DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Reducing Effort for Flaky Test Detection through Dynamic Program Analysis

Aaron Tacke

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Reducing Effort for Flaky Test Detection through Dynamic Program Analysis

Reduzierung des Aufwandes der Erkennung von Flaky Tests durch dynamische Programmanalyse

Author:	Aaron Tacke
Supervisor:	Prof. Dr. Alexander Pretschner
Advisor (1st):	Fabian Leinen
Advisor (2nd):	Daniel Elsner
Submission Date:	August 15, 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Garching, August 15, 2022 Aaron Tacke

Abstract

Flaky tests, that yield varying outcomes for an invariant codebase, hinder regression testing. If one is not aware that they behave non-deterministically, their outcome may incorrectly indicate new bugs or hide real faults. To prevent this and resolve their flakiness, flaky tests must be detected as such, which is generally costly. If the non-determinism of flaky tests occurs due to the test order, varying the test order allows an efficient detection. Otherwise, to identify Non-Order-Dependent (NOD) flakiness, tests are usually executed repeatedly to possibly detect differing results.

We propose an approach that reduces the number of required executions by identifying the majority of all non-flaky tests beforehand. Previous studies have shown that dynamic program analysis already helps to fix flakiness by evaluating program traces to find root causes of flakiness. Consequently we suppose that function call traces can indicate the existence of flakiness and that the absence of certain functions can identify a test as non-flaky.

First, we present different approaches to identify functions, whose execution is characteristic for test flakiness. Thereafter, we assess these approaches and the sets of functions they produce. Both manual and semi-automated approaches achieve promising results. Finally, we evaluate whether they help to reduce effort for NOD flakiness detection.

While we can greatly reduce the number of necessary reruns for NOD flakiness detection, this does not save execution time as desired, as we mostly exclude small tests while generating overhead costs for the function call tracing. In nearly half of all examined projects our approach performs better, otherwise, it misses some NOD flaky tests or leads to increased execution times. In conclusion, the introduced approach to detect flakiness is not generally recommendable but significantly better in certain cases.

Contents

1. Introduction

For regression testing, the outcome of a test after a change should show whether previously existing functionality still works properly [1]. This means, that a previously passing test should fail if and only if a bug was introduced. For this relation to hold, the result of a test has to be deterministic. In reality, however, test outcomes are often not deterministic. These non-deterministic tests are called *flaky tests* and drastically complicate development [2]. They do not only have a negative impact on the individual developer's work but also on the overall efficiency of projects [3].

As a developer, flaky tests are obstructive for debugging [4]. New test failures are supposed to indicate a new bug [1]. But if the test failed due to flakiness, developers are falsely led to investigate a non-existing fault, since the reason for the failure might be completely independent of the previously changed code [2]. However, knowing that a test is flaky might lead to the developer disregarding the result, even though flaky tests often belong to actual bugs in the Code Under Test (CUT) [5]. In both cases, the reliability of the test suite is diminished [3].

Looking at the development process as a whole, flakiness generally reduces the cost-effectiveness of regression testing. Google, for example, spends between 2% and 16% of test execution costs on running flaky tests [6]. This is because failing tests are rerun multiple times, to examine whether a test failure is caused by a newly introduced bug or by previously existing flakiness [2]. If one of the reruns passes, the test is known to be flaky. Thereby, increasing the number of reruns also increases the probability of detecting flakiness [7]. But without more efficient approaches to detect flakiness, an disadvantageous trade-off between the execution costs and the reliability of test suites remains [8].

As a result, test flakiness has emerged as an important research topic. To understand flaky tests, so-called *root causes* for flakiness are examined [2, 3]. Regarding the detection of flaky tests, the most efficient approaches only deal with specific root causes of flakiness [9, 10, 4]. Studies on fixes for flaky tests either also focus on their root causes [11, 12] or dynamically gather information about the test executions [13, 5]. These execution traces are compared for failing and passing runs to check where the non-determinism is observed initially. Often, the initial introduction of non-determinism is the origin of the flakiness [13].

Approaches that efficiently detect all types of flaky tests are lacking. Predicting flakiness based only on the textual content of the test code is insufficiently accurate [14]. Thus, the CUT should also be involved in flakiness detection. This could be done by tracing executions dynamically. Previous studies on fixing flakiness by using system call tracing [15] suggest that merely analyzing system calls does not suffice for all relevant root causes. Hence, all function calls should be traced when executing tests. Although these execution traces are already used successfully for locating the origin of flakiness [13, 5], they are not yet studied for flaky test detection.

Therefore, we suggest an approach for the detection of flaky tests based on function call tracing. Using function call traces of test executions, we estimate which tests are likely to be flaky. By only rerunning these likely flaky tests, we aim to reduce the effort of detecting flakiness without substantially missing flaky tests. Since there already are efficient methods for detecting Order-Dependent (OD) flaky tests whose outcomes depend on the execution order of the tests in a test suite [9], we focus on NOD flaky tests. These behave non-deterministically independently of their execution order (see Definition 1). For these NOD flaky tests, we want to find functions that their flakiness can be attributed to. We postulate, that at least one of these *characteristic functions* is called by the vast majority of NOD flaky tests.

We present, implement, and evaluate different approaches to obtain an optimal set of characteristic functions. First, we select characteristic functions manually based on existing literature, function call traces of test executions, the test code, and the CUT. Second, we optimize sets of characteristic functions for all root causes separately, based on a manually labeled dataset. Finally, a fully automated optimization without a separation of root causes is assessed.

Moreover, we evaluate the detection of NOD flaky tests using characteristic functions. For this, a statistical model quantifies the accuracy, required reruns, and speed of our approach. In contrast to other research [10, 4], our goal is to detect NOD flaky tests of all root causes. In addition, we do not predict flakiness directly. We use function call tracing and characteristic functions to select tests to be rerun. Thus, we still rerun unmodified tests in the end, which verifies suspected flakiness and filters out all false positives.

We find that obtaining characteristic functions is possible in different ways and yields promising results. Although all approaches are quantitatively similar (with regards to precision and recall), the connection of the functions with flakiness decreases with increasing automation. Therefore selecting characteristic functions fully automatically leads to erroneous results, while selecting them purely manually is not strictly necessary.

In order to evaluate the manually selected set of characteristic functions further, we use them to pre-select 38.4% of all tests that include 92.9% of the NOD flaky tests. Rerunning these pre-selected tests requires far fewer executions for equal recognition rates (42% fewer reruns for detecting 80% of all NOD flaky tests). Looking at the runtime of these executions, our approach is generally up to 20% less efficient but still recommendable in nearly 50% of the analyzed projects.

2. Foundations

In this chapter, we will explain and define the foundations, that this thesis is based on. First, we take a look at flaky tests in general and then discuss specifics, which are important for our analysis of Python tests.

2.1. Flaky Tests

The term *flaky tests* describe tests with non-deterministic outcomes [2]. This means, that multiple executions of such tests may produce different outcomes, without changing the test code or CUT. Definition 1 will specify the flaky tests that are relevant for this thesis in Section 2.1.2. In the next sections, we will clarify why flaky tests are problematic and how different kinds of flaky tests call for dedicated research.

2.1.1. Relevance of Flaky Tests

Flaky tests may fail independently of the previously changed code which can both incorrectly indicate new bugs and hide real faults. Trying to avoid this, many testing frameworks rerun all failing tests to rule out failures due to flakiness. Failing tests are therefore only registered as failing if all following reruns also fail. But even then, tests that are registered as failing might still be flaky, if they just consistently failed by chance. If at least one rerun of a failing test passes, it is registered as flaky, since the test yielded different outcomes on equal versions of the codebase. This approach has a big impact on the effectiveness of regression testing: 2% to 16% of testing costs of Google are generated by rerunning possibly flaky tests [6].

Apart from the execution costs, developers still have to fix flaky tests after their detection. Otherwise flaky tests could still fail coincidentally and waste the time of developers that look for non-existing bugs in their code. Furthermore, ignoring flaky tests would lead to ignoring real bugs, since many fixes for flaky tests also fix real bugs in the CUT [2].
And even if flaky tests could be detected and fixed easily, they would still be a source of friction in the development process: Developers, who know about flaky tests in a test suite tend to perceive it as more unreliable [3].

Depending on the type of flaky tests, preventing, detecting, reproducing [8], and fixing flakiness can be of varying complexity. This calls for extensive investigations in the fields of understanding, detecting, and fixing different kinds of flaky tests.

2.1.2. NOD Flaky Tests

NOD flaky tests are non-deterministic independently of the execution order of tests. In contrast to OD flakiness, the detection of NOD flaky tests is very cost-intensive. This is what we want to address in this thesis. Since NOD flaky tests are rarely defined and not always described identically [9, 16], we create an unambiguous basis for this thesis by giving the following definition:

Definition 1 *An NOD flaky test is a test that can both pass and fail when executed multiple times without changing the execution environment, i.e., the order of tests, the executed test code, or the CUT.*

Reasons for NOD flakiness can be manifold. So-called root causes all introduce non-determinism, but in very different ways. *Randomness* is a very simple example: Dividing by a random number between 0 and 9 will eventually result in a division by 0, which may lead to a failure of an otherwise passing test in 10% of the executions. As a second example, *Network* connection failures can also lead to failing tests, although the CUT is correct in most cases [2].

Because of these differences, efficient detection methods for NOD flaky tests normally focus on single root cause [10, 4]. To detect all kinds of NOD flaky tests, rerunning the test is the most prevalent option, although high numbers of reruns are necessary (e.g. 170 reruns for 95% certainty on the median test) [8].

Fixing NOD flaky tests is often based on their root causes [11]. However, approaches depending on root causes for fixing flakiness are not universally applicable, since considering all root causes is prohibitively complicated. More general approaches rely on execution traces, which are recordings of the test executions, for example by registering executed lines of codes, functions, or values of variables. Comparing these traces for passing and failing test executions allows pinpointing the introduction of non-determinism and often the reason for flakiness. [13, 5]. Unfortunately, we first have to obtain these traces for passing and failing runs, which is even more costly than detecting flakiness.

2.1.3. OD Flaky Tests

OD flaky tests are flaky due to the non-deterministic ordering of the tests in a test suite. It follows, that running tests in a predefined order automatically fixes the cause of flakiness. Unfortunately, reordering tests is important for cost-effectiveness [17], which is why a predefined order is obstructive for most test suites.

Tests, with an order-dependent outcome (therefore OD flaky tests) require a certain state, that is shared with at least one other test, to work correctly. This state can either be internal (e.g. a value of a global variable) or external (e.g. an open connection to a database). An erroneous execution of test t can happen in different ways: First, another test that should set up a required state beforehand is executed after t. Second, a test ruins the correct state for t by being executed beforehand instead of after t. Finally, combinations of multiple relevant states and various state-restoring and state-destroying tests can occur.

Detecting and reproducing OD flaky tests is easier than for NOD flakiness [9]. To detect OD flaky tests efficiently, we can deliberately reorder tests to provoke flawed states. This way, we can try to cause passing and failing runs deliberately, which reduces the necessary number of reruns. To reproduce flakiness, we just store one passing and one failing test order. The same methods will ruin or set up the same states and two differentiating outcomes can be caused repeatedly.

Fixing flakiness is generally complicated, but there are promising approaches for OD flaky tests. Varying the order of different tests helps understanding which tests depend on each other and therefore require a certain execution order [18]. Observing the internal and external state is more complicated, but yields the exact interconnection, that causes the dependency between two tests [15]. In more than 50% of OD flaky tests, the flakiness can be automatically fixed by identifying and calling state-setting functions beforehand [12].

2.1.4. Flaky Tests in Python

Gruber et al. explain why the occurrence and distribution of root causes depends on the used programming languages [8]. They found that this is mainly caused by the different characteristics and prevalent use cases. In Python, flakiness due to *Floating Point* operations is less likely, since all standard arithmetic operations are deterministic. And since Python is often used for web applications, *Network* is a more frequently occurring root cause than in other languages [8].

They also present a whole new class of flakiness [8]: Besides OD and NOD, infrastructure flaky tests describe non-deterministic behavior due to reasons outside the project's code. The possibly failing installation of dependencies might determine, whether the test runs successfully. Therefore, the outcome of a test depends on the execution environment. The dependence on the execution environments implies, that infrastructure flaky tests are per Definition 1 not NOD flaky tests and therefore not taken into account in this thesis.

With 0.86%, the occurrence rate of flaky tests in Python (also obtained from Gruber et al. [8]) is undeniably high. Even without infrastructure flaky tests, which may be excluded in other definitions of flakiness [15], 0.62% of the examined tests are (NOD or OD) flaky. This shows, that flakiness is a profound problem in Python that warrants further and more specific research.

2.2. Tracing Tests in Python

To trace tests in Python, we will first talk about testing in Python using the pytest[1] framework. Afterwards we deal with tracing Python executions in general, which also allows tracing test executions using pytest.

2.2.1. Pytest

Pytest is the most commonly used framework to execute tests in Python [8]. It usually executes all functions whose name starts or ends with `test` in files that start (or end) with `test_` (or `_test` respectively) [19]. For `assert` statements, pytest checks whether the following condition holds and marks the test as *F* (for failed) if it evaluates to `False`. If an unexpected exception appears during the test, it is marked as *E*, otherwise, the test passes [19].

Apart from these functionalities, pytest supports more advanced concepts. And if pytest does not support a feature natively, it is easily extensible using plugins. These plugins can extend and intervene in all parts of pytest (e.g. selecting, preparing, running, and evaluating tests) [19].

2.2.2. Tracing Function Calls

Since Python is an interpreted language, accessing the state of an execution is comparatively easy [20]. Regardless of whether the execution states are collected or not, the interpreter has to have access to them either way. The only additional effort is processing and outputting the collected states [20]. For these tasks, there are so-called *hooks*, prepared entry points for tracers and debuggers to interrupt the execution. Functions that should be called at these entry points can be specified [20]. To set a tracer, `sys.settrace(myfunc)` leads to the function `myfunc` being called at every executed line of code [21]. Setting a so-called profiler using `sys.setprofile(myfunc)` would only lead to calls of `myfunc` when entering or leaving a function during the execution. With `threading.setprofile(myfunc)`, we can extend the tracing to newly started threads. Unfortunately, tracing function calls does not work for independent programs started by the traced process, for which we would have to limit ourselves to tracing system calls [22].

