TUHH
Hamburg
University of
Technology

Bachelor Thesis

Fin Bießler

Randomized Generation of Flaky Test Suites

March 8, 2024

supervised by:
Prof. Dr. Sibylle Schupp
Daniel Rashedi

Hamburg University of Technology (TUHH)
Technische Universität Hamburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Declaration of Originality[1]

Hereby I confirm, Fin Bießler, Matr. Nr. 52906, that this assignment is my own work and that I have only sought and used mentioned tools.

I have clearly referenced in the text and the bibliography all sources used in the work (printed sources, internet or any other source), including verbatim citations or paraphrases.

I am aware of the fact that plagiarism is an attempt to deceit which, in case of recurrence, can result in a loss of test authorization.

Furthermore, I confirm that neither this work nor parts of it have been previously, or concurrently, used as an exam work – neither for other courses nor within other exam processes.

Hamburg, March 8, 2024

Fin Bießler

[1]University of Tübingen, Department of Sociology

Abstract

Automatic program repair and fault localization tools have been developed to aid developers in resolving software faults. State-of-the-art tools extensively use test suites to find causes of errors, fix them and lastly validate their solution, but tests can be flaky in the sense that they do not behave deterministically, tests might fail and succeed in independent test suite runs without altering source code. This is why flaky tests are often removed from repair benchmarks testing such tools which limits the scope of such approaches. This is why this thesis aims to provide an approach that can randomly generate flaky test suites to aid further development within the automatic program repair and fault localization research community. For this, the program focuses on generating flaky test cases that are flaky due to randomness or test order-dependency. As well we enabled the user of the program to configure the test suite generation such that parameters like test suite size, test suite composition or the failure chance of a flaky test could be determined. When evaluating we found that the program developed in that matter can generate flaky test suites with a majority of test cases that can be detected as flaky or faulty by the state-of-the-art fault localization tool `CharmFL`. In further work, the approach could be extended to cover more flakiness root causes.

Contents

List of Figures

Listings

List of Tables

Acronyms

APR	Automatic Program Repair. 1, 3, 5, 6, 18
AST	Abstract Syntax Tree. ix, 10, 23, 25
ESHS	Executable Statement Hit Spectrum. 4
FL	Fault Localization. 1, 3, 4, 6, 18, 27, 29, 31
JSON	JavaScript Object Notation. 21
SBFL	Spectrum-Based Fault Localization. 29, 30

1. Introduction

Software bugs play a huge role in the development and maintenance of software projects, they can be very expensive to fix and if not fixed they can have catastrophic impacts on the software system. This emphasizes the fact that software failures have to be dealt with and need to be fixed as soon as possible upon discovering them.

When a software malfunction is discovered the cause of it is often not obvious initially, developers have to investigate the cause and among others, dig through huge code bases, analyze log files and debug the code. All of this is very tedious and poses additional resource demand upon the company effectively resulting in increased costs. This is why Automatic Program Repair and Fault Localization tools have been developed, to aid developers in the process of troubleshooting. On the one hand APR tools such as `SemFix` [1], `Nopol` [2] or `Autofix` [3] aim at automatically locating software bugs and fixing them with minimal human intervention, reducing developer resources that have to be invested into software maintenance. On the other hand FL tools such as `CharmFL` [4] go a step back and just aid the developer in finding the faulty location in the first place. For these statements, methods/functions and classes are ranked by their suspiciousness, indicating the likelihood for them to be faulty.

For both approaches, the state-of-the-art tools utilize test suites to achieve this. Test suite outcomes are used as oracles to identify faulty locations within the code base, the problem with this is that test cases might not be deterministic. Non-deterministic test cases are called flaky, their verdict changes between runs even though neither the tested code nor the test code itself is altered. Problematically, flaky tests are often eradicated from benchmarks that test the previously mentioned tools, this limits the scope of such tools. It also reduces the confidence in these tools that they can be used to fix or identify faults in real-world software project.

We contribute an approach for generating randomized flaky test suites to test the previously mentioned APR and FL tools on flaky tests and evaluate their performance among these. The contribution includes the following key features:

- Randomly generating flaky test suites stemming from test order-dependency, randomness and async-wait.
- Providing different configurability options for the user of the tool, such as determining the total test number to be generated, the distribution of the number of test cases among the different flakiness categories and the probability with which a test case from the randomness category fails.
- Sharing experimental evaluation results regarding the following two research questions:
  - Are the generated test cases flaky?
  - Does `CharmFL` detect the test cases to be flaky or faulty?

The evaluation has shown that the test cases generated are flaky with some minor exceptions for test cases that were intended to be non-flaky. Additionally, `CharmFL` can detect the flakiness and faultiness of test cases stemming from the randomness root cause whereas the tool had a hard time detecting the flakiness caused by test order-dependency and async-wait.

The work is structured as follows. First, in Chapter, 2 related work regarding the presented approach is briefly presented. Second, in Chapter 3 theoretical concepts and definitions needed for the following discussions are introduced. Following that, in Chapter 4 the main contribution of how to generate randomized flaky test suites is discussed. Furthermore, in Chapter 5 some important aspects regarding the implementation of the tool are shared with the reader. Moreover, in Chapter 6 results for the evaluation of the previously discussed approach are presented. The thesis closes with Chapter 7, where conclusions regarding the discussions are drawn as well as further work that is worthwhile doing is mentioned.

2. Related Work

In the following chapter insights on related work will be given. First, Automatic Program Repair and Fault Localization techniques will be discussed. Second, given the knowledge of prior approaches to generate flaky test suites key differences between these and the presented one will be highlighted. The relation of these techniques regarding the approach presented in this work is that the presented approach aims at improving existing APR and FL tools. This is done by providing flaky test suites to APR and FL tool developers such that they can extend their work to also repair faults in software projects that contain flaky tests in their test suite.

