An Empirical Study of Flaky Tests in Python

Martin Gruber
BMW Group
Munich, Germany
martin.gr.gruber@bmw.de

Stephan Lukasczyk
University of Passau
Passau, Germany
stephan.lukasczyk@uni-passau.de

Florian Kroiß
University of Passau
Passau, Germany
kroiss@fim.uni-passau.de

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

Abstract—**Tests that cause spurious failures without any code changes, i.e., *flaky tests*, hamper regression testing, increase maintenance costs, may shadow real bugs, and decrease trust in tests. While the prevalence and importance of flakiness is well established, prior research focused on Java projects, thus raising the question of how the findings generalize. In order to provide a better understanding of the role of flakiness in software development beyond Java, we empirically study the prevalence, causes, and degree of flakiness within software written in Python, one of the currently most popular programming languages. For this, we sampled 22 352 open source projects from the popular PyPI package index, and analyzed their 876 186 test cases for flakiness. Our investigation suggests that flakiness is equally prevalent in Python as it is in Java. The reasons, however, are different: Order dependency is a much more dominant problem in Python, causing 59 % of the 7 571 flaky tests in our dataset. Another 28 % were caused by test infrastructure problems, which represent a previously undocumented cause of flakiness. The remaining 13 % can mostly be attributed to the use of network and randomness APIs by the projects, which is indicative of the type of software commonly written in Python. Our data also suggests that finding flaky tests requires more runs than are often done in the literature: A 95 % confidence that a passing test case is not flaky on average would require 170 reruns.**

Index Terms—**Flaky Test; Python; Empirical Study**

I. INTRODUCTION

Regression testing is a widely adopted practice in modern software development. When new code gets checked in to the version control system, an automated testing pipeline is triggered ensuring that the most recent changes did not break existing functionality. The basic assumption behind regression testing is that the tests themselves behave deterministically. If a test does not behave deterministically, but passes and fails when run multiple times without any changes to the code, the test is regarded as *flaky*. Flaky tests confront developers with a dilemma: If they continue taking all test failures seriously, they may waste precious time and resources trying to find bugs that might not even be in the system under test (SUT), but rather in the test code or in the test infrastructure. On the other hand, if they disable tests that show flaky behaviour, they may reduce the effectiveness of their test suite and may miss bugs.

The most common strategy to reduce the impact of flaky test failures is to rerun tests upon failure multiple times and accept a single passed execution to make the test qualify as passed. Many test frameworks and continuous integration systems support marking tests as flaky and running them up to a specified number of times upon failure before reporting them as actually failed. This practice, however, has multiple drawbacks: First,

it can only mitigate the problem by making flaky failures less likely, as a flaky test could still fail all reruns. For large test suites with high flaky failure rates, this approach may not be effective at making a build pass. Second, it might hide problems that should actually be fixed. While flakiness is often suspected to be rooted in the test code, it might also be caused by the code under test itself. Rerunning flaky tests might therefore mask actual bugs. Third, it wastes resources. Google, for example, reportedly spends 2 % to 16 % of their resources on re-running flaky tests [1]. Consequently, there is a need to study flakiness in depth in order to understand its nature and to devise strategies to avoid it.

Previous research aiming to provide a deeper understanding of flaky tests revolves around a limited set of Java projects. While Java is widely used, other programming languages gained huge popularity over the last decade, in particular Python: More than 200 000 packages are listed in the Python Package Index (PyPI)[1] at the time of this writing, ranging from web frameworks such as Django to data analysis tools such as NumPy or machine-learning libraries such as TensorFlow. Despite its huge popularity, very little is known about flakiness in Python, whether previous findings on Java also apply to Python, and what further research is required in order to mitigate the problem of flakiness in the Python world.

In this paper, we aim to fill this gap by conducting a large empirical study on 22 352 Python projects, consisting of 876 186 test cases. Using a total of 400 re-runs of these tests, we shed light on the questions of (1) how prevalent flaky tests are in Python, (2) what the root causes of flakiness are in Python, and (3) just *how* flaky these flaky tests really are. In detail, this paper makes the following contributions:

a) Dataset: We derive a large dataset of 22 352 Python projects with 876 186 tests, of which 7 571 tests from 1 006 projects show flakiness. The resulting dataset, which we share with the community [2], consists of all artifacts as well as the data produced by 400 test runs.

b) Study: We evaluate the test results with regard to the extent, the cause, and the degree of flakiness we observed. For each flaky test we provide a classification for the root cause of its flakiness, and we further manually investigate 100 non-order-dependent flaky tests to additionally provide a fine-grained classification into 13 established categories.

---

[1] https://pypi.org/, accessed 2021–01–18.

c) Methodology: Using the extensive amount of data on flaky tests in Python, we derive a novel, more stable approach to estimate the number of reruns needed in order to expose possible flakiness at a specific confidence level.

Our study shows that flakiness is an equally prevalent problem in Python as it has been shown to be in Java. The reasons for flakiness, however, differ: order-dependency between tests is a much more dominating reason in the context of Python than it is for Java, and non-order-dependent tests are predominantly caused by network and randomness APIs, which are representative of the common application areas of Python. We also identify infrastructure flakiness as a new type of test flakiness, which may in particular affect researchers conducting large experiments on flaky tests. By providing statistical estimates of the required reruns to detect or mitigate flakiness, and by releasing all data freely, we hope to foster research on test flakiness in Python, and on automated identification and classification techniques for flakiness.

II. Background

Several approaches have been proposed to automate the identification, classification, and elimination of flaky tests.

A. Types of Flakiness

Luo et al. [3] introduced 10 categories of flakiness (*Async Wait*, *Concurrency*, *Test Order Dependency*, *Resource Leak*, *Network*, *Time*, *IO*, *Randomness*, *Floating Point Operations*, and *Unordered Collections*), which were later extended by Eck et al. [4] with the categories *Too Restrictive Range*, *Test Case Timeout*, *Platform Dependency*, and *Test Suite Timeout*. Unlike most other causes, order dependencies can be properly identified automatically [5], [6]. Therefore, these categories are often grouped into *order-dependent* and *non-order-dependent* causes (the latter referring to all other 13 categories).

One might argue that failures caused by order dependencies can be easily avoided by enforcing a particular test order. However, dependencies between test cases can still cause failures as adding new tests or removing existing ones might break the test suite. Therefore, developers should always be interested in avoiding test dependencies.