Existing tracers (e.g. the trace module[2]) also use this functionality but offer more features [21]. They can aggregate and count different events, add timekeeping and apply various filters and output formatting. But all these possibilities of existing tracers come with a significant runtime overhead [20]. So for a narrowed use, where large amounts of data should be retrieved, it is worthwhile to implement and optimize a new specific tracer.

[1]https://pytest.org
[2]https://docs.python.org/3/library/trace.html

In Python, there are different types of function calls [21]. On the one hand, there are normal functions written in Python. On the other hand, functions that are written in C can be called. These C functions can either be built-in functions (e.g. the function `open` to open a file) [21] or functions added by a module (e.g. the function `numpy.random.uniform`) [23]. The advantages of writing C functions are the ability to optimize code further [23] and direct access to functions provided by the operating system [21]. Due to these advantages, many functions that are important in Python, are actually written in C. But not all function call tracers trace calls to these C functions [21]. Therefore, it is crucial to consider, which function calls are relevant when deciding for a tracer.

3. Related Work

Early studies by Luo et al. [2] about flaky tests examined their manifestation and yielded general implications for two main fields of study. First, the detection of flaky tests should be tailored to the root causes of flakiness and tests should be checked for flakiness when they are first added to the test suite, since 78% of flaky tests are already flaky when they were first implemented. Second, there is no silver bullet for fixing flaky tests but many of the fixes (22.56%) also fix a bug in the CUT, which is why flaky tests should not be removed or disabled [2].

Most studies about flakiness can be categorized in either detection or fixing of flaky tests. Defining, analyzing, and assigning root causes could be introduced as a third category. We will regard those studies as fixing flaky tests since understanding the root causes is a crucial subtask for resolving flakiness.

Some researchers pursue a broader vision. Instead of detection or fixes, they study the overall existence of flaky tests and examine their effect on developers, software systems, and testing practices:

Eck et al. conducted surveys with developers about both, specific flaky tests and implications of flakiness in general [3]. They found that flakiness is changing the perceived reliability of a test suite and impacts the resource allocation and the scheduling of tests. This emphasizes the negative effect of flakiness. The fact that some of the observed root causes had not yet been reported in previous studies shows, that flaky tests have not been sufficiently investigated.

Analyzing the lifecycle of flaky tests, Lam et al. proposed ways to reduce the runtime of flaky tests without affecting their outcome. They observed several cases where claimed fixes for flaky tests did not even reduce the frequency of failures due to flakiness [7]. Therefore, flaky test detection should be executed independently of the developer's beliefs.

3.1. Flakiness Detection

The simplest approach to detect flaky tests is to rerun each test multiple times and to check, whether varying outcomes occurred. Since rerunning is very time-consuming, many test environments only rerun tests that are annotated as possibly flaky by developers [2].

Gruber et al. [8] studied flaky tests in Python and concluded, that the median NOD flaky test requires 170 reruns to be detected with 95% certainty. This number is far higher than the number of reruns often reported in literature and industry and shows the trade-off between accuracy and costs of execution when detecting flaky tests. They also identified a new type of flakiness called infrastructure flakiness. Infrastructure flaky tests are deterministic on one machine, but produce different results in different environments due to external components. Thus, they cannot be identified as flaky by rerunning them on one machine at all.

In contrast, OD flaky tests can be detected comparatively easily as shown by Lam et al. [9]. By reordering classes and functions in a test suite, OD flakiness can be provoked repeatably and more efficiently.

To gain similar efficiency for some NOD flaky tests, Silva et al. [10] added load to the CPU and memory before rerunning tests. This way they caused flaky failures by *Concurrency* to happen more often. Although this only helps for one of the multiple root causes of NOD flakiness, they also analyzed annotations of the detected tests: 74 of the 75 detected tests had not been annotated as such by the developers and therefore were not even checked for flakiness before.

Similarly, Dutta et al. [4] were able to improve the detection of flaky tests caused by *Randomness*. Their FLASH algorithm dynamically determines how many reruns are necessary (with a default maximum of 500), by assessing differences in the values that occur in assertions of tests.

An even more specific approach was taken by Shi et al. [24]. Since some Java methods have non-deterministic specifications, they replaced 31 of them with their own implementations to enforce non-deterministic behavior. Therefore, flakiness that is induced by these methods occurs more often and is easier to detect.

Verdecchia et al. [14] tried to predict test flakiness using machine learning without rerunning tests at all. They identified words that are used in the code of flaky tests without examining the CUT. The precision of this approach (at most a mean of 0.83 over all projects) is too low to replace traditional detection by rerunning. Using this approach to choose which tests to rerun is insufficient due to the low recall (at most a mean of 0.55 over all projects).

Bell et al. [25] presented a way to maintain the high precision of rerunning tests without the cost of rerunning them at once: Instead, they trace the code that is executed by a test. Test executions on multiple versions of the codebase can then be treated as reruns, as long as the executed code does not change. Although this approach is very efficient, it does not check for flaky tests while someone works on them and is unable to directly confirm fixes for flaky tests.

3.2. Fixing Flaky Tests

Both automated and manual approaches to resolve flakiness depend on the information, where and why flakiness arises. Thus, identifying the root causes of flakiness aids developers and improves automatic procedures.

Since the flakiness introduced by individual root causes can be amplified based on the environment, Terragni et al. [11] set up different environments to determine root causes. Each of these so-called containers has specific properties to increase the chance of flakiness due to one specific root cause. By comparing the prevalence of flaky behavior between these containers, the root cause can be predicted. This approach is necessary if their assumption, that code instrumentation might prevent the manifestation of flakiness, is true. Unfortunately, setting up containers for every known root cause could be prohibitively complicated, and rerunning flaky tests in every container is very costly.

Ziftci et al. [13] show that code instrumentation does not always prevent the manifestation of flakiness. To locate the cause of flakiness, they store the code coverage for all test executions. If flakiness (at least one passing and one failing run) is detected, the first divergence of the stored code coverage recordings is determined. With 82% accuracy, this location of divergence is also the location for the cause of flakiness. Cases where the flakiness was introduced earlier (for example as a value of a variable), without diverging code coverage data, are problematic.

Collecting even more information about the execution, Lam et al. [5] store timestamps, code positions, caller information, and return values for function calls in the test execution. Similarly evaluated, the first divergence between passing and failing runs now contains more than just the position in a code file. This helps to debug even the most difficult cases: First, differing return values that are stored in variables are detected, without having to wait for diverging branches. Second, *Concurrency*-related flakiness might have the same code coverage in passing and failing runs, but the recorded sequence of function calls differs. Unfortunately, with increasing runtime overhead for the instrumentation, the number of reproducible flaky tests decreases.

For OD flaky tests, Shi et al. proposed iFixFlakies, a tool that fixes flakiness automatically [12]. For this, the tool first identifies tests that set or reset a required state. Then, it runs these identified tests before any flaky tests. Although this approach works for more than half of the examined tests, it is not flawless. If an OD flaky test does not depend on helper functions or they cannot be identified due to the complexity of the shared resources, iFixFlakies is helpless.

To better track shared resources of OD flaky tests and identify root causes of NOD flaky tests using machine learning, Roland Würsching carried out a master's thesis at our chair [15]. He analyzed system calls during test executions, which especially helped with detecting shared external resources of OD flaky tests. While some root causes of NOD flakiness (e.g.

Network or *Concurrency*) rely on certain system calls (e.g. `connect` or `fork` respectively), others (e.g. *Randomness* in Python) introduce non-determinism without any corresponding functionality of the operating system. A general problem is the small size of existing labeled datasets, which impedes the use of machine learning.

3.3. Research Gap

In contrast to OD flakiness, there is no efficient way to detect or fix NOD flaky tests [9, 12]. Although there are specific methods to expedite the detection for single root causes [10, 4], NOD flaky tests require generally more reruns [8]. Predicting flakiness is too unreliable to replace detection [14]. But with high enough recall, a good prediction might exceed humans in choosing which tests to rerun.

The origin of flakiness is often found in the CUT [2]. This explains why prediction based only on the test code alone (without the CUT) is insufficient due to low recall [14]. At the same time, tracing function calls (including the CUT) allows determining the root cause of flakiness with an accuracy of 90% [5]. The problem that tracing function calls might affect the reproducibility of flakiness does not have to be an obstacle: If we can select possibly flaky tests efficiently based on the function calls of a single run, rerunning them can happen without any tracing.

In this work we examine, whether function calls can predict NOD flakiness well enough, to select tests that should be further checked for flakiness. If this is the case, the total number of necessary reruns and the cost of flakiness detection could be lowered for NOD flaky tests, independently of their root cause.

4. Flaky Test Detection Using Function Call Tracing

We study whether we can reduce the effort of NOD flaky test detection by tracing function calls. Our approach is to only rerun tests, that execute functions that are characteristic for NOD flakiness. Therefore, we first have to gather the necessary information. This information includes the function call tracing and details about the tests, including their flakiness. In addition to knowing whether a test is flaky, we are also interested in why. First, the category of flakiness is important, since we focus on NOD flaky tests, but also the specific root causes of flakiness are interesting:
Common occurrences of particular root causes with certain function calls suggest that these functions may be related to the flakiness of the tests. Thus, by examining the co-occurrence of function calls and root causes, we can find functions that characterize NOD flakiness. After analyzing this strategy and other ways, to obtain function calls related to flakiness, we evaluate the usefulness of the function calls for flaky test detection.

Given a set of functions that characterize NOD flaky tests (e.g. `bar()` and `baz()` in Figure 4.1), we assume that tests that do not execute any of these functions (blue dots in Figure 4.1) are most probably not flaky. Hence, instead of rerunning all tests to detect NOD flakiness among them, it would be sufficient to only check tests that execute at least one of the characteristic functions (red and black dots in Figure 4.1). If many tests do not contain any of these functions, we only have to rerun a fraction of all tests which reduces the effort of flakiness detection. And if the set of characteristic functions is large enough, we do not miss any of the flaky tests (red dots in Figure 4.1). Using a realistic statistical model and the real runtimes of the test executions, we can assess how cost-effective our approach for detecting NOD flaky tests is.

The complexity of our approach forces us to choose one programming language to carry out our study. First, the requirements for existing data and detection approaches for comparison restrict our choice for a programming language. Second, the language-specific function call tracing and analysis would require additional effort for all considered programming languages. Finally, due to known differences in flakiness between different programming languages [8], one cannot expect that the findings of one programming language to apply to others. An extensive dataset with a tool to rerun tests [8] (useful for tracing) and an existing study about flaky test detection using reruns (for comparison and evaluation) [8] motivate us to choose Python for our study.

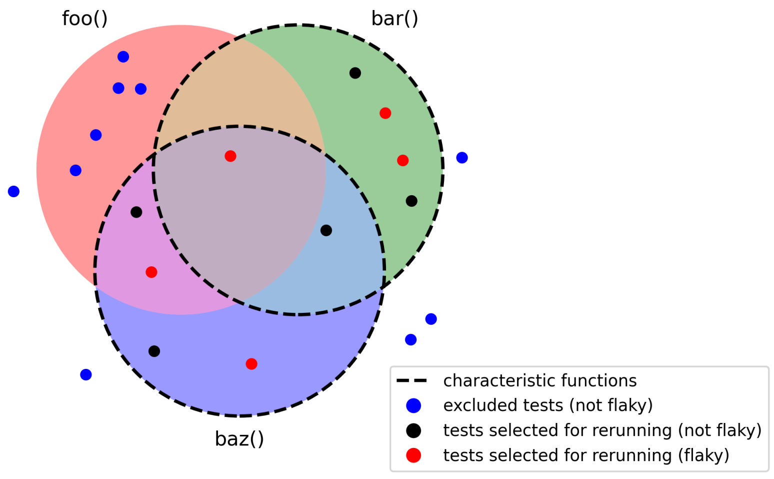
foo()
bar()
baz()
characteristic functions
excluded tests (not flaky)
tests selected for rerunning (not flaky)
tests selected for rerunning (flaky)

Figure 4.1.: Exemplary assessment of tests allocated to the partly characteristic functions they execute

4.1. Collecting the Required Data

To find a connection between specific function calls by a test and its flakiness, we need to trace function calls of test executions and know about their flakiness. Since some approaches to obtain functions that are characteristic for flakiness rely on the root causes of the flakiness, we need NOD flaky tests with labeled root causes. To assess our approach, we require the runtimes of the tests, and the necessary effort of the previous detection approaches.

4.1.1. Existing Dataset

We have various requirements for our dataset. First, it has to be large enough to allow for meaningful results. Then, it has to contain flaky and non-flaky tests that are reliably labeled as OD, NOD or non-flaky tests. The root causes of NOD flaky tests have to be stated (at least partly), and the original detection effort has to be statistically evaluable for comparison. Furthermore, the considered tests have to be freely accessible and executable.

Most datasets do not offer labeled root causes, either because they were not determined (for example in the International Dataset of Flaky Tests [9]) or determined but not published [2]. Lam et al. [5] performed manual root causing on Microsoft internal tests. While they provided the results of the root causing and the execution logs, we do not have access to the tests and therefore cannot run and trace them. The only dataset that meets our requirements was created by Gruber et al. [8].

They evaluated the necessary reruns for detecting flaky tests in Python [8]. For this they executed 400 reruns, which is more than often seen for flaky test detection [2]. Hence, we assume that the labeling is mostly reliable. Given the number of passed and failed runs, we can use their approach as a baseline in different scenarios. Since the 22,352 considered projects and *FlaPy*, the tool used to execute the tests, are all open source, we can access and run the tests freely. These projects include a total of 876,186 analyzed tests, a sufficient basis to provide meaningful results.

The root causes of 100 NOD flaky tests from the dataset were manually labeled by Gruber et al. [8]. In the course of Würsching's master's thesis [15], the root cause labels were verified and extended by 47 tests.