2.1. Fault Localization Techniques

According to Wong et al., [5] FL techniques can be categorized into *traditional FL techniques* and *advanced FL techniques*. As mentioned previously FL techniques are relevant for the presented work since the novel approach developed in this work aims at providing support to improve existing FL techniques. In the subsequent section, a brief overview of these FL techniques will be given.

Traditional Fault Localization Techniques

These kinds of techniques are traditional, basic and intuitive ways to locate faults in software. Even though they are not related to any recently developed FL techniques we will nevertheless briefly state them for completeness purposes. They also represent the tedious work developers would have to perform when no FL tools would be available to them.

- **Program Logging**: By adding amongst others `print` statements to the code base, program logs can be constructed that are used to identify faulty sections of the program
- **Assertions**: Constraints are added to the program's source code that makes the execution terminate if evaluated to false. Like this erroneous behaviour can be identified during runtime.
- **Breakpoints**: The execution of a program halts at breakpoints where developers can inspect the state of the program at that point of the execution and alter it. After the breakpoint, execution can be resumed to examine the progression of the respective bug.
- **Profiling**: Program metrics such as execution speed and memory usage are observed to optimize a program at most times. Nevertheless, it can be used to identify suspicious behaviour by for example detecting unexpected execution frequencies of different functions.

Advanced Fault Localization Techniques

Due to the large size and scale of today's software projects the previously described traditional FL techniques are not the most effective ways to identify the location of bugs within source code, as they include a lot of manual work to be conducted. Thus different advanced approaches have been developed, that address this problem in different manners. Some of the most relevant approaches for this work will be presented briefly in the following.

- **Slice-Based Techniques**: The idea is to reduce the search space in which developers look for the cause of a bug within the source code by deleting unconcerned parts of the program according to a certain specification. *Static slicing* was first proposed by Weiser in 1979 [6], it aims at reducing the search space in which faults are trying to be located. Suppose we have a failed test due to an incorrect variable value at a statement, then *static slicing* says the fault should be found in the *static slice* of the considered variable statement pair. This allows for reducing the search space to the respective *static slice* from the entire program.

- **Program Spectrum-Based Techniques**: They use probabilistic- and statistical-based causality models in which information about execution behaviour is retrieved to identify buggy locations. For example, code parts are ranked by suspiciousness according to how many failing or passing test cases executed them.

  Program spectra detail the execution information of a program. These can be used to track a program's behaviour and thus localize faults. Information on failing and passing test cases, and emphasising the difference between them, is used to pinpoint code which produces the fault. For this code coverage or Executable Statement Hit Spectrum indicate which program parts were executed during the test run. This information helps in narrowing down the search for the faulty component.

  In Chapter 6 we will evaluate the newly developed approach in this work, by using the existing spectrum-based FL tool `CharmFL` to identify flaky or faulty locations in the generated test cases. Since `CharmFL` uses a spectrum-based approach it is relevant to discuss this method in more detail in the following.

  As proposed by Renieres and Reiss [7] nearest-neighbour techniques analyze the difference between a test case that failed and one that is similar concerning the distance between them but succeeded. A statement seems more *suspicious* when its execution pattern is closer to the failure pattern of all tests because it is more likely to be faulty in this case. Likewise, the farther an execution pattern is to the failing pattern the less likely it is to be faulty and thus it seems less *suspicious*. Similarity coefficients are used to measure and quantify the distance between statements.

  Wong et al. [5] proposed the following notations and metrics to measure the suspiciousness of a statement.

Notation

- p: a program
- Considering failed test cases covering a statement of p, the count of these test cases is denoted by N_{CF}.
- Considering succeeded test cases covering a statement of p, the count of these test cases is denoted by N_{CS}.
- The count of all succeeded test cases is denoted by N_S
- The count of all failed test cases is denoted by N_F

Among these similarity-coefficient-based techniques is Tarantula [8] a standard similarity-coefficient-based metric that is used by `CharmFL`, for which the suspiciousness of a statement is calculated as

$$Suspiciousness(Tarantula) = \frac{\frac{N_{CF}}{N_F}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}}.$$

Another similarity coefficient that can be used when using `CharmFL` is Ochiai [9] which is calculated as follows

$$Suspiciousness(Ochiai) = \frac{N_{CF}}{\sqrt{N_F \times (N_{CF} + N_{CS})}}$$

- **Program State-Based Techniques**: The program state (program variables and their respective values) is compared to some reference state during execution, this is also called relative debugging. During runtime, the current program state is compared to a reference state where the bug was not present to conclude the location of the fault in the source code. Another approach would be to alter the values of certain variables to discover which variable caused the faulty execution.

  Regarding this Zeller and Hildebrandt proposed a technique called delta-debugging [10]. Delta debugging compares program states between executions of failed and successful test cases via their memory graph. Suspiciousness is determined by replacing variable values from successful tests with the ones from the same point of failed tests, afterwards, execution is continued. When the same error is observed the location is still considered suspicious, otherwise not.

2.2. Automatic Program Repair

Software maintenance is costly and drains many resources such as money and time, thus as the name suggests, APR aims at finding faults in software and providing fixes for these, so-called *patches*. Test suites play a significant role in such techniques as they function as executable software specifications. Test suite-based repair was initially introduced by `GenProg` [11], an APR system that is based on genetic programming. Research has developed and new APR approaches were discovered. These approaches can be categorized into generate-and-validate as well as synthesis-based techniques. In the following, a brief overview of both approaches will be given.

Generate-and-Validate Approaches

Generate-and-validate approaches like `Astor` [12], `CapGen` [13], or the previously mentioned `GenProg` method all fall into the APR category. The key characteristic of this approach is to first generate as many patches fixing the viewed bug. Afterwards, the test suite is used to validate whether there is a patch among the candidates that makes all test cases pass.