We can further categorize order-dependent tests as follows [7]: A test t can be order-dependent either because another test p running before t disturbs its execution, or because another test s is not run before t, although t requires s to run before it. In the first case, t is called a *victim* and p is called a *polluter*. Test p changes a shared state that t tries to read from in a way that t fails. When run in isolation, a victim passes, as the state is not affected by the polluter. In the second case, t is called a *brittle* and s is called a *state-setter*. Test t needs s to set up a shared state, e.g., a database connection, before it can run successfully. When run in isolation, a brittle fails if the required state has not been set up.

Besides the already introduced categories, there exists a 15th category not previously discussed: *infrastructure flakiness*. It describes a test being flaky due to reasons outside the project's code but inside the test execution environment, for example

Table I: Previous studies and their identification strategies

Source	identify flakiness via	no. repetitions
[3], [9]	search for commits fixing flaky tests	-
[4]	search for already fixed flaky tests	-
[5]	search in issue tracker	-
[10]	rerun, same order	5
[11]	rerun, different order	20-60*
[7]	rerun, same order	10
[8]	rerun, same order	100
[12]	rerun, different order chosen by PIT [13]	17
[6]	data-flow analysis**	-

* iDFLAKIES reruns failing test orders up to two times.
** data-flow analysis can only be used to detect order-dependent flakiness.

failing installation of dependencies. Infrastructure flakiness differs from other types of flakiness as it is not caused by the project itself, but by external components. It is therefore a form of transitively induced flakiness.

An example for infrastructure flakiness we experienced is the `pip` Python package management tool failing to install certain dependencies, resulting in `ModuleNotFoundErrors` or `FileNotFoundErrors` when executing the tests. Despite the cause of flakiness being the network, this cannot be classified as network flakiness, as it is not the project which is trying to access the network. Other causes involve permission errors and a lack of disk space. So far, infrastructure flakiness has not been formalized, although its effects have been previously observed, for example in a previous study which mentions that, out of 315 tests showing flakiness in continuous integration, only 44 cases were reproducible locally using 100 reruns [8].

B. Detecting Flaky Tests

To study flakiness, researchers have applied two strategies to build datasets containing flaky tests: (1) Search for commits fixing flakiness or issues reporting flaky tests in version control systems or bug trackers; (2) Rerun tests up to a certain number of times and check whether their verdicts change between runs.

Luo et al. [3], who conducted one of the first studies on flaky tests, used the first method by mining 201 commits that likely fix flaky tests in 51 open-source projects and manually verified this assumption. In order to obtain a larger set of flaky tests in an automated fashion, later studies employed the second method. They used various numbers of reruns (cf. Table I) without investigating this number in detail. By using 400 reruns, which is more than most prior studies, we aim to derive a proper estimation on how many reruns should be conducted to build a representative dataset of flaky tests.

Most previous work focused on a limited number of popular and large Java projects; several studies [7], [10], [11] use similar datasets, originating in the study by Luo et al. [3]. This sampling approach is reasonable for measuring the performance of a tool aiming to detect or classify flakiness, as it makes the evaluation more stable and comparable to other studies. However, this practice does not contribute to the overall understanding of flakiness, especially not outside the Java-world, where much less research is available.

C. Mitigating Flaky Tests

To mitigate flakiness, researchers have proposed several techniques that aim at detecting flaky tests in a resource efficient and automated way, classifying flaky tests in order to assist the debugging process, or automatically fixing flaky tests.

DEFLAKER [10] and IDFLAKIES [11] are both tools for automatically detecting flaky tests, offering performance advantages over repetitive reruns. DEFLAKER does so by analyzing coverage information and test verdicts from prior runs, therefore completely avoiding re-executions of any tests. IDFLAKIES aims to detect flaky tests by applying a smart random-order rerun strategy, allowing it to also partially classify the root cause of the flakiness. However, it can only distinguish between order-dependent and non-order-dependent flaky tests.

Several other techniques focus exclusively on order-dependent test: PRADET [6] tries to detect order-dependent flaky tests by using data-flow analysis to filter test orders containing potential order-dependencies, minimizing the number of test runs needed to expose order-dependent behavior. IFIXFLAKIES [7] aims at automatically fixing order-dependent flaky tests by suggesting patches extracted from other test code.

ROOTFINDER [8] aims to derive a more fine-grained classification for non-order-dependent flaky tests: Using a binary instrumentation framework it tries to collect distinctive information about a test execution, which it then compares between passing and failing runs in order to find patterns that predict test failures. Another classification tool proposed by Ziftci et al. [14] can also help to identify the root cause of flakiness by comparing the execution of passing and failing runs and pointing at the first line of code where the two executions diverge. Both approaches provide support, but cannot fully automate, finding the root causes of flakiness.

D. Types of Flakiness in Python

The categorization of flaky tests discussed in Section II-A is not language specific, and thus also applies to Python. There are, however, some language specific peculiarities that have an impact on some types of flakiness.

Floating point flakiness: Flakiness in Python cannot be caused by floating point operations, as floating point arithmetic in Python is—despite suffering from the well-known binary-decimal representation issues of IEEE-754—always deterministic.

Platform dependency: In many ways, Python hides the underlying system structure from the user for example by automatically extending the size of an integer in case its value reaches the word limit. Nevertheless, it is possible for an execution to differ because of platform dependencies, for example because the size of an object (which is accessible via `sys.getsizeof`) differs between 32-bit and 64-bit systems.

Unordered collections: The category *Unordered collection* plays a special role in Python: The internal ordering within a set depends on the `__hash__` function. This function used to be deterministic up until Python 3.2 from where on it was randomly seeded making it non-deterministic, a change made due to security reasons. After all, this means that Python (in its default configuration) only has unordered collections

(a) Number of tests   (b) Lines of code

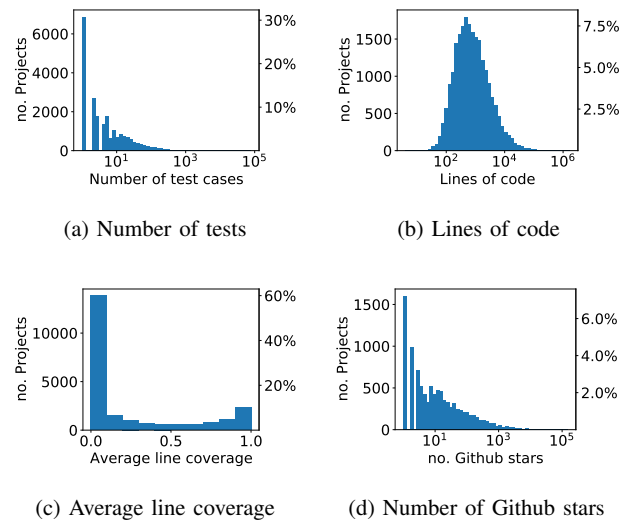(c) Average line coverage   (d) Number of Github stars

Figure 1: Statistics about the dataset of 22 352 Python projects

since Python 3.3. It is worth noting that the default behavior of collections is subject to frequent change in Python, as for example dictionaries are now order-preserving since Python 3.8.