Due to limited resources, we could not analyze all 22,352 projects with our approaches. Therefore, we limit ourselves to a subset of the dataset. Similar to Bell et al. [25], we choose to only consider projects, that include at least one relevant flaky test. Since we study the detection of NOD flakiness, we only examine the 276 projects with NOD flaky tests. This choice is justified in different ways.
On the one side, it is harder to detect the NOD flaky tests if all considered tests are more similar (concerning their function calls). Analyzing tests from the same projects allows us to evaluate, whether the function calls can help with detecting flaky tests. Otherwise, our approach could perform well, by just distinguishing projects with NOD flakiness from projects without. Without these higher requirements, a promising result would not be sufficient for our approach to be recommendable.
On the other side, from a practical point of view, flakiness detection is more important for teams and projects that have experienced flakiness. The perception, that flakiness is a problem, increases with the occurrence of flakiness [3]. And even teams that have experience with flakiness often only check tests that are known or expected to be possibly flaky [2]. Thus, our evaluation is more relevant in practice, when only considering projects that contain NOD flakiness.
Otherwise, our approach would probably perform better but would be harder to recommend, due to a favorable assessment.

4.1.2. Tracing Function Calls during Test Executions

In this section, we present how we trace the test executions of the previously chosen projects. Therefore we extend a tool provided by Gruber et al. to execute the tests automatically [8]. Since all tests are executed using pytest, we implement a pytest plugin that traces test executions efficiently. When parsing and exporting the traces, we make sure that their format resembles the existing dataset to simplify further processing.

FlaPy is the tool created by Gruber et al. [8] and was used to generate the chosen dataset (see Section 4.1.1). It downloads projects and their dependencies automatically and executes their test suites in isolation. Moreover, FlaPy offers different parsers to evaluate and format the results of the executions.

We extend FlaPy's functionality to trace all function calls during test executions efficiently. To run FlaPy, projects are specified in a CSV file. If a column named `FUNCS_TO_TRACE` is set to `all` for a project, the function tracer should be activated. Furthermore, we implement another parser to format all generated traces and store them as one file.

We implemented the function tracer as a pytest plugin. After FlaPy downloads a project that should be traced, it appends the tracer plugin to the `pytest_plugins` variable, which is one of the ways to load a new plugin at the startup of pytest [19]. The tracer plugin consists of three main parts, the *test preparation*, the *function tracing* and the *result storing*, which are schematically presented in Figure 4.2.

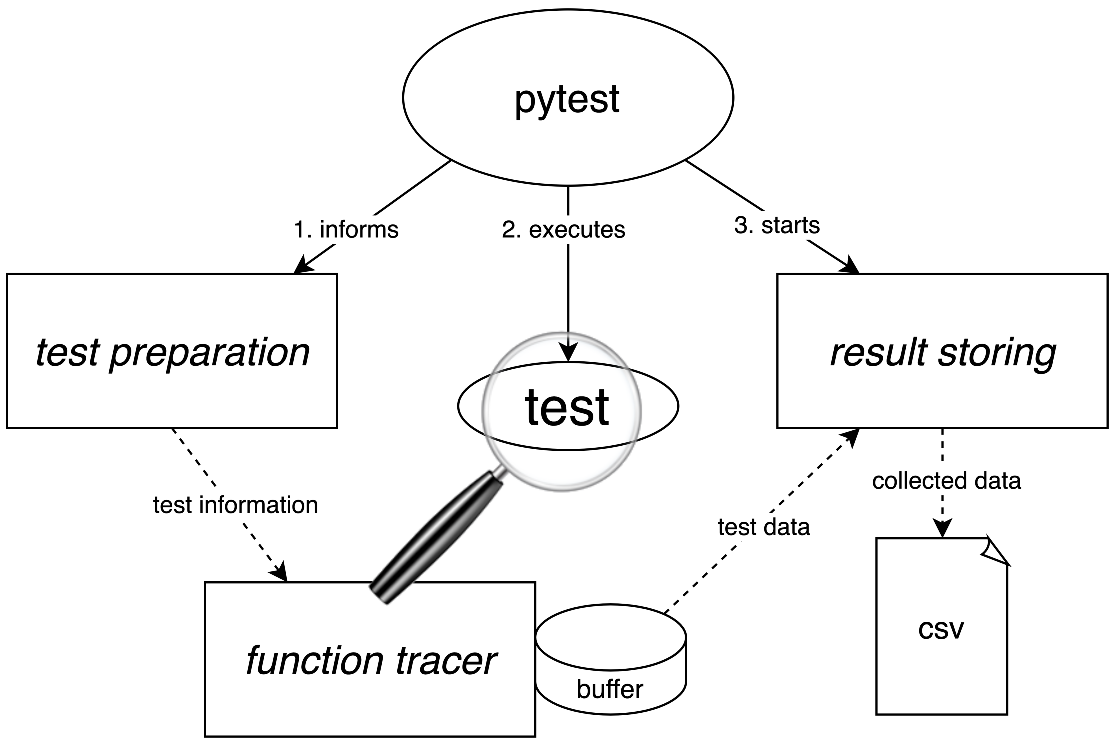

Figure 4.2.: Overview of the function call tracing procedure

The *test preparation* system collects identifying information for the test and activates the *function tracer* before every test. It is implemented as the `pytest_runtest_call(item)` function of the plugin, which is called for every test, with `item` storing information about the following test: Like FlaPy, we extract the file name, class name (if it exists), test function, and the parametrization of the test, to identify individual executions. In addition, we set the *function tracer* as a profiler (see Section 2.2.2), to be executed at every function call and return.

The *function tracer* traces all function calls (python functions and C functions, see Section 2.2.2) of the test execution but excludes the function calls caused by pytest. Thus, the *function tracer* only stores function calls that happen after the call of the test function and before the return of the test function. Since the name of the test function was already obtained during *test preparation*, comparing it with the name of called and returned functions is sufficient for this task. Between the call and return of the test functions, all calls of Python functions and C functions are parsed to determine the function name, class name (if it exists), and file name (only feasible for Python functions) to identify them. Multiple calls to the same functions are stored as one entry with a counter for the number of calls.

During *result storing*, a CSV file for the tracer output is created that enables easy further processing. Every function call entry, together with its number of calls and the identifying test information (obtained during *test preparation*), is written to this file. The file is stored in the same folder as FlaPy's own results and added to the list of files that FlaPy copies from the isolated execution environment to its results folder. Since repeating function calls from the same tests are aggregated and no unnecessary data is stored, the resulting file is fairly space efficient.

We just need one test execution to trace the function calls, since both passing and failing runs execute the functions that are related to flakiness [5]. For executing FlaPy with the tracer plugin for all chosen projects in our dataset, we optimize the plugin further. Testing on a modified version of the FlaPy example test suite, we reduce the function calls of the plugin, the cost of data structure accesses, and the necessary evaluations of conditions for the function tracer. Measured against 100 executions of the test suite, the time overhead generated by our tracer plugin is lowered to 36.1%.

The execution time of the parser for the function call traces is negligibly small. Given a results folder, it iterates through all projects to format and combine the CSV files created by the tracer plugin. The usage, approach, and created output are kept like the original FlaPy parser. This allows for easy combining of the tracer results with the existing dataset for further processing. Even for all examined projects, the performance of the parser is sufficient and no further optimization is necessary.

4.1.3. Obtaining the Runtime of Test Executions

FlaPy records and stores the runtimes of test executions in milliseconds, but offers no parser to obtain them in a usable format. Thus, we implement a simple parser that extracts test-identifying information (see Section 4.1.2) and the time required to run the test. Again, the usage and created output are kept like the original FlaPy parser. Therefore, we can add the runtime to datasets, whenever we need it. This way, a quantified evaluation is possible, and we do not have to rely on rough estimates.

Due to limited resources, we use a one-time value instead of an average of multiple runs, and measure while tracing the function calls. As previously shown (see Section 4.1.2), our function call tracer does not ruin the runtime measurements, and we assume the overhead of 36.1% to increase the runtimes proportionally. Furthermore, FlaPy always generates coverage data for test executions. This will also increase the runtime, but again we assume that all tests are affected equally. The number of projects makes it infeasible for us to execute all test suites without using FlaPy. Therefore, we have to accept these constraints for the obtained runtimes.

4.2. Identifying Functions that Characterize Test Flakiness

Our approach is based on the assumption, that specific functions characterize test flakiness. A similar idea has already been used to differentiate between root causes of flakiness based on executed system calls [15]. Since system call tracing only detects the usage of functionality offered by the operating system, not all root causes could be recognized equally well [15]. For accurate NOD flakiness detection, it is important to support all root causes of NOD flaky tests. Therefore, we do not trace system calls, but all function calls during the test execution.

We suppose that a set of functions, we call them *characteristic functions*, exists, such that most NOD flaky tests call at least one of them during the execution of the test. Since the first difference between function call traces of passing and failing runs often is the location, where test flakiness can be fixed [13], the flakiness seems to depend on the functions used by a test. If we find a strong connection between test flakiness and called (characteristic) functions, we can assume that tests that do not call any of these characteristic functions most probably are not flaky. Knowing that a test most probably is not flaky, we could skip it when detecting flakiness using reruns and therefore save resources, without missing flaky tests (see Figure 4.1).

As the success of our approach depends on the right choice of characteristic functions, we will perform and evaluate different methods for selecting them. First, we will choose characteristic functions manually. Second, we use an automatic optimizer to select characteristic functions based on root cause labels. Finally, we let the automatic optimizer pick a set of characteristic functions without manual preparatory work. Our goal is a small set of functions, that occurs in as many NOD flaky tests as possible without being widely used in non-flaky tests.

4.2.1. Selecting Characteristic Functions Manually

The manual selection of characteristic functions serves as a baseline for comparisons with automatically generated results. We use existing literature, the executed code (Test Code, CUT and information about library functions), as well as the dataset (see Section 4.1.1) with the results of our function call tracer (see Section 4.1.2). The advantage of selecting manually is that one can control the whole selection process and actively counteract potential problems.

One of these problems is overfitting, which happens for automatic selections if, for example, too few tests are labeled with a specific root cause and therefore their traces are not fully representative for functions of that root cause. This can be solved manually by searching for more general sub-functions if one is aware of this issue.
For example, all flaky tests labeled with *Concurrency* as their root cause may use the `start` function of the `Thread` module. Therefore, `Thread.start` could be a sensible pick for an automatic selection based on root causes. When selecting manually, we know that there are other ways to start threads and induce flakiness due to *Concurrency*. Thus, we pick the C function `start_new_thread` that is called not only by the `start` function of the `Thread` module but also used by other tests that could be flaky due to *Concurrency*.

To best understand these subtleties, we have to review previous research [8, 2] to understand root causes and their classification. Furthermore, previous flakiness prediction approaches [14] show that focusing only on the test code is insufficient. Literature about fixing flakiness [13, 5] helps us to understand, where to look for flakiness-inducing functions.

Similar to the root cause labeling performed by others [8, 15], we examine the test code and CUT to identify flakiness-causing functions. Given the function call traces, we can spot similarities for different tests that are flaky due to the same root cause. Using the documentation and implementation of project and library functions, we search for underlying functions that are called by multiple similar flakiness-causing functions. For tests of specific root causes, NOD flaky tests, and all traced tests, we determine the respective proportions that use these underlying functions. Thereby we can confirm or disprove our assumptions about certain functions indicating flakiness and make better choices in cases of ambiguity.

4.2.2. Automatic Selection Optimizer

For our automatic selection, we decide to use a greedy algorithm. Since our manual procedure was mostly greedy (in terms of selecting functions successively) and yielded useful results, this seemed more promising than some alternatives: We have too little labeled data for machine learning approaches but too many traced functions for computing the optimal subset of characteristic functions. Our optimizer starts with an empty set and adds functions (from a freely selectable set of functions) one after another (see Algorithm 1). Each time, the function that maximizes a given objective function is added, until no improvement (a higher value of the objective function) is possible. Given an objective function that quantifies the correlation between a set of functions and a set of tests, we can optimize characteristic functions for NOD flaky tests using this optimizer.

The better a given set of functions C characterizes a given set of tests T, the higher the value of our objective function applied to C and T should be. We say, functions C characterize tests T well, if all tests in T call a function in C, while all tests that call a function in C are

Algorithm 1 Automatic Selection Optimizer

```
 Input: Function set $F$, Objective function $O$, Test Set $T$
 Output: Set of characteristic functions $C$
$C \leftarrow \emptyset$
$max \leftarrow O(C, T)$
$new \leftarrow \emptyset$
repeat
  $C \leftarrow C \cup new$
  for all $f \in F$ do
    if $O(C \cup \{f\}, T) > max$ then
      $max \leftarrow O(C \cup \{f\}, T)$
      $new \leftarrow \{f\}$
    end if
  end for
until $O(C, T) = max$
```

part of T. We construct a binary classifier based on the functions C that classifies tests as flaky if and only if their function call trace includes at least one member of C. Thus, functions C characterize the flaky tests T well, if the corresponding classifier has the maximum recall (percentage of flaky tests that are identified) and the maximum precision (percentage of flaky tests among all identified ones). Our objective function should be a combined metric for the quality of the binary classifier, like Matthews Correlation Coefficient or the F-score.

Since we strongly prefer recall (finding many flaky tests) over precision (finding only flaky tests), we choose the generic F-score that allows modified weighting between recall and precision using a parameter β. Using this generic F-score as our objective function, we can choose β differently to specify, whether the resulting characteristic functions should identify many flaky tests but with more *false positives* (incorrectly identified tests), or if they should identify fewer flaky tests but with fewer *false positives*. While choosing $\beta = 1$ yields an insufficient recall, $\beta = 400$ leads to nearly every test being (mostly incorrectly) identified as potentially flaky. We will come back to a reasonable choice for β when deploying the optimizer.

4.2.3. Automatic Selection Based on Manually Labeled Root Causes

Functions that characterize different root causes are diverse. This assumption is reasonable, since executed system calls can be used to differentiate between root causes [15]. In addition to this, our insights gained during the manual selection (see Section 4.2.1) of characteristic functions confirm this theory; while function call traces of flaky tests with equal root causes resemble each other partly, most selected characteristic functions belong to only one root cause. To take advantage of this realization, we can deploy the optimizer (see Section 4.2.2) for all occurring root causes using the manually labeled tests.