Synthesis-Based Approaches

Synthesis-based approaches such as `SemFix` [1], `Nopol` [2] or `Autofix` [3] address the problem in a reversed manner. They first execute the test suite and extract constraints from the execution results, based on which a patch is synthesized by using a constraint solver. [14]

A problem that arises for these techniques is that flaky tests are often removed from the benchmarks implementing these techniques [15]. Though flaky tests are prevalent in real-world software projects, thus the absence of testing these tools on flaky tests might pose limitations to their scope.

2.3. Generation of Test Suites

Now that a brief overview of APR and FL techniques has been given, the following is a brief introduction to previous approaches on how to generate flaky test suites.

Regarding Gruber et. al. [16], the two state-of-the-art techniques for generating flaky test suites were more of a passive fashion, in the sense that no new code was generated ab initio. Rather existing code was collected and shipped as a dataset for flaky test suites.

For this two major approaches are known to the community. First, look for commits in version control systems or bug trackers that fix issues related to flakiness. Second, collect tests from existing software projects rerun them a certain number of times and determine whether their results change throughout the execution when not changing the code. Tests that had passing runs as well as failing runs are then coined as flaky and are collected into the dataset of flaky test suites.

So to emphasize the difference between existing approaches to generating flaky test suites and the one presented in this thesis, recall that previously existing code was passively collected whereas in the presented novel approach new code is actively randomly generated ab initio.

3. Theoretical Background

This section covers necessary definitions and terms needed for the subsequent chapters of this thesis.

3.1. Test Cases and Test Suites

Because we present an approach to generate randomized test suites in this thesis, the following section first defines what test cases are, how they are combined into test suites and what purpose they serve in a real-life software project.

Test Cases

A test case is a piece of software that is written to test parts of the production code of their respective software project. According to the widely used *Arrange Act Assert* pattern a test case can be divided up into three sections [17]. In the following, this pattern is going to be formalized and further described.

- n: Number of statements in the test
- s_n: Sequence of statements in the test
- s_i: Statement i of the test
- $s_i \forall i \in [1, .., k]$: Arrange statements of the test
- $s_i \forall i \in [k+1, .., l]$: Act statements of the test
- $s_i \forall i \in [l+1, .., n]$: Assert statements of the test

Now a brief description for each statement section follows.

The k arrange statements: For testing the production code several prerequisites have to be prepared such as setting up the objects to be tested for instance different variables have to be assigned. Furthermore some dependencies might have to be mocked because they are not tested in the concerned test case but are needed for the piece of code that is being tested.

The $l - k$ act statements: In this section the actual functionality that is going to be tested is carried out and the respective actual result is saved in some manner to later check its validity. This can be just one simple functionality or a more complex one that is composed of many simple ones.

The $n - l - k$ assert statements: In this last section the result is verified by asserting that the obtained actual result from the second section is equal to some expected result that is defined beforehand.

An example of a simple function and its respective test case is given in Listing 3.1. In lines one and two the function to be tested is defined. Following that in lines four to eleven the corresponding test is defined. Furthermore, in lines five to seven the needed variables are *arranged*, additionally, in line nine the actual value is produced by calling the function to be tested, which corresponds to the *act* section described above. Lastly, in line eleven, the actual and expected values are *asserted* to be equal.

Test Suites

Test suites are collections of several test cases that depict the functionality of the tested software. They are used to ensure that production code is working as intended by the developer. When new production code is deployed test suites can be carried out to check whether the newly deployed code breaks existing functionality, this method is called regression testing. Sizes of test suites depend on the size of the software project that is tested by the test suite, so they can vary from just a few test cases to an accumulation of many test cases.

```
 1  def add(a, b):
 2      return a + b
 3
 4  def test_add():
 5      a = 2
 6      b = 2
 7      expected = 4
 8
 9      actual = add(a, b)
10
11      assert actual == expected
```

Listing 3.1: A simple function and its respective test case

3.2. Flaky Tests

The term "flaky" means "unreliable", when a test is depicted as flaky it means that the outcome of this test is not reliable and can pass and fail for the same software version when run multiple times. Following Luo et al. [18] the tests are non-deterministic concerning a given software version. This creates problems when trying to locate software faults and later on resolve those faults automatically.

Luo et al. [18] have stated that flakiness in test suites always stems from one of 10 root causes. In the following subsections, the flakiness categories relevant to the approach presented in Chapter 4 will be described. In addition to that theoretical terms will be defined.

3.3. Flaky Tests Due to Randomness

As described by Luo et al. [18] the use of random numbers can make tests flaky. When using random number generators flakiness manifests when not handling all possible numbers that can be generated including all edge cases. For example, a certain test fails when a one-bit random number is generated to zero because the tests only handle the cases where it is one and undefined behaviour occurs when the generated number is zero resulting in the respective test failing.

3.4. Flaky Tests Due to Test Order-Dependency

First of all test order-dependent tests are tests where the outcome depends on the order in which the tests are run. Generally, tests should be written in an isolated way resulting in independence between them and enabling any execution order to succeed. But in reality, this is not always the case. This behaviour occurs when tests work on a shared state that is not properly set up or cleaned. This state can either be in main memory (*global* variables in Python) or some external resources like *files* or *databases*. On the one hand, a "polluter" test was run before another test which fails the latter test because it expects to find the state to be as initialized, but the first test altered this state. On the other hand, the "state-setting" test was not run before another test that expects a preceding test to prepare the state in a certain way to pass.

Definitions proposed by Shi et. al. [19] concerning test order-dependent tests, which we use to describe the approach on how to randomly generate test cases from that category, will be given in the following.