III. STUDY SETUP

In this paper we aim to empirically answer the following research questions:

RQ1: How frequently does flakiness occur in Python?
RQ2: What types of flakiness are prevalent in Python?
RQ3: What is the degree of flakiness of flaky tests in Python?

A. Dataset

We scanned the entire Python Package Index (PyPI) for suitable sample projects. PyPI is the official third-party software repository for Python. It features 284 112 projects and 4 787 968 users (as of 2021–01–18), and is used as the default package source by many package managers including `pip`.

By using PyPI we hope to create a dataset, which is large and diverse enough to represent the language without including overly small toy projects. Such toy projects will exist on GitHub, however, they are unlikely to be published to the community via PyPI. While our dataset does contain small projects, the share of tests contributed by projects having less than 100 LOC is only 3.5 %.

We limited our exploration to projects whose source code is available on GitHub and whose tests can be executed using PyTest[2], the most commonly used test execution framework in Python. We ended up with 22 352 Python projects matching these criteria. For each project, we consider its current state on 2020–08–16.

The projects contain between 1 and 76 301 (project 'cap-idup') test cases. The median number of test cases per project is 3, the mean is at 39.2. In total the projects contain 876 186

[2]https://pytest.org, accessed 2021–01–18.

150

test cases. Fig. 1a shows a histogram of the distribution regarding the number of test cases per project. To discover tests within a project, we rely on PyTest, which scans for files, classes, and methods, whose names contain the keyword 'test'[3]. Python tests can be parametrized, meaning the same test is executed multiple times with different inputs. PyTest reports each parametrization of a test as a separate test case. This practice is reasonable as parametrized inputs are often of complex nature (e.g. files), covering different functionality within the code, and can therefore be considered separate tests.

To estimate the sizes of our projects, we measured the non-comment source lines of code (LOC) for each project using CLOC [15]: The smallest projects contain less than 10 LOC. The largest project ('napalm-yang') features 1.68 million lines of Python code. The median number of LOC per project is 682, the mean is at 2 791.9. The total number of LOC is slightly above 62 million. Fig. 1b shows a histogram of the LOC-distribution.

As an indicator for the quality of the projects' test suites, we measured the average line coverage reported in our test runs, which is depicted in Fig. 1c. Aside from a large number of low-coverage test runs (which can partly be attributed to error-ing tests), we also see about $10\,\%$ of projects yielding a strong line coverage of above $90\,\%$. The mean line coverage across all projects is $24.6\,\%$, the median is $3.7\,\%$.

To give an impression about the popularity of the investigated projects, Fig. 1d depicts the number of Github stars per project. The number of stars ranges from zero up to 149 185 (project 'tensorflow') with a median of 4 and a mean of 117.9.

We also wanted to know which domains of application the selected projects address, which we measured by looking at the topics developers assigned for their projects on PyPI. PyPI provides a fixed set of 296 hierarchically organized topics, that developer can give their projects.[4] This field, however, is optional, so not all projects have topics: 7 480 of all investigated projects specified at least one topic. Furthermore, for some classifiers, multiple values of varying granularity can be specified without the hierarchy being enforced. A single project can for example specify "Topic :: Security :: Cryptography" without specifying "Topic :: Security". While this is a technical possibility, it is common to specify all matching topics within the hierarchy to make the project more visible to search algorithms. Table II depicts the 25 most frequently assigned topics. While the three top-most entries are very generic and offer little insights about the projects' domains, the following topics contain more information, showing that the selected projects cover a large variety of domains.

At last, we looked at the maturity of the projects in the dataset, by examining their development status as specified on PyPI. Of all investigated projects, 9 703 specified their development status. We depict the number of projects per status in Table III, showing a diverse picture with a tendency towards pre-production states.

Table II: 25 most used topics of the investigated projects

Topic	no. Projects
Software Development :: Libraries :: Python Modules	2 116
Utilities	1 289
Software Development :: Libraries	910
Scientific/Engineering	796
Software Development	623
Software Development :: Build Tools	439
Software Development :: Testing	435
Internet :: WWW/HTTP	420
Internet :: WWW/HTTP :: Dynamic Content	312
Scientific/Engineering :: Bio-Informatics	261
Scientific/Engineering :: Artificial Intelligence	259
Scientific/Engineering :: Mathematics	238
Scientific/Engineering :: Information Analysis	208
Scientific/Engineering :: Physics	205
Internet	171
Database	160
Software Development :: Quality Assurance	148
System :: Systems Administration	143
Software Development :: Libraries :: Application Frameworks	133
Scientific/Engineering :: Visualization	123
System :: Networking	121
Text Processing :: Linguistic	120
Security	114
Text Processing	114
Scientific/Engineering :: Astronomy	99
UNSPECIFIED	14 872

Table III: development status of the investigated projects

Development status	no. Projects
4 - Beta	3 173
3 - Alpha	2 566
5 - Production/Stable	2 281
2 - Pre-Alpha	1 466
1 - Planning	178
6 - Mature	30
7 - Inactive	12
UNSPECIFIED	12 649

B. Detecting Flaky Tests

To automate the detection of flaky tests, we built the FLAPY tool, which is implemented in Python and is available under the GNU LGPL Open Source license. We provide all source code of our analysis tool on GitHub[5].

FLAPY takes as input the project folder of a Python project and executes the tests of the project either in a constant order between the test runs or in a random order. The random-order execution allows choosing the level of granularity (i.e., class, module, package, or project level), at which the tests are shuffled. Further, one can choose the number of repeated test runs that shall be executed. At the core of our tool is PyTest[2], which collects all tests within the project directory and executes them. We export the test results along with coverage information. FLAPY uses BENCHEXEC [16] to separate the execution of the test cases from the parent process of our tool and to control the tests' access to external resources, such as network or hard disk.

---

[3]https://docs.pytest.org/en/stable/goodpractices.html# conventions-for-python-test-discovery, accessed 2021–01–18.

[4]https://pypi.org/classifiers/, accessed 2021–01–18.