After executing the optimizer for all occurring root causes, the union of the resulting function sets is considered as the set of characteristic functions. Although the principle is reminiscent of the manual selection, we have no direct control over the selected functions. Nevertheless, the division into root causes allows us to better comprehend the automatic selection retrospectively. Since we already dealt with the same function call traces extensively, a sensible automatic choice of characteristic functions should be recognizable. Unfortunately, we cannot study the consistency of the automatic selection based on labeled root causes, since the small number of tests with labeled root causes does not allow for cross-validation. We assume, that the findings about the consistency of the automatic selection of C functions (see Section 4.2.4) are meaningful for all applications of the optimizer. But the small number of root cause labels also has advantages: We can execute the optimizer on all traced functions of all labeled tests without having to manually choose a limited function set to lower the runtime.

Since we could not find any reliable information about the choice of the parameter β for the execution of the optimizer (see Section 4.2.2), we need to look into this in more depth. Although we know, that, for us, recall (finding many flaky tests) is more important than precision (finding only flaky tests), this preference is difficult to quantify. Therefore, we carry out the automatic selection based on labeled root causes for various β-values.
We want to find a value for β such that the number of correctly identified tests is similar compared to the manual selection. Only then, comparing the total number of tests that execute a characteristic function is fair. Initial tests have led us to believe that β-values around 10 are most reasonable. To understand the influence of the β-value, we execute the optimizer with β as all natural numbers up to 25 and all square numbers up to 400.

4.2.4. Automatic Selection of C Functions

Since the automatic selection of characteristic functions in the previous chapter still requires tests with manually labeled root causes, we test another approach to examine how useful a fully automatic selection of characteristic functions is. A good result would not only enable better selections based on larger datasets but also show that our approach is more transferable by automating selections of characteristic functions for other programming languages. Unfortunately, running the optimizer on all traced functions of all traced tests would be prohibitively expensive. Therefore, we have to find a superset of characteristic functions that is small enough to be optimized.

Choosing this superset manually would be possible, but it would also defeat its purpose since the process of obtaining characteristic functions would not be fully automatic anymore. To choose a set of functions heuristically, we divide all traced functions into broad categories. First, we identify project functions, that are introduced and implemented as part of the projects. Since their usage is inevitably limited to one project they cannot be characteristic. Next, we consider other Python functions, that are either built-in or part of imported modules. Although these include some functions that were previously chosen as characteristic, this set is too large and hard to narrow down further without loosing part of the possibly characteristic functions. Finally, we examine the set of functions that are implemented in C. Despite its small size, this set contains surprisingly many functions that were previously chosen as characteristic. Upon closer inspection, even some Python functions that were previously chosen as characteristic, call a C function that could replace them easily (e.g. `Condition.wait` calls the C function `allocate_lock`). Therefore, we choose it as the function set for the fully automated optimization.

Similar to the optimization based on manually labeled root causes (see Section 4.2.3), we vary the value of β to generate comparable results. Since a broad search for the correct value of β is infeasible due to the higher effort of optimizing on all examined tests, we use the insights gained during the automatic optimization on root causes to narrow down sensible values for β.

After identifying a suitable value for β, we want to study the consistency of the automatic selection. Therefore we use k-fold cross validation with $k = 5$ as commonly chosen in practice [26, 27]. This means, that we randomly assign all tests to 5 equally sized groups. One by one, we use every group as a test set to assess characteristic functions obtained by an automatic optimization on the union of all *other* groups. Despite the relatively small number of tests, we can thus perform 5 different optimizations on 80% of the NOD flaky tests each. The differences in the results of these optimizations provide information about the reliability of the automatic optimization.

4.3. Flakiness Detection using Characteristic Functions

Our goal for choosing characteristic functions was that every NOD flaky test calls at least one characteristic function during its execution. It is not ensured that only NOD flaky tests call characteristic functions, so checking the existence of characteristic functions does not suffice for flakiness detection. However, reaching our goal would imply that tests without characteristic functions in their execution cannot be flaky. Thus, we could use the conventional method of rerunning tests to detect flakiness, but reduce the necessary effort by omitting tests that do not execute any characteristic function.

In reality, not all NOD flaky tests will call a characteristic function. Hence, we will miss some flaky tests when rerunning only tests that call a characteristic function. Then again, due to the non-deterministic manner of flaky tests, rerunning all tests a finite number of times is also not expected to detect all flaky tests [8]. Given a specified execution time, we expect our approach to detect more flaky tests compared to rerunning all tests:
If the proportion of tests that do not call any characteristic function is sufficiently large, we can rerun the remaining tests, that are more likely to be flaky, a higher number of times. The increased probability of detecting flaky tests due to the increased number of reruns [8] should counteract missing a few tests whose execution lacks a characteristic function.

To verify this assumption we need a way to realistically compare our approach with the baseline; simply rerunning all tests, as usually done to detect as many NOD flaky tests as possible [8]. Solely calculating the expected number of detected tests when rerunning them a given number of times does not satisfy our demands for realism: The maximum number of reruns is only an upper bound for the detection cost since in reality rerunning stops as soon as two different outcomes have occurred [25]. Thus we need a statistical model that depicts all characteristics of flakiness detection realistically.

4.3.1. Statistical Model to Recreate Flaky Test Detection

During rerunning, a test is only said to be flaky if a passing and a failing execution occurs. Therefore, no false positives (non-flaky tests claimed to be flaky) are possible. The efficiency of rerunning a given set of tests to detect flakiness therefore only depends on the number of detected flaky tests and the cost of execution. Both the expected number of detected flaky tests and the expected cost of execution can be calculated using the previously obtained data.

First, we require the probability $U_t(n)$ for unveiling flakiness after n reruns for a test t. In accordance with Gruber et al., we assume that different runs are independent from each other [8]. Given the probability $P_t(pass)$ that t passes and the probability $P_t(fail)$ that t fails, we can calculate $U_t(n)$ [8]:

$$U_t(n) = 1 - (1 - P_t(fail))^n - (1 - P_t(pass))^n + (1 - P_t(pass) - P_t(fail))^n$$

This is equivalent to the probability, that the test does not always fail ($(1 - P_t(fail))^n$) or pass ($(1 - P_t(pass))^n$) or is not skipped consistently ($(1 - P_t(pass) - P_t(fail))^n$) [8].

Based on the independence assumption, we can calculate $P_t(pass)$ (and $P_t(fail)$) by dividing the number of passed (or failed) runs by the number of all runs.

To calculate the expected number of detected flaky tests, let $F_T(n)$ be the random variable for the number of detected flaky tests in a test set T after n reruns each. Let indicator variable $X_t(n)$ describe that test t was detected as flaky after n reruns. $X_t(n)$ is Bernoulli distributed with $U_t(n)$ as its probability of success. Since we consider NOD flaky tests, that (per Definition 1) are flaky without changes in the execution environment, we can assume independence of $X_{t_1}(n)$ and $X_{t_2}(n)$ for $t_1, t_2 \in T$ if $t_1 \neq t_2$. Thus, the expected number of detected flaky tests $\mathbb{E}[F_T(n)]$ can be traced back to the individual probabilities for unveiling flakiness that we already know:

$$\mathbb{E}[F_T(n)] = \mathbb{E}[\sum_{t \in T} X_t(n)] = \sum_{t \in T} \mathbb{E}[X_t(n)] = \sum_{t \in T} U_t(n)$$

This yields the expected number of detected flaky tests after n reruns. But in reality, n is just the maximum number of reruns, since the flakiness detection can be aborted after at least one failing and one passing run has occurred. Since we know the execution time of the tests in our dataset (see Section 4.1.3) and since the analysis of the test outcomes is negligibly small, we just need the expected number of reruns to calculate the resulting execution time.

Given a maximum number of reruns n, the real number of reruns $R_t(n)$ for test t is an integer random variable in the value range $W_R = [2, n] \cap \mathbb{N}$. For $x \in W_R \setminus \{n\}$, the probability $P[R_t(n) = x]$ that t is executed x times equals the probability that t showed flakiness after x runs, without being detectably flaky after $x - 1$ runs. If $x = n$, $P[R_t(n) = x]$ additionally includes the possibility that no flakiness was detected after n reruns. The standard definition of the expected value for random variables allows us to calculate the expected number of reruns:

$$\mathbb{E}[R_t(n)] = \sum_{x \in W_R} x \cdot P[R_t(n) = x] = (\sum_{x=2}^{n} x \cdot (U_t(x) - U_t(x-1))) + n \cdot (1 - U_t(n))$$

For a test t, the expected execution time of flakiness detection is the product of the expected number of reruns and the runtime of t. Given a test set T the expected execution time of detecting flakiness for all tests in T is the sum of the expected execution times for all individual tests in T.

Based on previously available information, we can calculate the expected number of detected flaky tests, the expected number of reruns, and the expected execution time for different test sets. As previously pointed out, this allows us to quantify the efficiency of flakiness detection and compare different approaches conclusively and fairly. By varying the maximum number of executions, our model is suitable for a wide range of scenarios, from short flakiness checks to extensive analyses.

Thus, we created a model to evaluate the cost-effectiveness of our approach to detect NOD flakiness by using function call tracing.

5. Evaluation

We first assess characteristic functions and the effectiveness of approaches to select them. We further evaluate whether the characteristic functions can reduce effort for flaky test detection. For this, we start with establishing Research Questions (RQs) and describing our procedure to answer them. After presenting the results, causes and specifics that may serve as a good starting point for future work will be discussed.

5.1. Research Questions

For a clear and incontestable evaluation, we will define RQs that clarify the problems we wanted to solve. Similar to previous chapters, we will first consider the characteristic functions before assessing their applicability for flakiness detection.

RQ1: How well do certain functions characterize NOD flakiness?

Our approach is based on the well-founded assumption, that the occurrence of NOD flakiness is related to the execution of certain functions. If this correlation is as strong as the literature suggests, the usage of certain functions should be characteristic for NOD flaky tests. We have to obtain these supposedly characteristic functions before evaluating this correlation. For this, we consider varyingly automated approaches and their respective results individually. Besides the accuracy, which is relevant for all approaches that obtain characteristic functions, we ask ourselves whether highly automated approaches yield results that match previous understandings of flakiness.

- **RQ1.1:** How accurate are manually selected characteristic functions?
- **RQ1.2:** How accurate is the selection of automatic approaches?
- **RQ1.3:** How sensible is the selection of automatic approaches?

Since the approaches differ widely, evaluating their different selections of characteristic functions gives us a good idea about the capabilities of characteristic functions in general. Next, we want to investigate whether they should be applied to detect NOD flakiness in a realistic scenario.

RQ2: How much NOD flaky test detection effort can be saved by using characteristic functions?

Since we did not further define detection effort, we can answer this question in different ways. Both previous approaches and our approach ultimately rely on running tests multiple times. Thus, we can compare the number of reruns required for similar results, which is a metric used in literature to evaluate flakiness detection [8]. The uneven distribution of test sizes (see Section 5.2) suggests that the number of reruns may not be representative for the execution time. Therefore, the execution times of the approaches must be evaluated separately. It is also necessary to clarify how generally valid the findings are by conducting evaluations for individual projects.

- **RQ2.1:** How far can we reduce reruns?
- **RQ2.2:** How much time can we save?
- **RQ2.3:** How consistent is the performance for different projects?

Before we start with the evaluation, we first present the data used to answer these questions.

5.2. Study Subjects

5.2.1. Collecting Data for Study Subjects

In the provided dataset (see Section 4.1.1) some projects are erroneously counted twice[1] and some entries are irregularly formatted. While these problems could be fixed, the re-execution of the tests is the difficult part. Flakiness is known to be hard to reproduce on certain platforms [2], therefore we deliberately chose an evaluation approach where reproducing flakiness is not necessary. However, tests must be executed once to trace the function calls.

The reasons why we cannot trace certain projects are manifold. In several cases, the project repository either did not exist or was not publicly accessible anymore. As personally confirmed by the creator of FlaPy, the software used to run the tests and generate the dataset, BenchExec[2], the framework used to execute tests in isolation, lately leads to problems for multiple projects. Furthermore, some projects could not be executed, since external components are often required that may be incompatible with our local system or no longer accessible. Finally, tracing individual tests is not possible, if they are skipped or lead to an error since the execution and thus the tracing cannot be continued beyond that point.

As a result, 204 of the 276 projects that contain a NOD flaky test were traced successfully. This yielded 10,511 traced executions of which 543 belong to NOD flaky tests. Despite all obstacles, 57.04% of the NOD flaky tests were successfully traced, which is enough data to allow representative results. Concerning the root cause labels, no executions were traced for the categories *Resource Leak* and *Unordered Collection* that only contained 1 and 2 tests respectively. Since these are the two smallest categories besides *Floating Point*, we suspect that

[1]https://github.com/se2p/FlaPy/pull/1
[2]https://github.com/sosy-lab/benchexec

the performance of our approach does not depend on their absence. In the other categories, 69.28% of tests were traced successfully. Not surprisingly, this number is higher than the correctly traced share of all NOD flaky tests (57.04%), as Würsching et al. only labeled recently executable tests (as personally confirmed by the author).

5.2.2. Quantitative Classification of the Study Subjects

To familiarize ourselves with the dataset, we examine the distribution of test sizes. For our study, we consider the runtime of tests as their size, which makes sense for evaluating the cost-effectiveness of their executions. Figure 5.1 shows how many tests exist, that are quicker than a specified runtime. For instance, 95.32% of all tests were each executed in less than 1 second. That the tests are mostly small with regards to their runtime is compatible with previous studies, which found that PyPI projects generally tend to be small [28].

