106–130.
[25] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT Software Engineering Notes*. 416–419.
[26] Gordon Fraser and Andrea Arcuri. 2013. EvoSuite: On the challenges of test case generation in the real world. In *International Conference on Software Testing, Verification and Validation (ICST)*. 362–369.
[27] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology* (2014), 1–42.
[28] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology* (2015), 1–49.
[29] Martin Gruber and Gordon Fraser. 2022. A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It. In *International Conference on Software Testing, Verification and Validation (ICST)*. 82–92.
[30] Martin Gruber and Gordon Fraser. 2023. Debugging Flaky Tests using Spectrum-based Fault Localization. In *International Conference on Automation of Software Test (AST@ICSE)*. 128–139.
[31] Martin Gruber and Gordon Fraser. 2023. FlaPy: Mining Flaky Python Tests at Scale. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. 127–131.

[32] Martin Gruber, Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2021. An Empirical Study of Flaky Tests in Python. In *International Conference on Software Testing, Verification and Validation (ICST)*. 148–158.

[33] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* (2011), 649–678.

[34] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* (2008), 1–32.

[35] Wing Lam. 2020. International Dataset of Flaky Tests (IDoFT). http://mir.cs.illinois.edu/flakytests

[36] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *International Symposium on Software Testing and Analysis (ISSTA)*. 204–215.

[37] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A Study on the Lifecycle of Flaky Tests. In *International Conference on Software Engineering (ICSE)*. 1471–1482.

[38] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *International Conference on Software Testing, Verification and Validation (ICST)*. 312–322.

[39] Wing Lam, August Shi, Reed Oei, Sai Zhang, Michael D. Ernst, and Tao Xie. 2020. Dependent-Test-Aware Regression Testing Techniques. In *International Symposium on Software Testing and Analysis (ISSTA)*. 298–311.

[40] Chengpeng Li, Chenguang Zhu, Wenxi Wang, and August Shi. 2022. Repairing Order-Dependent Flaky Tests via Test Generation. In *International Conference on Software Engineering (ICSE)*. 1881–1892.

[41] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. 168–172.

[42] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2020. Automated Unit Test Generation for Python. In *International Symposium on Search Based Software Engineering (SSBSE)*. 9–24.

[43] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. 2023. An empirical study of automated unit test generation for Python. *Empirical Software Engineering* (2023), 36.

[44] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *International Symposium on Foundations of Software Engineering (FSE)*. 643–653.

[45] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In *International Conference on Software Engineering (ICSE)*. 91–100.

[46] Phil McMinn. 2004. Search-Based Software Test Data Generation: A Survey. *Journal of Software Testing, Verification and Reliability* (2004), 105–156.

[47] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 233–242.

[48] John Micco. 2016. Flaky Tests at Google and How We Mitigate Them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[49] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The Human Side of Fuzzing: Challenges Faced by Developers During Fuzzing Activities. *ACM Transactions on Software Engineering and Methodology* (2023).

[50] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. 1–12.

[51] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2022. A Survey of Flaky Tests. *IEEE Transactions on Software Engineering* (2022), 17:1–17:74.

[52] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn. 2022. Surveying the Developer Experience of Flaky Tests. In *International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 253–262.

[53] Samad Paydar and Aidin Azamnouri. 2019. An Experimental Study on Flakiness and Fragility of Randoop Regression Test Suites. *Lecture Notes in Computer Science* (2019), 111–126.

[54] Qianyang Peng, August Shi, and Lingming Zhang. 2020. Empirically Revisiting and Enhancing IR-Based Test-Case Prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*. 324–336.

[55] Darrel A Regier, William E Narrow, Diana E Clarke, Helena C Kraemer, S Janet Kuramoto, Emily A Kuhl, and David J Kupfer. 2013. DSM-5 field trials in the United States and Canada, Part II: test-retest reliability of selected categorical diagnoses. *American journal of psychiatry* (2013), 59–70.

[56] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Journal of Software Testing, Verification and Reliability* (2016), 366–401.

[57] Wing Lam Ruixin Wang, Yang Chen. 2022. iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests. In *International Conference on Software Engineering: Companion Proceedings (ICSE Companion)*. 120–124.

[58] Sebastian Schweikl, Gordon Fraser, and Andrea Arcuri. 2022. EvoSuite at the SBST 2022 Tool Competition. In *International Workshop on Search-Based Software Testing (SBST@ICSE)*. 33–34.

[59] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. *USENIX Security Symposium* (2017).

[60] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *International Conference on Automated Software Engineering (ASE)*. 201–211.

[61] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. 2018. How do automatically generated unit tests influence software maintenance?. In *International Conference on Software Testing, Verification and Validation (ICST)*. 250–261.

[62] Samuel Sanford Shapiro and Martin B Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* (1965), 591–611.

[63] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A Framework for Automatically Fixing Order-Dependent Flaky Tests. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 545–555.

[64] Arash. Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An Empirical Study of Bugs in Test Code. In *International Conference on Software Maintenance and Evolution (ICSME)*. 101–110.

[65] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EvoSuite at the SBST 2021 Tool Competition. In *International Workshop on Search-Based Software Testing (SBST@ICSE)*. 28–29.

[66] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting Flaky Tests via Non-Idempotent-Outcome Tests. In *International Conference on Software Engineering (ICSE)*. 1730–1742.

[67] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* (1945), 80–83.

[68] Zhe Yu, Fahmid Fahid, Tim Menzies, Gregg Rothermel, Kyle Patrick, and Snehit Cherian. 2019. TERMINATOR: Better Automated UI Test Case Prioritization. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 883–894.