[5]https://github.com/se2p/FlaPy, accessed 2021–01–18

Table IV: Python APIs and keywords indicating flakiness

Category	Keyword
Async wait	`sleep`
Concurrency	`thread`
	`threading`
IO	`builtins.stat`
	`pathlib.Path.is_dir`
Network	`requests`
Time	`time`
Random	`random`
Unordered Collection	`__hash__`
	`builtins.set.__contains__`

C. Analyzing Flaky Tests

In order to support the classification of flaky tests, we developed an approach which searches the execution-traces of a test for a set of keywords, that are indicative for different categories of flakiness. To determine relevant keywords, we created a minimal representative Python test for each known category of flakiness and traced multiple executions of these while collecting all called functions using Python's tracing API. By looking at differences between these traces, we identified the function calls that are most characteristic for the specific type of flakiness. Table IV shows the keywords we extracted from the traces of our minimal examples respective to each category for which we were able to find distinctive keywords.

When aiming to classify a given flaky test, we can now trace the execution of the flaky test, and search for appearances of the representative flaky APIs (or keywords) within these traces. If one of the keywords matches, this suggests that the test might be flaky due to the corresponding category. If a keyword does not appear, that category is unlikely.

This techniques suffers from two obvious limitations: (1) There are multiple ways to trigger the same type of flakiness. We mitigate this issue by considering execution traces as well as the tests' code, enabling us to find indirect usages of our minimal flaky APIs. (2) In case multiple flaky APIs are used, we have no way of telling which caused the test to flake. This, however, is a general limitation of a text-based search, in contrast to a semantic program analysis. In consequence, we use this technique only to support a manual classification.

D. Methodology

1) RQ1: Prevalence of Flakiness in Python: In order to identify flaky tests in Python projects, we execute the tests of a project in an isolated manner using FLAPY. For each project, we executed its tests 200 times in the same order and 200 times in a random order, shuffled on project level to expose as many order dependencies as possible. We performed the test executions on a SLURM-managed cluster [17] consisting of 91 nodes giving each test job 16 GB of RAM as well as a 24 h timeout. In total, this took 484 h.

We consider a test to be flaky iff it passed at least once and failed at least once. Inspecting the topics of the projects that contain flaky tests, we hope to find an indication which domains are prone towards flakiness. We also look at the maturity of projects containing flakiness, expecting to find more flakiness in alpha- and beta-phase projects, rather than in mature projects.

2) RQ2: Types of Flakiness in Python: Besides investigating the mere extent of flakiness, we also want to provide explanations for the observed non-deterministic behavior, and therefore classify the flaky tests identified as part of RQ1.

We noticed that infrastructure flakiness often appears in failure bulks, where all runs executed on the same machine within a short period of time fail. In order to distinguish infrastructure flakiness from flakiness within the project, we therefore sliced the 200 runs into 10 iterations of 20 runs each. All 20 runs within an iteration were executed on the same machine in an uninterrupted sequence. The 10 iterations were distributed across different machines and always had several hours of temporal distance. We consider a test to be flaky due to non-determinism in the test infrastructure, if it exhibits flaky behavior only between iterations but not within an iteration (e.g., the test passed for all runs within an iteration and failed for all runs within another). In contrast to that, we consider a test to be flaky due to reasons lying within the project's code, iff it exhibits at least one passing and at least one failing run within the same iteration for at least one iteration.

For all non-infrastructure related flaky tests, we distinguish between order-dependent flakiness (OD) and non-order-dependent flakiness (NOD) as follows: If a test shows flaky behavior in test runs featuring the same test order, it is being categorized as NOD, regardless of its behavior when executed in random order. If a test shows no flaky behavior when run in the same test order but does show flaky behavior when executed in random order, it is categorized as OD.

Following a previous approach [7], we further categorize order-dependent tests by running them in isolation: If an OD test always passes when run in isolation, it is a victim, if it constantly fails, it is a brittle. For NOD flakiness we furthermore distinguish the remaining 13 categories. Unlike the classification of OD tests into victims and brittles, the classification of NOD tests cannot be easily automated, which is why we classify the NOD flaky tests manually.

As we found more NOD flaky tests than we could classify in a reasonable amount of time, we selected 100 of them via random-stratified sampling: We randomly chose 100 projects out of the 279 projects that contained at least one NOD flaky test. For each project we then randomly selected one of its NOD flaky test cases. In doing so, we hope to retrieve an unbiased sample. We specifically avoided picking multiple flaky tests from the same project, as they are likely to be flaky due to the same root cause.

Each test case was then manually classified independently by two authors using (1) the project's code, (2) the test-execution reports from all 200 runs therefore including at least one failure-trace, and (3) the category-distinctive keywords found by the keyword-trace-search (Section III-C). In the next stage, we resolved all cases in which the two authors came to a different conclusion or were both unsure regarding the category of flakiness, via an in-depth discussion.

3) RQ3: Degree of Flakiness: Besides root causes of flakiness, we also measure its degree. This is mainly of interest to researchers aiming to derive representative datasets on flakiness, who might be asking questions such as:

(a) "How many times do I have to rerun a test to be $95\,\%$ sure that the test is not flaky?"
(b) "If I rerun my tests ten times, which portion of the existing flakiness can I expect to find?"
(c) "How many reruns do I need in order to find $80\,\%$ of all existing flakiness?"

From a practical point of view it is mainly relevant to understand how often to rerun *failing* tests to identify flakiness. Thus, for practitioners a more relevant question might be:

(d) "In case a test execution fails, how many reruns should I conduct in order to be sure the failure indicates a bug and not just a flaky test?"

We propose an alternative metric for calculating the recommended number of reruns: In order to estimate the number of reruns required to find flakiness, previous studies [10], [12] based their decision on the number of reruns needed to unveil flakiness *once*. For a test t we call this number $n_{t,\text{once}}$, which is defined in the following way: Assume we run our tests n times, resulting in a list of runs $\langle r_1, r_2, \ldots, r_n \rangle$ with the function $\mathrm{verdict}(t, r_i)$ defining the outcome of test t in run r_i. Following PyTest's JUnit-XML plugin[6], a test's verdict can take the values $PASS$, $FAIL$, $ERROR$, or $SKIP$. $n_{t,\text{once}}$ is the first index, for which the test t showed both a passing and a non-passing ($FAIL$ or $ERROR$) execution:

$$n_{t,\text{once}} = \max(\min(\{i \mid \mathrm{verdict}(t, r_i) = PASS\}),$$
$$\min(\{i \mid \mathrm{verdict}(t, r_i) \in \{FAIL, ERROR\}\}))$$

Exposing all flaky tests within a test suite T therefore requires $max(\{n_{t,\text{once}} \mid t \in T\})$ reruns.