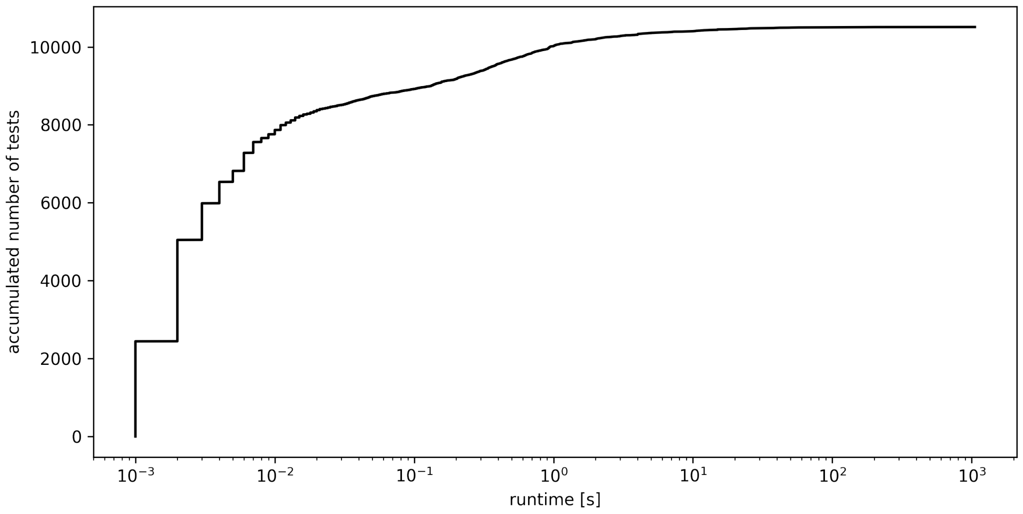

Figure 5.1.: Cumulative distribution of runtimes of successfully traced tests

Next, we wanted to study our function call traces. The overall number of function calls of a test in relation to the test size should provide information about whether the tracings are reliable. We also distinguish between NOD flaky and other tests to identify possible differences. The scatter plot (Figure 5.2) shows a consistent upper bound for the *number of function calls per second* of a test, as every function call requires some execution time. Thus contradicting results would indicate tracing errors. Furthermore, there is a clear correlation between the number of function calls and the runtime of a test, which is also expected. We do not recognize significant characteristics of NOD flaky tests.

We manually examined the distinct outliers (Figure 5.2) that show high runtimes despite

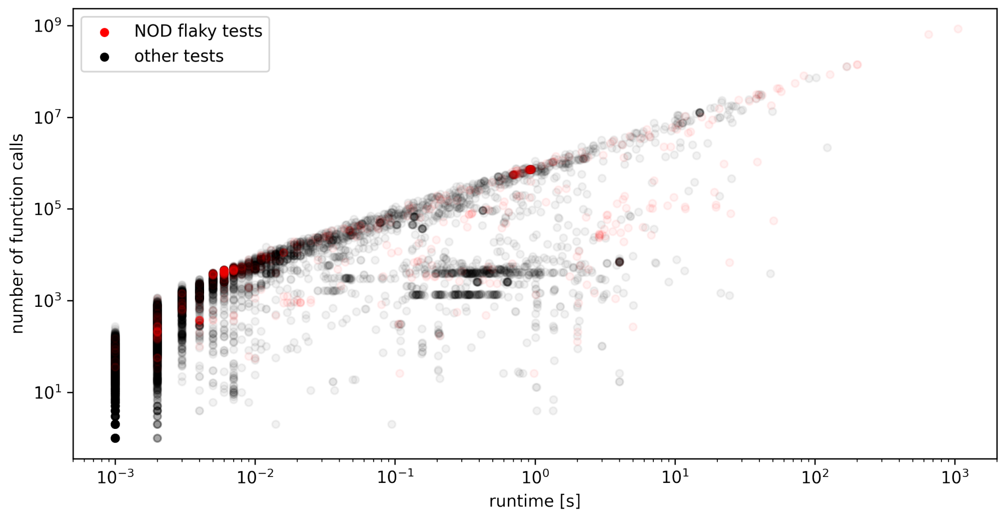

Figure 5.2.: Number of function calls of tests with their runtime

few function calls. The 3 tests with a runtime above 1 second despite less than 10 function calls all use the C function `_baseAssertEqual` for large Python objects. This shows a limitation of Python tracers that know about the call and return of C functions, but cannot further trace the execution in between, since the Python interpreter that enables tracing is not active during the execution of C functions. And the 6 tests that run for more than 10 seconds despite executing less than 10,000 function calls all start a child process that cannot be traced using profilers (see Section 2.2.2). All in all, these findings confirm known limitations but lead us to believe that no unknown problems occurred.

Previous research has exposed a correlation between flakiness and the size of the test code, as well as flakiness and used memory (RAM) by a test [29]. We use our data to check simple heuristics that might help to quickly encounter flaky tests, therefore we again regard the runtime of a test as its size. Since larger tests are more likely to be flaky [29], we investigate whether it makes sense to check them first. The number of all tests that have to be examined to find a certain number of NOD flaky tests is shown in Figure 5.3. This shows, that one could for example detect 85.8% (466 of 543) of the NOD flaky tests by only examining 40% (4204 of 10511) of all tests.

But even if flakiness was detected by executing a test only once, the efficiency of this approach would be poor. The sizes of the tests vary by several orders of magnitude (see Figure 5.1). The additional costs of running large tests first far exceed the increased probability of finding flakiness among them. Looking at the blue plot in Figure 5.4, it can be seen that it would take 92.5% (6201 of 6703 seconds) of the total runtime to encounter 20% (109 of 543) of the NOD flaky tests when starting with the largest tests.

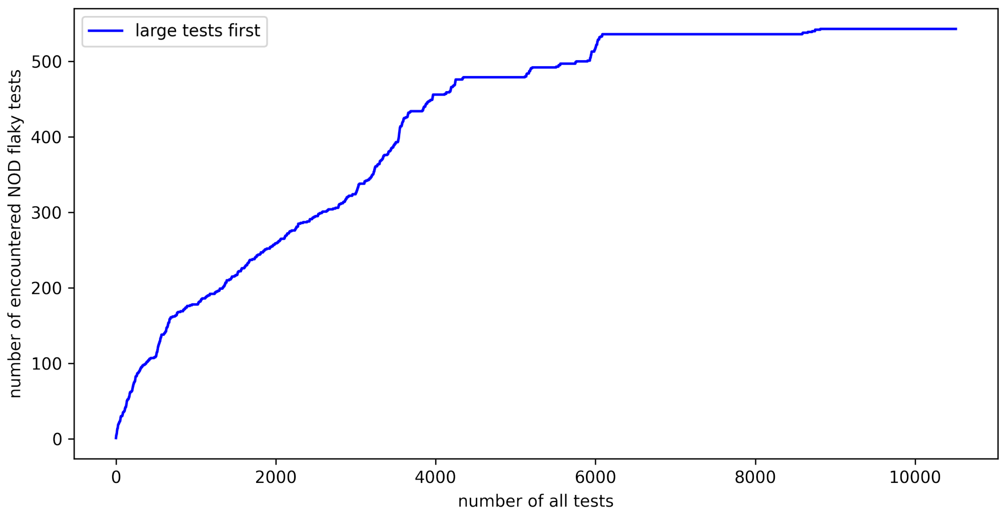

Figure 5.3.: Number of all tests in relation to the number of NOD flaky tests they contain when ordered by descending runtime

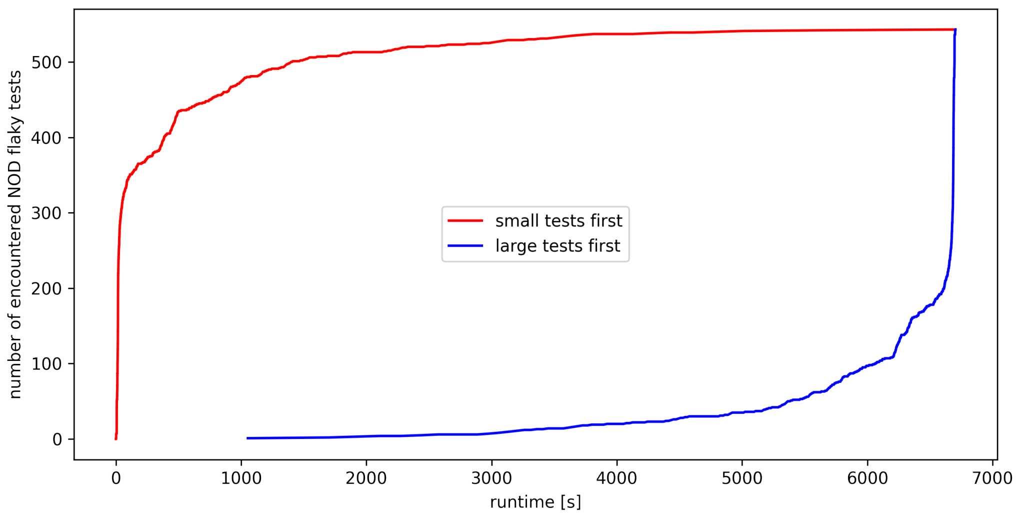

Figure 5.4.: Required runtime to execute all tests in relation to the number of NOD flaky tests they contain

Since this result is so devastating, we want to see if running small tests first makes sense to quickly encounter flaky tests. As illustrated by the red plot in Figure 5.4, we can encounter

80% (435 of 543) of the NOD flaky tests in only 7.5% (502 of 6703 seconds) of the total runtime when executing small tests first.
We will readdress these findings in the discussion (see Section 5.5.2).

Nevertheless, we will still use the rerunning of all tests as a baseline for multiple reasons: On the one side, this is the approach that is prevalent in research [8] and therefore yields a useful comparison. On the other side, we would first have to study the implications of categorically excluding large tests from flakiness detection, before considering it as a possible baseline.

5.3. Setup for Evaluation

In the following, we will show how we proceeded to answer the RQs unbiasedly and meaningfully. Like the RQs themselves, we will divide the approaches into obtaining characteristic functions and applying them for flakiness detection.

5.3.1. Evaluating Characteristic Functions

As explained in Section 5.1, we have to examine the different approaches to obtain characteristic functions, before a meaningful statement about the general quality of characteristic functions is possible. Establishing characteristic functions should be a one-time task, at least while the underlying programming language does not change. As long as applying the approaches is not prohibitively expensive, their costs are less important than the results. In our case, with only a few days of work for planning, implementation, and execution for every approach, the required resources are in the same reasonable order of magnitude.

To evaluate the results, the suggested sets of characteristic functions, we need to check how well they achieve their goal of exposing NOD flaky tests. At best, these functions should be executed by all NOD flaky tests. Thus, the simplest metric is the real proportion of NOD flaky tests that execute at least one of the suggested characteristic functions. This value was already introduced as the so-called *recall* of a set of characteristic functions (see Section 4.2.2). Since the set of all functions would expose all tests as possibly NOD flaky and therefore obtain a perfect recall of 100%, we require a second metric that assesses whether too many tests are exposed. As the second metric besides the recall, we choose the proportion of all tests that execute any of the characteristic functions. Comparing the proportion of NOD flaky tests to the proportion of all tests should provide clear and useful findings about the accuracy of a set of characteristic functions.

Especially for the automated approaches, we have to check whether the results are reliable. To evaluate the consistency of individual runs using cross-validation, we present the minimum, maximum, and average values of the accuracy metrics defined above. For functions to be meaningfully characteristic, there should be a connection to a general property of NOD flakiness. On the one hand, if functions expose NOD flaky tests in only one project, the associated property will probably not generally apply for NOD flaky tests. Thus, we count the occurrence of these not commonly characteristic functions, that only expose NOD flaky tests in at most one project. On the other hand, we manually check how many of the suggested characteristic functions are related to previous findings in research about flaky tests, to assess whether the results are representative for NOD flakiness. Examining these traits allows us to determine the quality of the obtained characteristic functions beyond the numerical metrics for accuracy.

After evaluating the results of all approaches to obtain characteristic functions, we summarize our findings to assess how well flakiness can be characterized by certain functions.

5.3.2. Evaluating Flaky Test Detection

To meaningfully evaluate our approach for flaky test detection, we compare it to a baseline; rerunning all tests multiple times and checking for varying outcomes. This baseline was presented by Gruber et al. [8] and used to create the dataset we use (see Section 4.1.1). Both our approach and the baseline ultimately rely on rerunning a certain set of tests (although for the baseline it is just the set of all tests). The statistical model can calculate the expected number of detections, reruns, and execution time (see Section 4.3.1) and works for both approaches. This yields a fair, extensive, and practice-oriented comparison without having to rely on non-deterministic simulations.

Apart from rerunning, there are overhead costs generated by our approach while specifying the set of tests that should be rerun. These are indispensable because we have to trace the function calls of one execution for every test, to determine whether a characteristic function is called. Since studies suggest that tracing, as a form of code instrumentation, may influence the flakiness of a test [11, 5], counting the traced execution as our first rerun could falsify the results of the statistical model. Thus, we have to add the overhead costs to the costs calculated by the statistical model. Fortunately, since tracing and storing all functions is slightly more complex than tracing and comparing them with a small set of characteristic functions, we have a narrow upper bound for this overhead; the execution times using our tracer.

If both the number of detected NOD flaky tests and the execution time differ between the approaches, we may face a trade-off between accuracy and execution time. This would make a clear comparison impossible, since choosing a *better* approach would depend on a subjective weighting between accuracy and execution time. By varying the maximum number of reruns for both approaches we can get multiple results that will be equally realistic and applicable. Consequently we can choose two results where the number of detected NOD flaky tests is similar, and obtain clear results by comparing the required reruns and execution times. As limits for varying the maximum number of reruns, we choose 2 and 170. 2 is the lowest number of reruns at which flakiness can be detected. 170 is the number of reruns specified by Gruber et al. to reliably check for NOD flaky tests in Python [8].

This allows us to not only choose which approach is *better* but also to assess how much they differ and how much effort can be reduced by choosing one over the other.
On the one hand, the number of successfully traced tests (see Section 5.2) should be sufficient to make generally valid statements.
On the other hand, we can figure out the *better* approach for all individual projects by evaluating only tests from one specific project at a time. Consequently, we can examine the consistency of our result by verifying whether the *better* approach would be preferable in most projects. For the flakiness detection on individual projects, we set the maximum number of reruns to 170 as suggested by Gruber et al. [8].

5.4. Results

To answer the RQs, we discuss the different results one by one.