- All tests of a test suite collected in a set can be denoted by T.
- The sequence of a subset of tests from T is called a *test order*.
- Given a test order O that contains a test $t \in T$, when running t in test order O $run_t(O)$ denotes the result of test t.
- Consider the last test run in order O, the result of that test is denoted by $run(O)$.
- A test order consisting of just one test t is denoted by $[t]$.
- Concatenating two test orders O and O' is denoted by $O + O'$.

Definition 1. *A test $t \in T$ has a **passing test order** or a **failing test order** O if $run_t(O) = PASS$ or $run_t(O) = FAIL$, respectively.*

Definition 2. *An **order-dependent test** $t \in T$ has a passing test order O and a failing test order $O^{'} \neq O$*

Order-dependent tests can be classified into two categories, *victim* and *polluter*. Furthermore, other tests that are related to test-order-dependent tests, but are not flaky themselves can be classified into the following three categories: *cleaner*, *brittle* and *state-setter*. In the following, these categories will be defined.

Definition 3 (Victim)**.** *An order-dependent test $v \in T$ is a **victim** if $run([v]) = PASS$*

Definition 4 (Polluter)**.** *A test order (with one or more tests) P is a **polluter** for a victim v if $run(P + [v]) = FAIL$*

Definition 5 (Cleaner)**.** *A test order (with one or more tests) C is a **cleaner** for a polluter P and its victim v if* $run(P + C + [v]) = PASS$

Definition 6 (Brittle)**.** *An order-dependent test* $b \in T$ *is a **brittle** if* $run([b]) = FAIL$

Definition 7 (State-Setter)**.** *A test order S is a **state-setter** for a brittle if* $run(S + [b]) = PASS$

3.5. Abstract Syntax Tree

There are two reasons for generating the randomized test suites on the Abstract Syntax Tree level. First, to avoid directly working with source code strings. Second, to conveniently transform them to source code and write them to their respective files in the file system. Thus a brief introduction to the AST is given in the following.

An AST is a tree representation of source code that contains the program's constructs such as if-statements, loops, or variable assignments as its nodes the edges between these nodes indicate the relation between the program's constructs. For example, all statements that are carried out within a loop are children of the loop node in the tree. Depending on the implementation of the represented programming language, the tree can carry additional information such as the source code lines in which the respective nodes are written. Conveniently each node and its children form a new subtree that is also an AST, which makes traversing the tree recursively and interchanging parts of the structure very natural.

In Figure 3.1 the AST for the expression `a=a+1` is shown. The *root node* is a variable assignment where the right child is the source, in this case, a binary addition operation with the operands variable *a* and constant 1. The left child of the variable assignment node is the target to which the result of the operation should be assigned, in this case, the variable *a*.

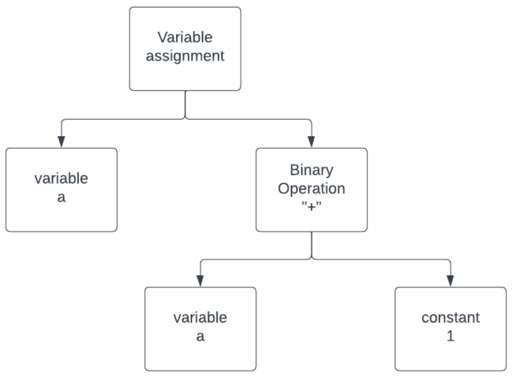

Figure 3.1.: AST of the expression `a=a+1`

4. Generating Randomized Test Suites

In the following chapter, the novel approach to generating randomized test suites is going to be presented. In contrast to the approach presented in Section 2.3, the novel approach actively generates new test cases and respective functions ab initio, rather than passively collecting them from existing software repositories or bug-tracking systems.

4.1. Generators

Regarding Luo et. al. [18] flakiness stems from one of ten following flakiness categories:

- Async-wait
- Concurrency
- Test order-dependency
- Resource leak
- Network
- Time
- I/O
- Randomness
- Floating point operations
- Unordered collections

One can develop approaches for each category and then combine them and generate test suites containing tests that are flaky concerning the implemented categories. Depending on the flakiness category different approaches have to be followed. For this generators for all considered categories in this thesis are implemented. The following subsections present all regarded flakiness root causes and their respective generators.

4.1.1. Randomness

The randomness root cause is the first one to be considered. We decided to cover this flakiness category to provide a simple start to initialize the work on this approach. In the following subsection, the approach on how to implement generators for this root cause is going to be presented. The key idea is to generate arithmetical expressions of random length composed of random operands and operators which are then randomly interfered by some noise value. To elaborate on this in more detail the following is defined:

- $(o_k)_{k\in[1,\ldots,n]}$: Sequence of n random operators

- $(v_k)_{k\in[1,\ldots,n+1]}$: Sequence of $n+1$ random numbers as operands
- X: Random variable that represents the noise interference subexpression, consisting of an operator o_X and a value v_X, that can be defined as
  - $P(X=+0)=0.7$, here $o_X=+$ and $v_X=0$
  - $P(X=+1)=0.3$, here $o_X=+$ and $v_X=1$

using the above symbols the previously discussed arithmetical expressions can be defined like the following:

$$v_1o_1v_2\cdots v_no_nv_{n+1}X = v_1o_1v_2\cdots v_no_nv_{n+1}o_Xv_X$$

Example

Let

$$(v_k)_{k\in[1,\ldots,n+1]} = (+,-,\cdot,\%)$$
$$(o_k)_{k\in[1,\ldots,n]} = (2,2,3,4,5)$$

Then with a probability of 0.7 the arithmetical expression equals

$$\frac{(2+2-3)\cdot 4}{5}+0$$

and with a probability of 0.3 it equals

$$\frac{(2+2-3)\cdot 4}{5}+1.$$

For the randomization subexpression, one can differentiate two cases, either the arithmetical expression is interfered with the noise or it is left untouched. Considering the first case, five operator and operand combinations have to be excluded, namely division and zero as it would result in an undefined operation, subtraction or addition paired with the operand zero and multiplication or division with the operand one, as these are the neutral operations for the respective operands and would lead into not interfering the arithmetical expression at all.