There are two issues with this approach: (1) it is hard to reproduce, as it is based on single-time events, which might have been an "(un)lucky punch"; (2) it is unstable, as by design it utilizes only a limited amount of data which can hardly be extended by conducting more reruns—all verdicts seen after the flakiness was exposed once are ignored.

We therefore propose a new method for estimating the required number of reruns to expose flakiness, which is based on all verdicts the tests exhibited, and is able to provide a confidence level: For every test t we calculate its passing rate $P_t(PASS)$ as well as its non-passing rate $P_t(FAIL/ERROR)$ as the ratio between the number of passed (respectively failed or errored) executions, and the number of all executions. Note that these may not add up to 1.0, as a test can also result in the verdict $SKIP$. Assuming the executions of the same test in different runs are independent from each other, these rates are also the probabilistic chances for the test to pass/not pass its next execution. The independence assumption is strengthened by the fact that we already filtered out bulked failures that occurred due to infrastructure flakiness.

[6]https://docs.pytest.org/en/stable/usage.html#creating-junitxml-format-files, accessed 2021–01–18.

We define the probability $U_t(n)$ for unveiling the flakiness of test t after n reruns as one minus the probability of not seeing any flakiness, meaning the test never fails, never passes, or is always skipped (meaning it neither passes nor fails):

$$U_t(n) = 1 - (1 - P_t(FAIL/ERROR))^n$$
$$- (1 - P_t(PASS))^n$$
$$+ (1 - P_t(PASS) - P_t(FAIL/ERROR))^n$$

We then calculate $n_{t,p}$ as the minimum number of reruns needed to unveil the flakiness of test t with a probability p, calling it the *statistical number of reruns* with confidence p:

$$n_{t,p} = \min(\{n \mid U_t(n) > p\})$$

This metric addresses both issues of $n_{t,\text{once}}$, as it is not based on single-time events and utilizes data from all reruns. Question (a) can now be answered by calculating the median value of $n_{t,0.95}$ ($95\,\%$ confidence) for all tests t within a given dataset. We visualize our results and answer questions (b) and (c) by looking at the sum of flaky tests found after n reruns, suggested by metric x for a set of tests T:

$$S(n, x, T) = |\{t \mid n_{t,x} \leq n \ \wedge \ t \in T\}|$$

To answer question (d), we calculate the chance of a failing flaky test to pass at least once within the next n iterations, which is $1 - (1 - P_t(PASS))^n$, and derive n such that this probability exceeds $95\,\%$. We answer RQ3 separately for OD and NOD flaky tests, as for OD flaky tests repeated execution is only one way to expose their flakiness [6], while for NOD flaky tests, there is currently no alternative to repetitive reruns.

E. Threats to Validity

Internal Validity. Python execution tracing has limitations; for example, some builtin functions cannot be traced, hence we potentially miss these calls in our traces during the keyword-trace-search (Section III-C). However, two authors manually classified all cases independently, and the traces were only one source of input. While we did execute the tests on different machines, these were still very similar both in their hardware- and their software-configuration, which might have had a negative effect on the number of platform-dependency related flakiness we were able to expose.

External Validity. We conducted our analysis on a large sample of projects from PyPI. However, our sampling procedure resulted in 22 352 projects, which is only about $10\,\%$ of all available projects on PyPI. Therefore, our conclusions might not generalize to other Python projects, and they also might not generalize to other projects in other programming languages.

Construct Validity. The number of iterations necessary to expose all cases of flakiness is unknown. By using 400 reruns, we achieve a fairly high confidence that we exposed a large percentage of flaky tests. Furthermore, by introducing a statistical metric we are able to report a necessary number of runs to expose flakiness with a given confidence level. However, it is still possible that we misclassified non-order-dependent flaky tests as order-dependent, if they showed flakiness only in random order executions.

153

Table V: Development status of flaky projects

Development status	no. flaky projects	no. projects	flakiness rate	average no. tests / project
5 - Production/Stable	124	2 281	5.3 %	79.5
UNSPECIFIED	577	12 649	4.5 %	34.8
3 - Alpha	116	2 566	4.4 %	22.5
4 - Beta	133	3 173	4.1 %	53.9
2 - Pre-Alpha	50	1 466	3.4 %	14.5
1 - Planning	6	178	3.4 %	18.8

Table VI: Topics of flaky projects

Topic	no. flaky projects	no. projects	flakiness rate	average no. tests / project
Scientific/Engineering :: Artificial Intelligence	17	259	6.6 %	35.3
Scientific/Engineering	47	796	5.9 %	82.3
Scientific/Engineering :: Bio-Informatics	15	261	5.7 %	83.1
Utilities	65	1 289	5.0 %	30.0
Software Development :: Testing	21	435	4.8 %	23.2
Software Development :: Libraries :: Python Modules	100	2 116	4.7 %	82.7
Software Development	29	623	4.7 %	60.2
UNSPECIFIED	667	14 872	4.5 %	36.1
Software Development :: Build Tools	16	439	3.6 %	17.4
Software Development :: Libraries	33	910	3.6 %	34.8
Internet :: WWW/HTTP	15	420	3.6 %	23.6

IV. RESULTS

A. RQ1: Prevalence of Flakiness in Python

In total, we found 7 571 tests exhibiting non-deterministic behavior in 1 006 projects. This gives us a ratio of 0.86 % of all investigated tests being flaky and a ratio of 4.5 % of all investigated projects containing at least one flaky test.

Table V shows projects containing flakiness grouped by their maturity. Against our expectation, flakiness is more common in projects with higher levels of maturity, than those in earlier development phases: 5.3 % of all stable projects contain flakiness whereas only 3.4 % of all pre-alpha projects do so. This observation can be explained by the fact that projects in later phases in general do more testing (which can be seen in the last column of Table V) and are therefore more likely to also contain at least one flaky test.

Table VI depicts the flaky projects by topic. We can observe a tendency towards the science and engineering domain, in particular towards artificial intelligence. In contrast to that, build tools, libraries, and internet-related projects seem to contain less flakiness than average projects. Unlike for the development status, however, these differences can not all be attributed to certain topics conducting more tests: Projects specifying the topic with the highest flakiness rate have a comparable amount of tests to projects which do not specify a topic, suggesting that the topic does have direct influence on the prevalence of flakiness within a project. Note that for Table VI we show only topics which occur at least 15 times in all flaky projects.