5.4.1. Effectiveness of Characteristic Functions

The effectiveness of the different approaches is presented in Table 5.1. For clarity, the automated optimization of characteristic functions based on individual root causes and considering all executed functions is abbreviated as *Optimized on Root Causes*. Accordingly, *C Functions Optimizations* denotes the average values for the 5 automated optimizations of characteristic functions using cross-validation based on all existing tests considering only C functions as possibly characteristic.
The resulting characteristic functions of the manual selection, the C functions optimization, and the functions optimized on root causes are shown in the appendix (see Tables A.1, A.2, and A.3 respectively). The tests that are *exposed* by a set of characteristic functions are the ones that execute at least one of the characteristic functions and therefore are considered possibly flaky.

Table 5.1.: The accuracy of characteristic functions from different selection approaches.

Selection Approach	Exposed Share of NOD Flaky Tests	Exposed Share of All Tests
Manual Selection	92.27%	43.82%
Optimized on Root Causes	89.50%	41.75%
C Functions Optimizations	92.90%	41.46%

The manual selection of characteristic functions worked as expected for most Root Causes. However the `Random` and `RandomState` classes of the `NumPy`[3] module offer many C functions that are used by NOD flaky tests. Since we cannot trace internal procedures of C functions,

[3]https://numpy.org

we cannot identify common sub-functions that would normally be chosen as characteristic functions (see Section 4.2.1). In order not to miss any tests that are flaky due to *Randomness*, we considered all functions of the classes `Random` and `RandomState` as characteristic functions. For the sake of clarity, we combine these as `Random.*` and `RandomState.*`. Furthermore, we only had one test whose root cause was identified as *Floating Point*, which is insufficient to identify generally characteristic functions. We expect that most tests that flake due to *Floating Points* will still be detected, since the one we analyzed executed multiple other characteristic functions.

Answer to RQ1.1: 43.82% of all tests execute a manually selected characteristic function. Although these tests account for fewer than half of all tests, they still contain 92.27% of all NOD flaky tests. This high accuracy allows ignoring most of all tests without missing more than 7.73% of the NOD flaky tests.

To assess the accuracy of automatic approaches, we must first determine useful β-values for the F-score as the objective function for the optimizer. As explained in Section 4.2.3, we run the optimization on root causes for various β-values. The accuracy of the resulting sets of characteristic functions is depicted in Figure 5.5. Many β-values lead to similar and sometimes equal sets of characteristic functions. Nevertheless, we can control how widespread the characteristic functions should be among all tests by varying β. The correlation between the occurrence in all tests and the exposed NOD flaky tests can be seen in Figure 5.5. Using this chart, we can determine that $\beta = 24$ yields a similar number of exposed NOD flaky tests as the manual selection. Therefore we choose the optimization run with $\beta = 24$ to represent the optimization of characteristic functions on root causes.

We also need to choose a β-value to assess the accuracy of the automatic optimization of C functions. The approach is similar, but we search directly for the optimal β-value since another broad search is infeasible because of the more costly optimization. Using $\beta = 5$, the number of detected NOD flaky tests again is similar compared to the manual selection and the optimization run is therefore representative for comparison with the other approaches.

The values for accuracy for automatic approaches in Table 5.1 originate from these choices of β (24 and 5 respectively). Thus we can also answer RQ1.2.

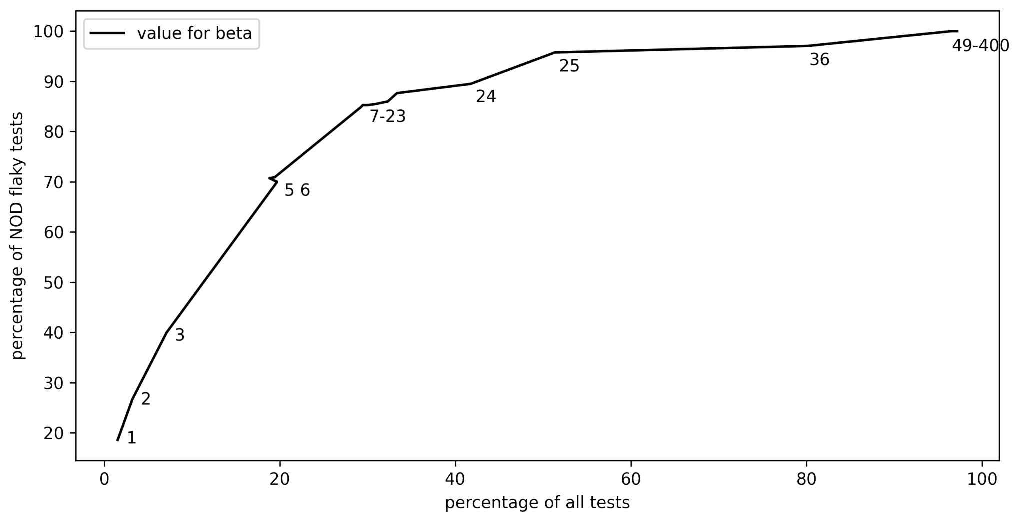

Figure 5.5.: Accuracy for sets of characteristic functions optimized on root causes for different β-values

Answer to RQ1.2: Despite the different automatic approaches, the accuracy of the resulting characteristic functions is very similar. Exposing 92.9% of the NOD flaky tests although only 41.46% of all tests contain their characteristic functions, the C function optimizations are slightly more accurate than the manual selection. The characteristic functions of the optimization on root causes occur in 41.75% of all tests and in 89.5% of the NOD flaky tests, which is slightly less accurate than the manual selection. As with the manual selection, the accuracy of both automatic approaches thus appears sufficient.

To evaluate how sensible it is to use an automatic optimization of characteristic functions, we check how consistent the results are. The 5-fold cross-validation for the optimization of C functions has resulted in 5 different sets of characteristic functions. The best, worst and average values of all 5 optimizations are given in Table 5.2. Due to the small number of tests in the test sets of the cross-validation, the visible variation was expected. However, the results are rather consistent and do not contain strong outliers.

Besides consistency, a sensible approach to obtain characteristic functions should also yield sensible results. Therefore, we analyze the properties of the characteristic functions in Table 5.3: To verify that the chosen characteristic functions are really *characteristic* for NOD flaky tests, we expect them to be useful in more than one project. For all approaches, we determine *Solo*, the number of characteristic functions that only expose NOD flaky tests in at most one project. In addition, we try to assign previously identified root causes [8] to the characteristic

Table 5.2.: The accuracy of the optimizations on C functions using cross-validation

Value of Cross Validation Runs	Exposed Share of NOD Flaky Tests	Exposed Share of All Tests
Minimum	88.18%	39.79%
Average	92.90%	41.46%
Maximum	95.45%	42.63%

functions, to check whether they correspond to previous findings about NOD flaky tests. The assignment can be viewed in the appendix (see Table A.4). *Cryptic* denotes the number of functions with no assignable root cause. We consider functions that are *Solo* or *Cryptic* to be not sensibly characteristic. While *Original* describes the number of all selected characteristic functions, *Remaining* is the number of characteristic functions that were not exposed as *Solo* or *Cryptic* (see Table 5.3). For the C function optimization using cross-validation, we again present the average value of all 5 optimizations.

Table 5.3.: Analysis of properties of different sets of characteristic functions

Selection Approach	Original	Solo	Cryptic	Remaining
Manual Selection	8*	0	0	8*
Optimized on Root Causes	16	8	7	5
C Functions Optimizations	19.2	5.2	6.2	9.6

*The entries `Random.*` and `RandomState.*` count as one item each

After removing the functions denoted as *Solo* and *Cryptic*, the remaining characteristic functions of the optimization on the root causes only expose 10.7% of all tests containing 27.4% of NOD flaky tests. While this accuracy is clearly insufficient, the remaining functions of the C function optimization still expose 38.2% of all tests containing 91.23% of the NOD flaky tests on average. As expected from the previously verified consistency, there are no strong outliers for these values either.

Answer to RQ1.3: For the root cause optimization, 11 of the 16 functions are either *Solo* or *Cryptic* (or both), therefore not really characteristic. The set of the 5 remaining functions exposes too few NOD flaky tests to be sensibly called characteristic.
Optimizing C functions produces 50% *Solo* and/or *Cryptic* functions, but the (on average) 9.6 remaining characteristic functions are a sensible choice for characteristic functions. They cover various root causes and achieve a similar accuracy compared to the manual selection. Furthermore, this approach is consistent for different executions and thus recommendable to obtain characteristic functions.

For further analysis, we choose one of the sets of characteristic functions that the different approaches generated. Since we do not want to jeopardize our result using inexplicably selected functions, we leave out functions exposed as *Solo* or *Cryptic*. The sets of remaining functions each expose slightly fewer NOD flaky tests than the manually selected set of characteristic functions. Thus, we decide to further investigate the manually selected characteristic functions.

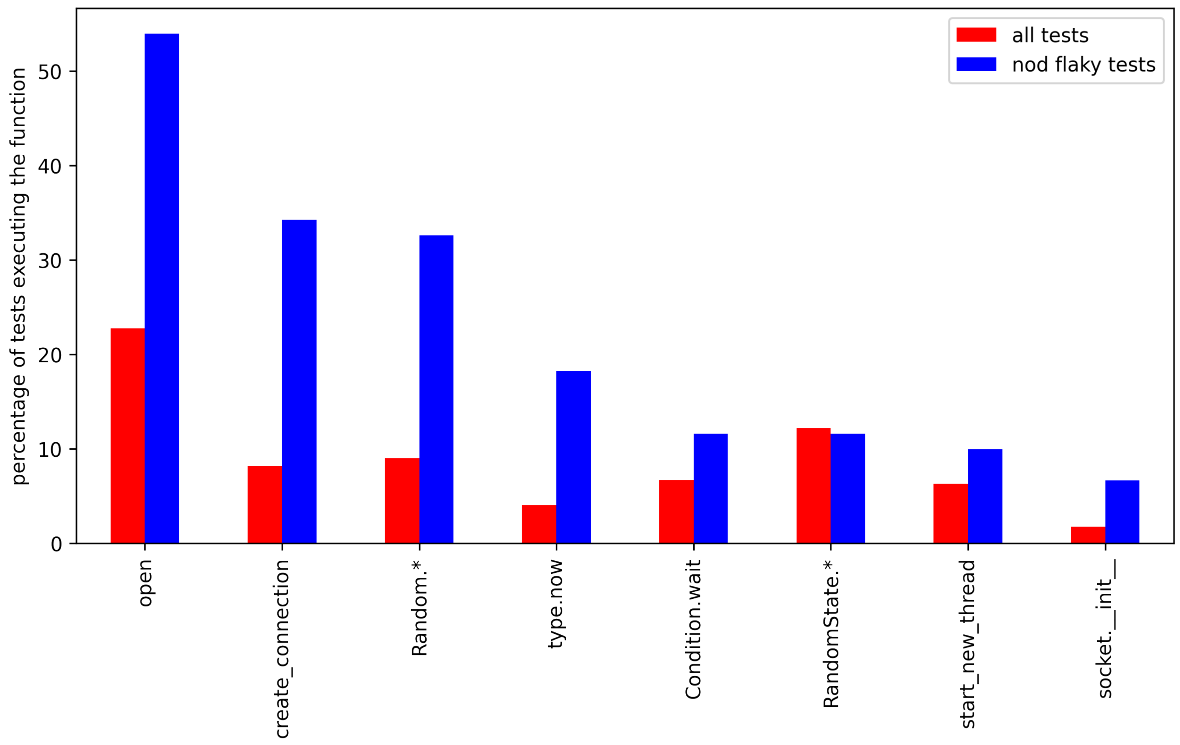

Figure 5.6.: Usage comparison of characteristic functions between NOD flaky tests and all tests

Figure 5.6 shows the proportion of NOD flaky tests and the proportion of all tests that execute each characteristic function. All but one are proportionately used significantly more often in NOD flaky tests than in all tests. This confirms that they are a reasonable part of the characteristic functions. The functions in the `RandomState` class are the only entry that exists more often in all tests than in NOD flaky tests. Objects of the class `RandomState` are independent random generators, normally used with seeds [30]. Thus, in most cases, `RandomState` behaves deterministically and does not cause flakiness. But sometimes it *does* cause flakiness since `RandomState` uses a random seed provided by the operating system if none is specified otherwise. So if we remove the `RandomState` functions from our set of characteristic functions, we would detect 8.48% fewer NOD flaky tests. Hence, the entire manual selection of characteristic functions is well justified.

Answer to RQ1:
Functions clearly related to root causes of flakiness exist and commonly appear in NOD flaky tests. More than 90% of the NOD flaky tests execute such functions, but they occur in fewer than half of all tests. The probability of hitting an NOD flaky tests doubles, when only considering tests that execute characteristic functions instead of all tests.

To obtain characteristic functions, both manual and semi-automated approaches lead to good results. Fully automated processes select a superset of the characteristic functions and are therefore not sufficient on their own.

5.4.2. Performance of Flaky Test Detection using Characteristic Functions

Applying the statistical model for flaky test detection to both the baseline and our approach for different numbers of maximum runs, we obtain Figure 5.7. The only case where the baseline performs better is when the maximum number of reruns is 2 and thus the additional executions for function call tracing are obstructive. In any other case, we need far fewer reruns to detect the same number of flaky tests (47% fewer reruns for detecting 67% of the NOD flaky tests). Since a few NOD flaky tests do not execute any characteristic functions (see the accuracy of characteristic functions in Table 5.1), they can only be detected by the baseline approach. Thus, the baseline can surpass the maximum number of detected NOD flaky tests by our approach, but only after executing more than twice the reruns.