Considering the second case, we also have to exclude the operator and operand combination division and zero as it would again result in an undefined operation. In contrast to the first case, the goal is to not interfere with the arithmetical expression with the interference noise subexpression, thus the previously excluded combinations are the only ones that are allowed. Namely subtraction or addition operators with zero as the operand and division or multiplication operators with one as the operand.

4.1.2. Test Order-Dependency

For the second root cause that is examined in this thesis, one can use the patterns discussed in Section 3.4, namely victim and polluter alongside brittle and state-setter. In the following section, we will formalize an approach to how to randomly generate flaky tests based on these patterns.

First one can define the concatenation of two sequences as the following.

Definition 8 (Sequence Concatenation)**.** *Let* $(a_n)_{n \in N}$ *and* $(b_k)_{k \in N}$*, then*

$$(a_n) + (b_n) \coloneqq (a_1, a_2, \cdots, a_n, b_1, b_2, \cdots b_k)$$

Victim- and Polluter-Pattern

For this pattern, a global state has to be defined, which is then used by the polluter and victim test cases. Therefore we have to look at the generated test cases in an accumulated fashion, in the following, we call these concatenated test cases a test case module.

This test case module T consists of multiple parts defined in the following sequences:

- $(g_n)_{n \in N}$: Sequence containing all statements related to the initialization of the global state used by the victim and polluter test cases
- $(v_k)_{k \in N}$: Sequence containing multiple test case definitions that function as victims regarding the global state
- $(p_l)_{l \in N}$: Sequence containing the polluter test case definition regarding the global state

Using the previously described sequences we can define the test case module as the concatenation of them

$$T = g_n + v_k + p_l$$

The state initialization statements contained in the sequence g_n must be carried out before the victim and polluter test cases contained in sets v_k and p_l, since the global state variables initialized in g_n are used in the test cases contained in v_k and p_l.

The victim test cases assert that the global state contains a success state and thus only passes if this is the case, in contrast, the polluter test case firstly *pollutes* the global state by altering it to contain a failure state and then secondly asserts the global state to contain the failure state. For the victim- and polluter pattern the global state is initialized with a success state, this leads to the fact that the polluter test case always succeeds and the victim test cases only succeed when they are run before the polluter.

We consider the previously discussed test cases to be a sequence of the form

$$t_1, t_2, \cdots, t_{n-1}, t_n.$$

Where t_i is the i-th test case in the sequence. Now let t_p be the polluter test case within the sequence, then the following holds

$$\forall i, p \in [1, ..., n] : t_i = \begin{cases} PASS & if\ i \leq p \\ FAIL & if\ i > p. \end{cases}$$

Brittle- and State-Setter-Pattern

As well as for the victim- and polluter-pattern a global state has to be defined for the brittle- and state-setter-pattern. For the brittle- and state-setter-pattern the generated test cases concatenated together also form a test case module. The module can be defined as the concatenation of the following sequences:

- $(g_n)_{n \in N}$: Sequence containing all statements related to the initialization of the global state used by the brittle and state-setter test cases
- $(b_k)_{k \in N}$: Sequence containing multiple test case definitions that function as brittles regarding the global state
- $(s_l)_{l \in N}$: Sequence containing the state-setter test case definition regarding the global state

Using the previously described sequences the test case module yields

$$T = g_n + b_k + s_l$$

The +-operator has the same notion as in the previous section. The main difference between the victim- and polluter-pattern is that the state is initially set to a failure state rather than a success state. The brittle asserts that the global state contains a success state in the same fashion as the victim previously. The state-setter in contrast to the polluter sets the global state to a success state rather than to failure state. This leads to the fact that the state-setter test case always succeeds and the brittle test cases only succeed when the state-setter test case was run before them.

To formalize this a bit further again assume the test cases to be a sequence of the form

$$t_1, t_2, \cdots, t_{n-1}, t_n.$$

Where t_i is the i-th test case in the sequence. Now let t_s be the state-setter test case within the sequence, then the following holds

$$\forall i, s \in [1, ..., n] : t_i = \begin{cases} FAIL & if\ i \leq s \\ PASS & if\ i > s. \end{cases}$$

Thus when executing the test cases described in the previous approaches in random order the test will be flaky. Note that even though the tests are executed in random order this does not fall into the randomness flakiness category, since order dependencies within the tests are the actual root cause of the flakiness. One can emphasize this by considering random order execution of order-independent tests, which in turn would be non-flaky assuming that they are non-flaky internally.

4.1.3. Async-Wait

The third flakiness root cause to be considered in this thesis is *async-wait*. We decided to cover this flakiness root cause in the work since it is regarded as one of the three most prevalent root causes by Luo et. al. [18]. The approach presented is a proof of concept and is not as extensive as the other ones presented.

Considering the generation the general idea is to create race conditions that render the test cases flaky. For this first a global state and two random delays, a success delay and a failure delay, are initialized. Second two asynchronous functions are generated, where one sets the global state to the success state after suspending for the success delay time. The other sets the global state to the failure state after suspending the failure delay time. Finally, a test case is generated that creates two asynchronous tasks that perform the two state-setting operations described above. In the end, the test case asserts that the global state is in the success state.

This results in flaky behaviour. Depending on the delay amount, one of the state-setting operations overrides the result of the other. This leads to the global state not deterministically holding the success state at the end of the test case execution.