Summary *(RQ1: How frequently does flakiness occur in Python?)* We found 7 571 tests that exhibit flaky behavior, making up a portion of 0.86 % of the tests we examined, with flakiness being more common in more mature projects as well as projects from the scientific and engineering domain.

Table VII: Root causes of the flakiness we observed

Root Cause	relative	total
Infrastructure	28 %	2 158
Test Order Dependency	59 %	4 461
victims		3 168
brittles		738
could not be analyzed		555
Non-order-dependent	13 %	952
sample		100
Network		42
Randomness		37
IO		7
Time		4
Async Wait		3
Concurrency		3
Resource Leak		2
Test Case Timeout		1
Unordered Collections		1
Too Restrictive Range		0
Platform Dependency		0
Test Suite Timeout		0

```
1 def test_komoot_multi_result():
2     g = geocoder.komoot(location, maxRows=3,
   timeout=10)
3 >   assert g.ok
4     assert len(g) == 3
5
6 AssertionError: assert False
7   where False = <[ERROR - HTTPConnectionPool(host=
   'photon.komoot.de', port=80): Read timed out. (
   read timeout=10)] Komoot - Geocode [empty]>.ok
```

Figure 2: Network timeout causing flaky failure

B. RQ2: Types of Flakiness in Python

Table VII shows the number of flaky tests we found for each category of flakiness. We found 2 158 tests to be instances of infrastructure flakiness. Furthermore, we encountered 4 461 tests, which are flaky due to order dependencies. Running the order-dependent tests in isolation, we found a majority of them to be victims (3 168) and a minority to be brittles (738). Roughly ten percent of all order-dependent tests could not be analyzed, mostly due to our test framework being unable to find the specified test. Common obstacles for test re-identification are inheritance between test case classes, complex parametrizations, and test IDs differing from JUnit-XML IDs.

In the following, we discuss the most prominent categories of NOD flakiness (952 tests) we found in our sample together with representative examples. The most common category in our sample is network. Fig. 2 shows an example for network flakiness taken from project 'geocoder', a Python library that takes a location in form of an address and returns its geographic coordinates using online services. The test failed because of an HTTP request exceeding the given timeout of 10 s.

We also found many flaky tests due to randomness, with an example shown in Fig. 3. The test is part of project 'gamble', a library that implements functionalities concerning cards and dice. The depicted test creates a deck of cards, shuffles them,

```
1  def test_deck_init() -> None:
2  """test that we can create a deck of cards"""
3      deck = Deck(shuffle=False)
4      # omitted
5      last_top = deck.top
6      deck.shuffle(times=10)
7  >   assert last_top != deck.top
8
9  AssertionError:
10     assert <Card:7> != <Card:7>
11     where <Card:7> = <Deck[46]>.top
```
Figure 3: Randomness causing flaky failure

```
1  def test_cronrule():
2      tests = [
3          # omitted
4          ('*/5 * * * *', operator.gt, 0),
5          # omitted
6      ]
7      for condition, op, result in tests:
8          cr = CronRule(condition)
9  >       assert op(cr.interval, result)
10
11 AssertionError: assert False
12     where False = <built-in function gt>(0, 0)
```
Figure 4: System time causing flaky failure

```
1  def test_mpr121_irq(GPIO, bus):
2      # omitted
3      for i in range(13):
4          dev.write_word(0, 1 << i)
5          irq()
6          sleep(0.01)
7  >       handlers[i].assert_called_once_with(True, i)
8          # omitted
9
10 AssertionError: Expected "mock" to be called once.
       Called 0 times.
```
Figure 5: Asynchronous waiting causing flaky failure

```
1  from hypothesis import given
2  from hypothesis.strategies import from_regex,
     integers, lists, tuples
3
4  strat_strings = from_regex(r"\A[^0-9]*\Z")
5  strat_mod = tuples(integers(), lists(integers(
     min_value=0), max_size=10))
6  strat = strat_strings, lists(strat_mod, max_size
     =5)
7
8  @given(lists(tuples(*strat)))
9  > def test_sort_property(e):
10     # omitted
11
12 hypothesis.errors.DeadlineExceeded: Test took
     201.15ms, which exceeds the deadline of 200.00ms
```
Figure 6: Fuzzing framework causing flaky timeout

and checks if the top-most card has changed. With a low, but existing chance, this test will fail as despite the cards have been shuffled, the top-most card might still be the same.

A less common, but existing cause of flakiness, is the wrong usage of the system time. Fig. 4 presents an example for that, taken from project 'cronjob', an API for the UNIX job scheduler cron. The test creates a cronjob running every five minutes and checks if there is at least one second between the current system time and the next execution of the job. It fails if the next execution happens within the next second.

Of all tests in our sample, 3% were flaky because they did not properly wait for an asynchronous call to complete before making assertions on it. One of them is depicted in Fig. 5, which is part of project 'piripherals', a tool to interact with peripherals for the RaspberryPi. As it uses a mocking framework, its tests can also be executed successfully on other hardware. The test fails in case a handler, which is a mocked object, has not been called within 0.01 seconds.

Fuzzing tools can also produce flakiness as shown in Fig. 6: The depicted test belongs to 'humansort', a tool that sorts filenames in a more human readable way. The project uses HYPOTHESIS [18], a property-based testing library for Python, which dynamically parametrizes a test case, searching for edge cases that might cause it to fail. However, HYPOTHESIS itself can also cause test failures, like in this case, where a deadline was exceeded. We also found cases in which HYPOTHESIS caused flakiness falling into the randomness category: As its search for test input values is non-deterministic, it might find a bug in one execution, but not in the next one. This can be avoided by using its builtin cache mechanism.

Some cases also match multiple categories, like the one shown in Fig. 7, which belongs to project 'arlib', a com-

mon interface for archive manipulation. The test is flaky because the order in which file names are returned by `ar.member_names`, which internally calls `os.listdir`, is not deterministic, however, it is compared against an ordered data structure. The category of flakiness in which this test case falls, is debatable: We classified it as flaky due to unordered collection, but it could also be counted as flaky due to IO, as the non-determinism involves the file system. Furthermore, the flakiness could be removed by not insisting on a certain ordering between the elements, hence indicating a too restrictive range. The case exemplifies, that the categories of flakiness are not distinct and one flaky test might match several categories.