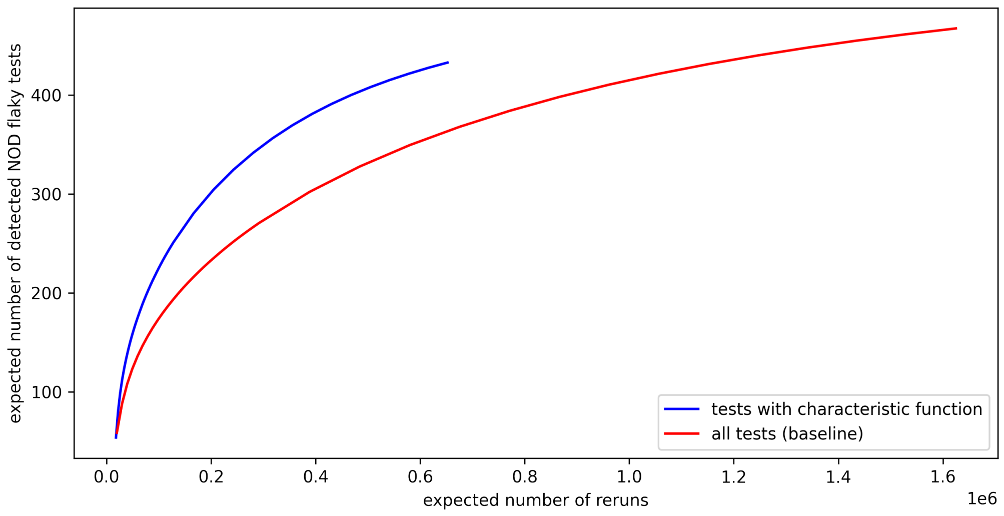

Figure 5.7.: Detected NOD flaky tests in relation to the required number of reruns

Answer to RQ2.1: Apart from edge cases, our approach requires about half the reruns to produce results similar to the baseline. Tracing function calls for flaky test detection is worthwhile (for the number of reruns) whenever tests are executed more than twice. Although our approach misses a few tests, the number of reruns necessary to make this effect noticeable is much larger than usually chosen in practice [2].

Since we obtained the runtimes of the tests, our statistical model provides not only the number of reruns but also the required execution times for the approaches as seen in Figure 5.8. Rerunning only tests with characteristic functions performs worse than the baseline. While it takes 47% fewer reruns to detect 67% of the NOD flaky tests, the execution time is 12% higher compared to the baseline. Although the execution times are not vastly different, our approach is permanently slower and still has the disadvantage of missing tests without characteristic functions. We will further discuss the reasons for this result and the following conclusions in Section 5.5.2.

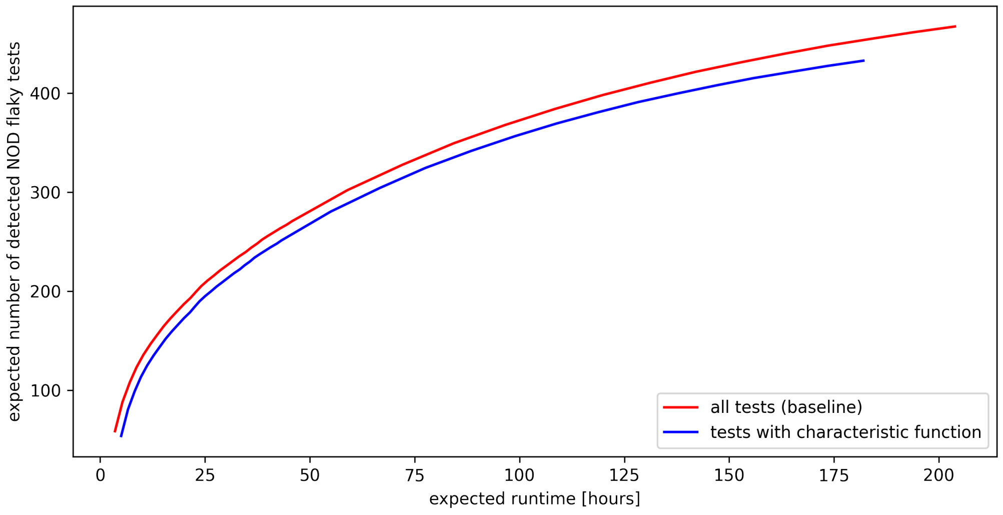

Figure 5.8.: Detected NOD flaky tests in relation to the required execution time

Answer to RQ2.2: Performing flaky test detection by using characteristic functions does not save time. While the difference between our approach and the baseline is rather small (less than 20%, apart from boundary values), the baseline is universally preferable. Even if we evaluate the runtimes without the overhead of the function call tracer, our approach would still never beat the baseline.

This result shows, that our approach should be advised against when trying to detect flaky tests among our dataset. How generally valid this advice is can only be verified by running the evaluation on different datasets. Looking at each project as a single dataset, we get the following results:

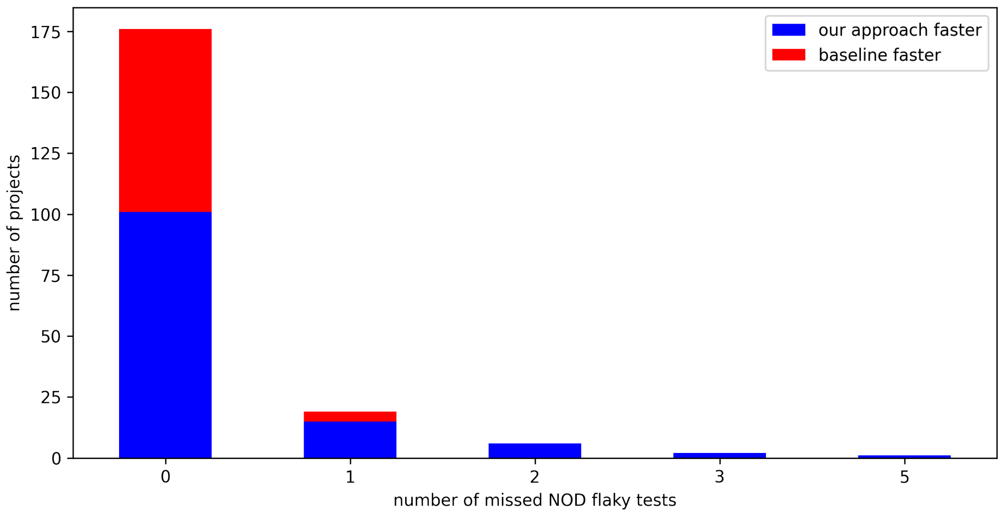

Figure 5.9.: Evaluation of flakiness detection on individual projects with error margin and runtime comparison

As shown in Figure 5.9, in 176 of the 204 projects our characteristic functions do not miss any NOD flaky tests. Since the average number of NOD flaky tests per project is only 2.66, we consider our approach as disadvantageous if any NOD flaky test is missed. The division in *blue* and *red* (see Figure 5.9) represents whether our approach or the baseline was faster. While our approach is on average 665 seconds faster (among *blue* projects), the baseline is on average only 62 seconds faster (among *red* projects). For the 101 *blue* projects of the 176 projects without missed NOD flaky tests, our approach is on average 379 seconds faster.

Answer to RQ2.3: For different projects, the performance of our approach to detect flaky tests varies greatly. While the characteristic functions are accurate for 86% of the projects, our detection approach is also faster for only 49.5% of the projects. Although our approach saves more time on average, it has a significantly worse accuracy in the worst case. For most projects, one of the approaches is clearly better, but the better choice is not known beforehand.

Although our approach might be inferior among all tests combined, it is the better choice for 49.5% of our projects where it saves an average of 379 seconds. We were not able to identify project properties that are interrelated to the approach that should be chosen for flakiness detection. However, when analyzing the results, we came across a way to beat the baseline. If we split all tests into large and small tests we can use our approach for large tests only, while still rerunning all small tests. The fact that this approach is faster than rerunning all tests shows that the approach can be useful, although not as generally as expected.

Answer to RQ2:
Although our approach requires much fewer reruns, it is permanently slower than the baseline when detecting flaky tests among *all* tests. But for about half of our individual projects, flakiness detection using characteristic functions is the more sensible approach. However, we are not able to estimate beforehand for which projects this will be the case.

On the one hand, we cannot say that our approach generally saves effort for flaky test detection. On the other hand, our approach performs better in some cases (e.g. for large tests). Since test suites with flakiness exist that contain larger tests (for example for integration testing [6]), our approach should not be rejected on principle.

5.5. Discussion

5.5.1. Obtaining Characteristic Functions

Obtaining characteristic functions works in different ways, but ultimately requires knowledge about flakiness and the used programming language. While automatic selections can be helpful, the obtained characteristic functions still need to be manually checked and corrected. Analyzing the results of these processes, we expected to find patterns that could improve selecting characteristic functions fully automatically. For example, functions selected earlier have higher accuracy, so we considered terminating the optimization after a specified threshold. Unfortunately, none of the considered theories also improved the sensibleness of the automatically selected characteristic functions.

We derive that obtaining characteristic functions always implies manual work. Since flakiness occurs differently for different programming languages [8], we assume that the types of characteristic functions also differ. While datasets with pre-labeled root causes can save a lot of work, they are not available for many programming languages (see Section 4.1.1).

A resulting problem is the assessment of characteristic functions. To determine the quality of a set of characteristic functions, comparative sets are necessary. Due to the difference of flakiness between programming languages [8], comparing characteristic functions that belong to different programming languages may be misleading; while the choice of characteristic functions may be good, flakiness in the examined programming language may be generally harder to characterize. Thus, one requires several comparative sets in the same programming language. We have solved this by applying vastly different approaches, some of which we executed several times. This multiplies the effort of determining characteristic functions and therefore increases the likelihood that it will not be worthwhile.

To clarify whether the time-consuming but doable selection of characteristic functions is worthwhile, we will discuss their usability for flakiness detection next.

5.5.2. Flaky Test Detection

Instead of rerunning all tests to detect flakiness, we only choose tests that execute at least one characteristic function. While the major reduction of reruns shows, that the characteristic functions have been well chosen and correctly applied, we have to examine why this improvement does not affect the runtime required for flakiness detection. The poor performance can be explained by fixed and variable costs (with regards to execution time) of our approach. We will discuss fixed and variable costs individually and conclude the usefulness of characteristic functions for flakiness detection.

While for the baseline approach, all tests are rerun without prior computation, our approach induces fix costs; the overhead generated by tracing the function calls. The executions of our approach in Figure 5.8 contain an overhead of approximately 1.77 hours. Thus, without considering the function tracer, the efficiencies of our approach and the baseline approach are almost identical for quick flakiness checks (if the maximum number of reruns is smaller than 6). For shorter flakiness detection, the overhead is proportionally larger, and determines that our approach is slower. For larger executions, the overhead does not change much, but our approach is already slower due to the rerunning alone.

We refer to the execution time of the reruns as variable costs since they are adjustable by varying the maximum number of reruns. They depend on the number of reruns and the runtime which each rerun requires. Since we already know that our approach decreases the number of required reruns, the individual runtimes of tests that execute characteristic functions have to be decisive for the poor result.

We confirm this conjecture by re-examining the runtimes of the tests, this time separated into different categories. Again, we choose the runtime of a test as a metric for its size. First, we find that tests that execute characteristic functions are on average more than twice as big as the average of all tests. Since some characteristic functions indicate constructs like networking or concurrency, we expected an increased runtime, but not to this extent. Also, the NOD flaky tests with characteristic functions are larger than all NOD flaky tests on average. Considering only tests with characteristic functions, we try to filter out non-flaky tests beforehand. But the tests that we can exclude are on average seven times smaller than the average test. Furthermore, the excluded tests contain some missed NOD flaky tests, that are much smaller than the average NOD flaky test and can therefore be cost-efficiently detected by the baseline approach.

Similar to running large tests first (see Section 5.2.2), we also examine tests that are more likely to be flaky. But due to the drastically higher runtime, this effect is nullified by the higher costs of rerunning. The baseline in comparison wastes more executions on non-flaky tests, but these are generally faster. Additionally, the few NOD flaky tests among the smaller tests are quickly detected, while our approach mainly detects large (and therefore costly) NOD flaky tests.

This begs the question of whether our approach can be helpful for large tests. Therefore, we slightly adapt our procedure. We use the baseline approach for small tests since they are quick to check and we do not want to miss any cost-efficiently detectable NOD flaky tests. For large tests, we use our approach and only rerun tests that execute characteristic functions, to exclude some costly tests that are most probably non-flaky. To distinguish between small and large tests, we choose 4 milliseconds as a threshold, which is the median runtime of the NOD flaky tests that we miss. This means that our new approach will only miss half as many NOD flaky tests and still exclude as many larger tests as possible. Evaluated as usual, with 170 as the maximum number of reruns, our new approach detects 8.7% more NOD flaky tests per second than the baseline.
So possibly our approach is well suited for test suites whose tests are generally larger since less small NOD flaky tests would be missed and more significantly large tests could be excluded. However, our dataset is inadequate to confirm this assumption.

5.5.3. Future Work

For the sake of feasibility, we had to limit our research to flakiness in Python. Since flakiness is reported and researched in various programming languages [9, 2, 13, 5], the methods we studied to identify characteristic functions could be applied to them as well.

When analyzing test suites that use manual annotations to select tests for rerunning [10], one could evaluate approaches that extend or reduce the set of annotated tests based on characteristic functions.

While we executed early tests that showed no correlation between the frequency of flakiness and the number of occurring characteristic functions, one could also analyze the position, common occurrences, and prevalence of characteristic functions besides their pure existence.

With access to more data, one could not only improve the identification of characteristic functions but also evaluate our approach for other kinds of projects. Although we have shown that our approach is not recommendable for most PyPI projects whose tests are generally small, we suspect that it could be useful for other types of test suites. Finding features of test suites that recommend the use of our approach would result in significant time savings. But still, we would not have fulfilled the original goal of *generally* improving the detection of NOD flakiness.

6. Threats to Validity

Lastly, we want to highlight problems and our corresponding solutions to be confident about the correctness of our results.

6.1. Threats to External Validity

We only analyzed Python projects. Thus, all results and findings are not necessarily valid for other programming languages. Due to our high requirements for the dataset and the insufficient availability of data for different programming languages, we have to accept this limitation. We hope that our research will be considered when examining flakiness in other programming languages, to consequently expand our findings.

Moreover, we did not use multiple systems for execution but only one computer running Linux. On the one hand, this increases the comparability of our execution time measurements, which is good. On the other hand, we may miss tests that can only be executed on certain systems. Fortunately, these tests were explicitly marked as *Infrastructure Flaky* by Gruber et al. [8] and are not included in the NOD flaky tests. As long as this classification is reliable, the underlying system should not influence the execution. This implies repeatability on other machines and operating systems.