To formalize this idea further one can group the previously described generation in the following sequences of statements:

- $(g_n)_{n \in N}$: Sequence containing all statements related to the initialization of the global state used by asynchronous functions and the test case
- $(d_k)_{k \in N}$: Sequence containing the delay initialization statements
- $(a_m)_{m \in N}$: Sequence containing the asynchronous function definitions statements
- $(t_l)_{l \in N}$: Sequence containing the test case definition statements

Using the previously described sequences the test case module yields

$$T = g_n + d_k + a_m + t_l$$

One can observe that the delay initialization decides whether the test case succeeds or fails. There is one delay initialized for the function setting the global state to the success state and one for the function setting the global state to the failure state. Later both functions are called asynchronously, meaning that the function with the greater delay will be carried out last. Thus the respective value will be the one that is held by the global state in the end. Even though we use a random approach to initialize these delays the root cause of the flakiness is still the *async-wait* itself. This just mimics real-world behaviour where one asynchronous task is suspended longer than the other one due to uncontrollable reasons. These reasons could vary, for example, they could be network-related such as waiting for a server to respond.

Concluding this, the theoretical approach on how to generate the test cases was elaborated. The following section elaborates on how to randomize the generation further. For the two approaches covered in the previous sections, the one concerning randomness is already inherently randomized due to the random arithmetical expression. For the test order-dependent tests some steps have to be taken to randomize their generation, which will be elaborated in the subsequent sections. With this, one tries to ensure that the generated test cases within one flakiness category differ. After that implementation details will be given in Chapter 5, and problems that occurred and decisions that were made will be discussed.

4.2. Randomizing by Naming and Value Selection

Considering the variable that holds the state that results in tests to be order-dependent, it is straightforward to randomize this by randomly picking the variable name. Furthermore, the previously discussed success and failure states can be of any data type and value, since the actual value is irrelevant. The key relevance lies in a value being defined as a success state and all other values being a failure state. For example, the states data type could be `integer`, where the success state is the value `1` and all other integer values are failure states. Another example would be to define the states to be of `string` type, where the success state is the value `"success"` and the failure state is the value `"failure"`.

Thus the state might be of any data type and for each collection of order-dependent tests the success value can be defined independently leading to randomized tests. Moreover when the value of the failure state can be randomly picked from the data types range excluding the success value.

Example

In the following two examples class definitions taken from a randomly generated test suite are taken and showcased. These use the previously discussed randomization strategy. The state variable `member_state_m` is randomized by adding the random post-fix `m`, `member_state_n` and `n` and vice versa.

```python
class class_victim_polluter_0dac8877ef91402a9a80a3851b964588:
    member_state_m = 'success_state'

    def set_member_state_m(self, value):
        self.member_state_m = value

    def get_member_state_m(self):
        return self.member_state_m
```

Listing 4.1: Class definition used in a flaky test from the victim- and polluter-pattern

```python
class class_brittle_state_setter_61030948535442d0b9d02d470eacf05f:
    member_state_n = 'failure_state'

    def set_member_state_n(self, value):
        self.member_state_n = value

    def get_member_state_n(self):
        return self.member_state_n

    def dummy_function(self):
        return 48
```

Listing 4.2: Class definition used in a flaky test from the brittle- and state-setter-pattern

For Listing 4.1 the flakiness is obtained by performing multiple operations on the `member_state_m` variable. In the polluter test case, the setter method of the class is used to alter the value of the state to contain a failure state. Victim test cases use the getter method to assert that the `member_state_m` variable contains a success state. All victim test cases that are executed after the polluter test case fail. The test cases are made flaky by randomizing the order in which they are executed.

For Listing 4.2 the flakiness is obtained by performing similar operations on the `member_state_n` variable. In the state-setter test case, the setter method of the class is used to alter the value of the state to contain a successful state. In the brittle test cases, the getter method of the class is used to assert that the state contains a successful state. All brittle test cases that are executed before the state-setter test case fail. The test cases are made flaky by randomizing the order in which they are executed.

4.3. Randomizing by Combining Flakiness Categories

Within the randomness flakiness category different arithmetical expressions can be generated, for example, expressions using all arithmetical operators or expressions using just one operator. Following that, one can generate test cases that assert the outcome of multiple arithmetical expressions at once in contrast to only one at a time. This allows for further randomization by combining different expression types and varying the number of expression results to be asserted.

Note that asserting multiple expression results within a test case that fails with some probability results in the whole test case to be more flaky. To describe that in more detail assume there is a test case T with multiple assertions, then the following can be defined:

- n: Number of assertions carried out within test case T
- P_n: Set of n probabilities with which the respective assertions succeed

Then the overall flakiness probability of the whole test case yields

$$(1 - p_1) \cdot (1 - p_2) \cdot \dots \cdot (1 - p_{n-1}) \cdot (1 - p_n)$$

with $p_i \in P_n$.

Note that the success probability of an assertion directly corresponds to the probability p used in Section 4.1.1.

4.4. Randomizing Test Suite Composition

Previously we discussed *intra*-randomization techniques, in the sense that they randomized the generation of a single test case in particular. The following measures to be discussed are *inter*-randomization techniques, in the sense that they randomize the generation of the whole test suite.

Test Suite Size

The number of test cases generated for the test suite can also be randomized. Therefore a lower- and upper-bound for the test case number is being configured. After that, for each run of the test suite generation, a random number between the lower- and upper-bound is picked. That number then indicates how many test cases have to be generated in total for that generation run.

Flakiness Categories

As discussed in previous sections the generation of test cases within the test suite is split among the different flakiness categories stemming from the actual flakiness root causes. The three elaborated flakiness categories have a few sub-categories originating from different implementation approaches. This is why there are several different flakiness sub-categories from which a test case could be generated, this offers more room for randomization which will be briefly discussed in the following.