Summary *(RQ2: What types of flakiness are prevalent in Python?)* 59% of all flakiness we observed was caused by order dependencies, 28% by infrastructure flakiness, and 13% mostly and to equal degrees by the use of networking and randomness APIs.
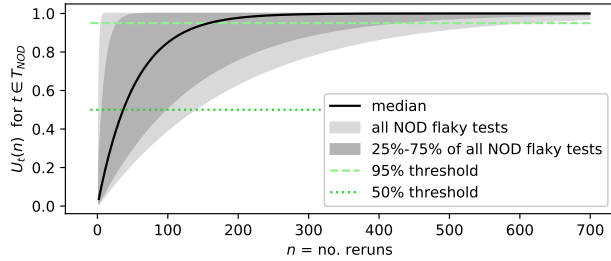
```
1  def test_arlib_read(fname):
2      if sys.version_info[0] >= 3:
3          with arlib.open(fname, 'r') as ar:
4  >           assert ar.member_names == ['a.txt', 'b.txt']
5
6  AssertionError:
7      assert ["b.txt", "a.txt"] == ["a.txt", "b.txt"]
```
Figure 7: Flakiness matching multiple categories (Unordered collection, IO, Too restrictive range)
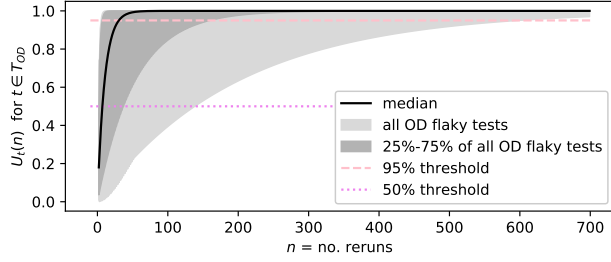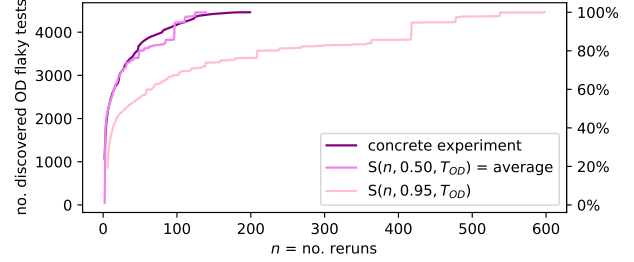
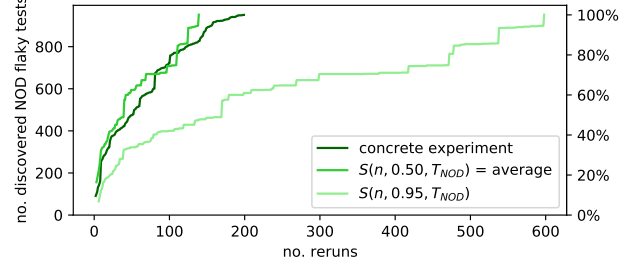(a) Chance to unveil flakiness after n reruns for all OD flaky tests

(b) Chance to unveil flakiness after n reruns for all NOD flaky tests

Figure 8: Statistical chance to unveil flakiness

(a) No. reruns necessary to unveil order-dependent flakiness

(b) No. reruns necessary to unveil non-order-dependent flakiness

Figure 9: Flakiness found after n reruns

C. RQ3: Degree of Flakiness

Fig. 8a and Fig. 8b depict the cumulative probability distribution of $U_t(n)$ of all OD tests $t \in T_{OD}$ and NOD tests $t \in T_{NOD}$. The average number of reruns needed to expose a test's flakiness—which is the answer to question (a)—can now be derived visually, as the intersection between the probability of exposing the flakiness of an average flaky test (black line), and the 95 % threshold (dashed line).

Fig. 9a and Fig. 9b show the sum of flaky tests $S(n, x, T)$ found after n reruns for all OD flaky tests ($T = T_{\text{OD}}$) and all NOD flaky tests ($T = T_{\text{NOD}}$). Metric $x = \text{once}$ shows the number of flaky tests we found in our concrete experiment, $x = 0.5$ is the average number of flaky tests one can expect to find, and $x = 0.95$ is a more conservative estimation.

For order-dependent flakiness, the number of flaky tests discovered in our concrete experiment is very close to the average-curve. For NOD flaky tests, this relationship is weaker, however still strong, showing that the confidence gained from studying the number of tests needed to unveil flakiness once is only 50 %. Fig. 9a shows that the majority of order-dependent flaky tests is discovered within the first 50 reruns, followed by a more steady growth, and finally almost saturation when exceeding 150 reruns. In contrast to that, the rate at which new non-order-dependent flaky tests were discovered decreased only slightly even after 150 reruns.

With the help of Fig. 9 we are now able to answer all the researcher's questions introduced in Section III-D3.

(a) In order to be 95 % sure that a test case is not flaky due to non-order-dependent reasons, one would have to run it at least 170 times. For order-dependent flakiness, this

number is lower, at 31 random-order executions.

(b) If you rerun your tests ten times, you should not expect to find more than 33 % of all NOD flaky tests and 54 % of all OD flaky tests on average.

(c) Finding 80 % of all NOD flakiness requires on average at least 110 reruns; or at least 472 reruns, in case you want to be 95 % sure. For OD flaky tests, these numbers are less than half as large with 49 reruns for 50 % confidence and 209 reruns for 95 % confidence.

Answer (c) suggests that even after 200 reruns, we did not find all flaky tests in our dataset. To estimate the percentage of flakiness we found, we calculate the fraction of flaky tests one can expect to find after 200 reruns with a 95 % confidence:

$$\frac{S(200, 0.95, T_{\text{NOD}})}{\mid T_{\text{NOD}} \mid} = \frac{580}{952} \approx 61\,\%$$

Therefore, with a 95 % chance, we found at least 61 % of all NOD flakiness in our dataset. This statement is based on the assumption, that the flaky tests still hidden in the test suite behave similarly to the flaky tests we were able to expose, with regard to their passing and not-passing rates.

The practitioner's question is answered based on the passing probability $P_t(\text{pass})$ for $t \in T_{\text{NOD}}$ and $t \in T_{\text{OD}}$:

(d) To check if a failure occurred due to flakiness, for the majority of NOD flaky tests, only 1 rerun is needed to achieve a statistical confidence of 95 %. For OD flakiness, 3 reruns are needed to achieve the same confidence.

This means that common practice of failure reruns [19] is generally sufficient (when ignoring infrastructure flakiness, which often manifests itself in failure bulks).