Using experiments that include non-deterministic elements like flaky tests is bound to introduce variation. Thus, we carefully examined the possible variations, for example by checking that both passing and failing runs will execute possibly characteristic functions. Individual fluctuations in runtimes should be compensated by the large number of values. Additionally, we only conducted experiments when necessary and therefore decided to use statistical calculations in the evaluation. Hence, our evaluation is not subject to chance but representative for the studied approaches.

Potentially there exist other methods to obtain characteristic functions that may yield much better results. This would have a large impact on the detection of NOD flakiness and could lead to a different evaluation. To counteract this possibility, we analyzed vastly different methods of selecting characteristic functions. We observed similarly accurate results despite differences in the level of automation and the underlying data (see Table 5.1). The coherence of the results leads us to believe that no significantly better procedures are conceivable.

6.2. Threats to Internal Validity

We countered possible programming errors by testing. We checked the function call tracer using an extensive test suite and manually verified the results of the runtime parser.

The function calls of parallel executions can only be traced when using the standard threading module. Since we were aware of this problem, we analyzed the traces (see Figure 5.2) and found that it only occurs for a few outliers. The other way our tracer fails is if the tracer is overwritten in the test itself. While this is technically possible, it is very unlikely since it would also impede the use of debuggers to resolve faults.

The manually obtained root causes in our dataset could be faulty. Since both manual and part of the automatic approaches to obtain characteristic functions rely on this data, this would be problematic. Fortunately, they were double-checked [15] with an error rate of only 2% in the first pass. Similarly, the manual selection of characteristic functions may contain human errors. Its coherence with the automatic selection and separate analysis (see Figure 5.6) however both validate the manual selection.

Finally, our evaluation is only based on the expected values for the number of detected NOD flaky tests and the necessary execution time. Besides the high practical relevance, this choice is particularly suitable for comprehensible comparisons. Since this comparison consists of very similar approaches (rerunning a specific set of tests), the result should be stable with respect to other confidence levels. Although the absolute values of the evaluation would change for different confidence values, they would change consistently for the different approaches and would therefore not affect the final result.

7. Conclusion

Flaky tests, that may pass and fail without changing the underlying code, hinder regression testing. Since they affect the perceived reliability of a test suite [3], it is essential to identify them. Flakiness detection does not only increase the perceived reliability of the other tests but also allows to apply approaches that locate and fix flakiness [12, 13, 5, 11, 15]. Unfortunately, detecting flaky tests is expensive; Google, for example, spends 2% to 16% of test execution costs on rerunning tests to detect flakiness [6].

Some types of flakiness can be detected efficiently [9], but detecting NOD flaky tests is generally time-consuming: The prevalent method of detecting NOD flakiness is to rerun a set of tests multiple times and check for varying outcomes, that would imply flakiness. While rerunning all tests is costly [8], rerunning only tests that are manually annotated as potentially flaky misses many NOD flaky tests [10]. Hence, we studied selecting tests based on dynamic program analysis, to rerun as few tests as possible without substantially missing NOD flaky tests.

Since function calls are used to fix flakiness [7] and connections between system calls and root causes of NOD flakiness were discovered [15], we traced the function calls of tests in a flakiness dataset. We studied multiple approaches to identify functions that characterize flakiness. In doing so we discovered that fully-automated selections are not reliably characteristic, while manual and semi-automatic approaches lead to promising results. The identified characteristic functions appear in more than 90% of the NOD flaky tests but in fewer than 40% of all tests.

Compared to rerunning all tests, we required 47% fewer reruns when only executing tests that contain characteristic functions to detect the same number of NOD flaky tests. But upon further analysis, this effect is offset by the increased runtime of the selected tests. In our dataset, our approach performed similarly to rerunning all tests, but not better. Looking at the analyzed projects separately, our approach is preferable in nearly 50% of them.

We confirmed that there are indeed functions that are particularly common with NOD flaky tests and whose selection does not pose a problem. It turned out that the number of reruns, which are a commonly used metric for flakiness detection [8], can be very misleading in terms of efficiency. Due to the increased runtime, our approach is not generally recommendable and shows, that additional research on NOD flakiness detection is needed. Examining other projects and programming languages may expose features of test suites that recommend the use of our approach to reduce effort for flakiness detection.

A. Appendix

A.1. Sets of Characteristic Functions

A.1.1. Manually Selected Characteristic Functions

Table A.1.: Manually selected characteristic functions

Class	Function	Deriving Root Cause
Condition	wait	Async Wait
Random	[all]*	Randomness
RandomState	[all]*	Randomness
socket	__init__	Network
type	now	Time
	create_connection	Network
	open	IO
	start_new_thread	Concurrency

*All functions of this class were selected as characteristic

A.1.2. Automatically Selected Characteristic Functions Based on Root Causes

Table A.2.: Automatically selected characteristic functions based on root causes for $\beta = 24$

Class	Function	Deriving Root Cause
AveragedFunction	diffse	Randomness
ForkContext	SimpleQueue	Concurrency
MediaWiki	_load_page	Async Wait
PosixPath	_init	IO
Random	random	Randomness
RandomParser	get	Time
SelectSubjectRawDatasetReader	_assert_input_dict	Floating Point
str	replace	Network
	_diag_dispatcher	Randomness
	_report_hook	Time
	generate_alias	Network

Class	Function	Deriving Root Cause
	get_titles	Randomness
	int	Randomness
	search	IO
	sleep	Async Wait
	test_forward	Randomness

A.1.3. Automatically Selected Characteristic Functions From C Functions

Table A.3.: Automatically selected characteristic functions from C functions for $\beta = 5$

Run	Class	Function
1	DirEntry	is_file
1	FontMathWarning	with_traceback
1	frozenset	__contains__
1	Match	end
1	Random	getrandbits
1	RandomState	binomial
1	RandomState	chisquare
1	RandomState	rand
1	str	encode
1	String	__reduce_ex__
1	StringIO	getvalue
1	UTC	fromutc
1		collect
1		dot
1		getdefaulttimeout
1		perf_counter
2	bytes	count
2	CaptureIO	truncate
2	complex128	reshape
2	FASTA_error	with_traceback
2	FontMathWarning	with_traceback
2	frozenset	__contains__
2	ndarray	__array_prepare__
2	NoneType	take_2d_axis0_float64_float64
2	Random	getrandbits
2	RandomState	chisquare
2	RandomState	multinomial
2	RandomState	randn
2	str	encode

Run	Class	Function
2	String	__reduce_ex__
2	StringIO	getvalue
2	UTC	fromutc
2		ascii
2		collect
2		locals
2		mkdir
2		sleep
3	BlockManager	_rebuild_blknos_and_blklocs
3	CaptureIO	seek
3	complex128	reshape
3	coroutine	send
3	FASTA_error	with_traceback
3	FontMathWarning	with_traceback
3	Random	getrandbits
3	RandomState	chisquare
3	RandomState	rand
3	RandomState	randn
3	str	encode
3	str	rstrip
3	String	__reduce_ex__
3	StringIO	flush
3		collect
3		dot
3		isenabled
3		locals
3		start_new_thread
4	BlockManager	_rebuild_blknos_and_blklocs
4	CaptureIO	truncate
4	complex128	reshape
4	FASTA_error	with_traceback
4	Random	getrandbits
4	RandomState	chisquare
4	RandomState	multivariate_normal
4	RandomState	rand
4	RandomState	randn
4	str	encode
4	str	rstrip
4	String	__reduce_ex__
4	StringIO	getvalue

Run	Class	Function
4	UTC	fromutc
4		collect
4		exec
4		pow
4		sleep
4		start_new_thread
5	BlockManager	_rebuild_blknos_and_blklocs
5	bytes	rstrip
5	CaptureIO	truncate
5	complex128	reshape
5	datetime	utcoffset
5	FASTA_error	with_traceback
5	int64	any
5	Random	random
5	RandomState	chisquare
5	RandomState	multivariate_normal
5	RandomState	rand
5	RandomState	randn
5	str	encode
5	str	rstrip
5	StringIO	flush
5	TextIOWrapper	readlines
5	UTC	fromutc
5		_excepthook
5		collect
5		isenabled
5		sleep

A.2. Assignment of Root Causes to Characteristic Functions

The root causes originate from the description of Gruber et al. [8].

Table A.4.: Assignment of root causes to characteristic functions

Class	Function	Assigned Root Cause
AveragedFunction	diffse	-
BlockManager	_rebuild_blknos_and_blklocs	-
bytes	count	IO
bytes	rstrip	-
CaptureIO	seek	IO
CaptureIO	truncate	IO
complex128	reshape	Floating Point*
coroutine	send	Concurrency
datetime	utcoffset	Time
DirEntry	is_file	IO
FASTA_error	with_traceback	-
FontMathWarning	with_traceback	-
ForkContext	SimpleQueue	Concurrency
frozenset	__contains__	-
int64	any	-
Match	end	-
MediaWiki	_load_page	Network
ndarray	__array_prepare__	Floating Point*
NoneType	take_2d_axis0_float64_float64	-
PosixPath	_init	IO
Random	getrandbits	Randomness
Random	random	Randomness
RandomParser	get	Randomness
RandomState	binomial	Randomness
RandomState	chisquare	Randomness
RandomState	multinomial	Randomness
RandomState	multivariate_normal	Randomness
RandomState	rand	Randomness
RandomState	randn	Randomness
SelectSubjectRawDatasetReader	_assert_input_dict	-
str	encode	Network
str	replace	-
str	rstrip	-
String	__reduce_ex__	-

*Kilitcioglu et al. verified that NumPy may produce non-deterministic results due to numeric instability [31] which may lead to *Floating Point* flakiness [2].

Class	Function	Assigned Root Cause
StringIO	flush	IO
StringIO	getvalue	IO
TextIOWrapper	readlines	IO
UTC	fromutc	Time
	_diag_dispatcher	Floating Point*
	_excepthook	Concurrency
	_report_hook	Network
	ascii	Network
	collect	-
	dot	Floating Point*
	exec	Concurrency
	generate_alias	IO
	get_titles	-
	getdefaulttimeout	Network
	int	-
	isenabled	Network
	locals	-
	mkdir	IO
	perf_counter	Async Wait
	pow	-
	search	-
	sleep	Async Wait
	start_new_thread	Concurrency
	test_forward	-

*Kilitcioglu et al. verified that NumPy may produce non-deterministic results due to numeric instability [31] which may lead to *Floating Point* flakiness [2].

List of Figures

List of Tables

Acronyms

Bibliography

[1] S. Yoo and M. Harman. “Regression testing minimization, selection and prioritization: a survey”. In: *Software Testing, Verification and Reliability* 22.2 (Feb. 2012), pp. 67–120.

[2] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. “An empirical analysis of flaky tests”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Nov. 2014.

[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli. “Understanding Flaky Tests: The Developer’s Perspective”. In: (July 2019).

[4] S. Dutta, A. Shi, R. Choudhary, Z. Zhang, A. Jain, and S. Misailovic. “Detecting flaky tests in probabilistic and machine learning applications”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, July 2020.

[5] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta. “Root causing flaky tests in a large-scale industrial setting”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, July 2019.

[6] J. Micco. *The State of Continuous Integration Testing @Google*. 2017.

[7] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta. “A study on the lifecycle of flaky tests”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, June 2020.

[8] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser. “An Empirical Study of Flaky Tests in Python”. In: (Jan. 2021).

[9] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. “iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests”. In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, pp. 312–322.

[10] D. Silva, L. Teixeira, and M. d’Amorim. “Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2020.

[11] V. Terragni, P. Salza, and F. Ferrucci. “A container-based infrastructure for fuzzy-driven root causing of flaky tests”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, June 2020.

[12] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov. “iFixFlakies: a framework for automatically fixing order-dependent flaky tests”. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Aug. 2019.

[13] C. Ziftci and D. Cavalcanti. “De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google”. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sept. 2020.

[14] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino. “Know You Neighbor: Fast Static Prediction of Test Flakiness”. In: *IEEE Access* 9 (2021), pp. 76119–76134.

[15] R. Würsching. “Determining Root Causes of Flaky Tests Using System Call Analysis”. MA thesis. Technische Universität München, May 2022.

[16] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov. “Understanding Reproducibility and Characteristics of Flaky Tests Through Test Reruns in Java Projects”. In: *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, Oct. 2020.

[17] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky. “Selecting a Cost-Effective Test Case Prioritization Technique”. In: *Software Quality Journal* 12.3 (Sept. 2004), pp. 185–210.

[18] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. “Empirically revisiting the test independence assumption”. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, 2014.

[19] H. Krekel. *pytest Documentation*. Release 7.1. merlinux. Apr. 2022.

[20] A. Zeller. *The Debugging Book*. CISPA Helmholtz Center for Information Security, 2021.

[21] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009.

[22] A. Kervinen, J. Jaakkola, A. Nieminen, and T. Mikkonen. “Towards Eased Debugging of Python Applications on Maemo Platform”. In: *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*. Mobility ’09. New York, NY, USA: Association for Computing Machinery, 2009.

[23] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362.

[24] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. “Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications”. In: *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Apr. 2016.

[25] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. “DeFlaker: automatically detecting flaky tests”. In: *Proceedings of the 40th International Conference on Software Engineering*. ACM, May 2018.

[26] D. Anguita, A. Ghio, S. Ridella, and D. Sterpi. “K-Fold Cross Validation for Error Rate Estimate in Support Vector Machines.” In: Jan. 2009, pp. 291–297.

[27] S. Arlot and M. Lerasle. "Choice of V for V-Fold Cross-Validation in Least-Squares Density Estimation". In: *Journal of Machine Learning Research* 17.208 (2016), pp. 1–50.

[28] J. Ruohonen, K. Hjerppe, and K. Rindell. "A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI". In: (July 2021).

[29] J. Listfield. *Where do our flaky tests come from?* Google Testing Blog. Apr. 2017.

[30] W. Spotz. "Cool Stuff You Can Do with PyTrilinos DistArray and Jupiter." In: (Oct. 2015).

[31] D. Kilitcioglu and S. Kadioglu. "Non-Deterministic Behavior of Thompson Sampling with Linear Payoffs and How to Avoid It". In: *Transactions on Machine Learning Research* (2022).