For each flakiness sub-category, a relative frequency is configured. When the generation for an individual sub-category starts, the relative frequency is then multiplied by the previously randomized total count of test cases contained in the test suite to indicate how many test cases from the respective category should be generated. Since the total count of test cases is random within a previously defined range, the number of test cases generated from an individual sub-category is random as well.

4.5. Randomizing by Adding Dummy Code

Another way of randomizing the generation of the test suite is to add dummy code to the test files. This code does not contribute to the flakiness of the tests it rather adds diversity to the randomization. Random functions and methods are generated that are then called within the tests but their return values do not contribute to the test's assertions and thus do not affect the tests result when executing them. The actual behaviour of the dummy code can be very diverse, to name a few one could perform side effects such as printing to the standard output, calculating some random operation or returning some static value.

Besides the added diversity of the generation, including dummy code brings another benefit. It adds complexity to the generated test suite in the form of non-flaky methods which allows for better evaluation when using APR or FL tools on the test suite. There are not only flaky methods in the test suite that should be detected by the respective tools, but there are also non-flaky methods that should not be detected by such tools. Listing 4.2 shows a class that implements a dummy function that returns a static value, this function is then called in a test case of the generated test suite.

5. Implementation

In the following chapter implementation details on the theoretically presented approach will be given. For that certain decisions that were made during the implementation process will be explained and discussed. First, the selection of the used testing framework will be discussed. Second, measurements to enable the configurability of the test suite generation will be presented. After that, the software architecture will be described briefly along with a presentation of a short code example. The chapter will close with a brief validation of the implementation.

5.1. Testing Framework

Concerning the testing framework for which the test cases are randomly generated with the presented tool, there were two major options to choose from. One being `pytest` the other being `PyUnit`. In this case, we decided to use `pytest` rather than `PyUnit` mainly due to its simpler usage. Less "boilerplate" code is needed for test cases in `pytest`, a test function must be inside a file that follows one of the following two file name patterns, test_*.py or *_test.py and the function itself must be prefixed with the word test to be discovered.

Additionally, there are multiple packages for `pytest` that are handy while implementing the presented approach. First, being `pytest-random-order` allows for the configuration of the test runs such that tests are run in random order. Like this, the test order-dependent test cases become flaky in the first place. Second, `pytest-json-report` simplified the evaluation of the research questions because it puts out a .json file that contains all information on the outcome of a test suite run after it was performed, allowing for analysis regarding the number of tests run or which tests passed or failed in which runs.

5.2. Configurability of Test Suite Generation

In the following section the configurability measurements that were taken when developing the presented tool will be presented.

Users of the tool can access it through the command line interface and give a path to a configuration file in JavaScript Object Notation format as an argument. When not providing this argument, the tool uses a predefined config.json file that is contained in the project. This JSON file has the following structure and semantics.

Listing 5.1 provides an example of such a config file. First, it contains values for the maximum and minimum number of test cases generated, between these a random number is then chosen to define the number of tests that should be generated in a generation run.

Second, for each flakiness category, there will be a JSON object always containing at least one value depicting the share of the total test case number to be from the respective flakiness category.

Additionally, there can be flakiness category-specific configuration values, for example flakiness_prob, indicating the probability with which a test case from the randomness category is flaky.

```json
{
  "max_total_test_count": 550,
  "min_total_test_count": 500,
  "random_api": {
    "summation": {
      "test_number_share": 0.1,
      "max_summation_depth":  50,
      "max_summand": 1000,
      "flakiness_prob":  0.5
    },
    "multiplication": {
      "test_number_share": 0.1,
      "max_multiplication_depth":  100,
      "max_multiplicand": 10,
      "flakiness_prob":  0.5
    },
    "arithmetical": {
      "test_number_share": 0.1,
      "max_expression_depth":  100,
      "flakiness_prob":  0.5
    },
  },
  "test_order_dependent": {
    "basic_victim_polluter": {
      "test_number_share": 0.1,
      "flakiness_prob":  0.5
    },
    "basic_brittle_state_setter": {
      "test_number_share": 0.1,
      "flakiness_prob":  0.5
    },
    "classes_brittle_state_setter": {
      "test_number_share": 0.1,
      "flakiness_prob":  0.5
    }
  }
}
```

Listing 5.1: Examplary `config.json` file

5.3. Software Architecture

In the centre of the implementation are generator classes for each flakiness category from which the test cases can stem from. These generator classes then generate test cases on the AST level using the `ast` package. The `ast` package provides the data structures that are used to construct and generate the test cases. Amongst others it contains classes for expressions, statements and function definitions. Following that the ASTs of the test cases are converted into source code using the `astor` package. The `astor` package supplies methods to convert the generated ASTs to source code. Finally, the generated source code is then written to a file and persisted in this way.