Table VIII: Comparison: iDFlakies dataset & our dataset

Study	Projects	Tests	Flaky				
			Projects	Tests	OD	NOD	Infr.
iDFlakies	683	89 568	12 %	0.47 %	50 %	50 %	-
This study	22 352	876 186	4.5 %	0.86 %	59 %	13 %	28 %

While recommending an ideal number of reruns remains an inherently complex problem, our results still show that finding flakiness via rerun requires a tremendous amount of test executions, which should be a further motivation to find other ways of exposing flakiness.

Summary *(RQ3: What is the degree of flakiness of flaky tests in Python?)* Flaky tests in Python have a low failure rate, resulting in a low number of reruns necessary to check, if a failure was flaky (1 rerun for 95 % confidence on NOD flaky tests), but a high number of reruns necessary to check if a test in general contains flakiness (170 reruns for 95 % confidence on NOD flaky tests).

V. RELATED WORK

The largest study on test flakiness in Java is the ID-FLAKIES [11] study. In this study, a set of 683 Java projects, consisting of 639 popular GitHub projects and 44 projects from previous studies, was screened for flakiness. While a total of 1 974 084 tests were reported for the overall dataset, only 945 out of 5 171 modules were analyzed. To allow for a better comparison, we determined that this dataset contains 89 568 test cases. Table VIII puts these and our findings in contrast. While the rate of projects containing at least one flaky test is higher in the Java dataset, this can be attributed to the biased sampling technique by which the Java projects were chosen, as all 26 projects taken from Bell et al. [10] are chosen specifically because they contain flakiness. Looking at the total number of flaky tests discovered, on the other hand, both datasets exhibit a similar rate of around 1 in every 120 (this study) vs. 1 in every 210 (iDFlakies) test being flaky. We also see that order-dependency seems to be a more pressing issue in Python while non-order-dependency is rarer. Note, however, that our study does not use the exact same setup as the iDFlakies study, as we conduct more reruns without using smart scheduling.

A fine-grained categorization of non-order-dependent flaky tests was previously performed on 400 known flaky tests found in the commit history of Apache projects [3] and the Mozilla issue tracker [4]. Table IX demonstrates that, whereas they reported Async Wait and Concurrency to be the most common root causes of NOD flakiness, we found only little evidence of these categories. On the other hand, the categories causing most NOD flakiness in our dataset (namely Networking and Randomness), played only a minor role in the two other studies.

Order-dependent flakiness was investigated in detail by Shi et al. [7], who found 100 victims and 10 brittles in a set of 110 order-dependent Java tests. The same trend towards far more victims than brittles can also be observed in our results. One reason why brittles are rarer than victims might be that in

Table IX: Root causes of non-order-dependent flaky tests

Root Cause	This study	[3]	[4]
Async Wait	3	74	52
Concurrency	3	32	61
Resource Leak	2	11	14
Network	42	10	0
Time	4	5	4
IO	7	4	0
Randomness	37	4	3
Floating Point Operations	-	3	6
Unordered Collections	1	1	0
Too Restrictive Range	0	-	40
Test Case Timeout	1	-	18
Platform Dependency	0	-	10
Test Suite Timeout	0	-	4
Infrastructure	2 158	-	-

modern test frameworks state-setters can be declared actively, which restricts possible test orders and therefore avoids brittles.

Zhang et al. [20] proposed a technique to predict the failure rate of software before deployment. Although we also used probabilistic models, our aim is to derive sound recommendations on how to determine flakiness. Our methodology to classify flaky tests involved traces, which were also used to localize error causes [21]. Test flakiness may have knock-on effects on other aspects such as test prioritization [5], mutation analysis [12], or build crashes [22], and for research purposes flakiness can also be seeded [23]; since our paper represents the first study on flakiness in Python, we focused on a basic understanding of flakiness before considering such applications.

VI. CONCLUSIONS

Flaky tests represent a fundamental challenge in modern software development. While previous research investigated this problem predominantly in the context of Java software development, we demonstrated that flaky tests are equally prevalent in the Python ecosystem. Using a dataset of 22 352 projects from the Python package index we found 1 006 projects in which flakiness exists and a total of 7 571 flaky tests. Although this number is comparable to prior findings on Java, the reasons for this flakiness differ, which is indicative of the different target domains for Python, such as scientific software or web applications.

Besides a demand to extend existing techniques and tooling also to the Python environment, these findings suggest that future work on the peculiarities on flakiness in Python will be required, for example to address flakiness in machine learning or scientific software. To this purpose, we provide our dataset [2] and the tooling[7] as open source to the community, hoping to foster research on new techniques to automatically identify, classify and eliminate flakiness. We encourage the investigation of flakiness in other upcoming languages such as Go or Rust as well as a more detailed exploration of infrastructure flakiness.

---

[7]https://github.com/se2p/FlaPy, accessed 2021–01–18

REFERENCES

[1] J. Micco, "The state of continuous integration testing@ google," 2017.

[2] M. Gruber, "An empirical study of flaky tests in python," Jan. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4450434

[3] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 643–653.

[4] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "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)*. ACM, 2019, pp. 830–840.

[5] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test indenpendence assumption," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2014, pp. 385–396.

[6] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 2018, pp. 1–11.

[7] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "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)*. ACM, 2019, pp. 545–555.

[8] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 101–111.

[9] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 2018, pp. 534–538.

[10] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 433–444.

[11] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "Idflakies: A framework for detecting and partially classifying flaky tests," in *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2019, pp. 312–322.

[12] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2019, pp. 112–122.

[13] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 449–452.

[14] D. C. Celal Ziftci, "De-flake your tests: Automatically locating root causes of flaky tests in code at google," in *ICSME 2020-International Conference on Software Maintenance and Evolution*, 2020.

[15] A. Danial, "Cloc: Counting lines of code," 2015.

[16] D. Beyer, S. Löwe, and P. Wendler, "Benchmarking and resource measurement," in *International SPIN Workshop on Model Checking Software (SPIN)*. Springer, 2015, pp. 160–178.

[17] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 2003, pp. 44–60.

[18] D. MacIver and Z. Hatfield-Dodds, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[19] J. Micco, "Flaky tests at google and how we mitigate them," https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html, 2016, accessed: 2020–10–19.

[20] X. Zhang and H. Pham, "Software field failure rate prediction before software deployment," *Journal of Systems and Software*, vol. 79, no. 3, pp. 291–300, 2006.

[21] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2003, pp. 97–105.

[22] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 857–862.

[23] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment. A case study on mutation testing and program repair," *CoRR*, vol. abs/1912.03197, 2019.