Now one exemplary generator class and its respective test case generation functions will be presented and explained. The presented generator class implements the generation of test cases from the randomness category, to be more specific it generates flaky test suites that use random summation expressions that are then made flaky by adding some noise with a certain probability.

```
class SummationGenerator(RandomApiGenerator):
    def __init__(self, flakiness_prob):
        self.flakiness_prob = flakiness_prob

    # Generates function that adds a summand as often as summation_depth
        indicates and then adds a noise with some prob
    # like summation_depth=3, summand=5: 5 + 5 + 5 or 5 + 5 + 5 + 0.1
    def generate_flaky_function_tree(
        self,
        summation_depth,
        function_identifier
    ):
        epsilon = ast.Constant(0.1)
        zero = ast.Constant(0)
        statements = []
        result = ast.Name('result')

        if_expr = ast.IfExp(
            self.generate_compare_lt_expression(
            self.generate_random_float_number_expression(), ast.Constant(
                self.flakiness_prob)),
            zero,
            epsilon
        )

        summation_expression = ast.Expression(
            ast.BinOp(left=ast.Name(id='summand'),
            op=ast.Add(),
            right=if_expr)
        )
        assignment = ast.Assign([result], summation_expression)
        statements.append(assignment)

        for i in range(summation_depth-1):
            summation_expression = \
                ast.Expression(
                    ast.BinOp(left=ast.Name(id='summand'),
                    op=ast.Add(),
                    right=result)
                )
            assignment = ast.Assign([result], summation_expression)
            statements.append(assignment)

        statements.append(ast.Return(result))

        return ast.FunctionDef(
            'flaky_summation_' + function_identifier,
            ast.arguments([], [ast.arg(arg='summand')], defaults=[]),
            statements,
            []
        )

    # Generates one line function that asserts equality between the call of
        the flaky function and non-flaky summation
    def generate_test_tree(
        self, summand,
        summation_depth,
        function_identifier
    ):
        actual = ast.Name('actual')
        expected = ast.Name('expected')
        actual_value = ast.Call(
            func=ast.Name('random_api_summation.flaky_summation_' +
                function_identifier),
            args=[ast.Constant(summand)], keywords=[]
        )

        statements = [
            ast.Assign(
                targets=[actual],
                value=actual_value,
                type_ignores=[]
            ),
            ast.Assign(
                targets=[expected],
                value=ast.Constant(summation_depth * summand),
                type_ignores=[]
            ),
        ]

        random.shuffle(statements)

        statements.append(
            self.generate_assert_equality_expression(expected, actual)
        )

        test_function = ast.FunctionDef(
            'test_sum_' + function_identifier,
            ast.arguments([], [], defaults=[]),
86              statements,
87              []
88          )
89          return test_function
```

Listing 5.2: Generator class from the randomness category

Note that there are two methods implemented in this class one for generating the flaky test case and the other for generating the function that makes the test case flaky. We will first cover the function generate_flaky_function_tree.

First, in lines 12 to 15 variables holding `Constant` and `Name` AST nodes and an array for holding the functions statements are initialized. Second, in lines 17 to 22 a ternary conditional expression that yields the previously generated `epsilon` with some flakiness probability, given as a parameter of the method, otherwise zero is generated. Following that in lines 24 to 42 summation expressions of length summation_depth are generated and appended to the statement array. As well as that a return statement returning the result of the previous additions is appended. In conclusion, a function definition containing the previously generated statements is generated.

Now we will discuss the generation of the test case itself, which uses the previously generated flaky function. First, in lines 57 and 58 two variables are initialized that hold `Name` AST nodes for the actual and the expected values that are compared later in the test case. Second, in lines 59 to 62, a AST node for the function call of the previously generated flaky function is generated. Following that, in lines 64 to 75 a statement array holding the actual value just generated, as well as the expected value calculated by multiplying the summation depth with the addend, is initialized. Additionally, in line 77 the statement array is then shuffled to randomize the generation. Furthermore, in lines 79 to 81 the equality assertion expression is generated and appended to the statement array. Moreover, a `FunctionDef`AST node is generated that holds all the previously generated statements. Finally, the generated function definition is returned.

5.4. Validation of Implementation

In the following we validate whether the configurability of the test suite composition discussed previously works. To recall the concept, the idea was to enable the user of the generation tool to configure what percentage of all tests generated should stem from a certain flakiness category or subcategory. For this, a relative frequency for all categories and subcategories could be defined which then is multiplied by the overall number of test cases to be generated and thus indicates the number of test cases to be generated in the respective category. To answer the question:

Are the configured relative frequencies for the different flakiness categories depicted in the generated test suite?

One can generate multiple test suites and then count the total number of test cases generated within one category and compare this number with the multiplication of the pre-configured relative frequency of the category and the total test case count:

$$f_c \cdot n$$

where f_c is the pre-configured relative frequency of category c and n is the total test count. Again considering the test case name test_order_dependent_classes_brittle_state_setter _61030948535442d0b9d02d470eacf05f_test::test_brittle_0 from the previous section there is a pattern in the naming of tests which is the following: FlakinessCategory_Identifier_TestCase. Using the flakiness category prefix it is possible to assign each test to a flakiness category and thus count the number of tests within the respective category.

Results

The first characteristic to observe when evaluating is that the pre-configured relative frequencies for the randomness categories coincide with the actual total numbers of tests generated in these categories in ten out of ten considered test suite generations. The only minor deviation is that the calculated number of tests to be generated within that category is decimal and thus there is at maximum a difference of $\epsilon \in [0, 1]$ between the actual number of tests and the target number of tests generated within a randomness category. This is because the calculation of the number of tests to be generated within a category might yield a decimal number depending on the respective relative frequency and the total number of tests. However, the total number of tests generated in reality can only be a whole number since generating a fraction of a test is impossible.

For the test order-dependency categories, there is slightly more deviation compared to the randomness test cases. In eight out of ten test suite generations the maximum difference, between the actual number of tests and the target number within a flakiness category, is still $\epsilon \in [0, 1]$. For two out of the ten generated test suites, the maximum difference for at most two test order-dependent categories is slightly higher, namely $\epsilon \in [0, 2]$. This is because the other test order-dependent and randomness categories are very accurate in these runs, in the sense that the actual test number and the target test number are equal for most of the categories and just slightly different for the others. Combining this with the fact that the respective more inaccurate categories were generated in the end one can deduce that this is because the total number of tests to be generated within the whole test suite had to be reached and thus a bigger deviation had to be accepted for the last two categories generated.

6. Evaluation

The ensuing chapter presents research questions to evaluate the work done throughout this thesis. Following that the asked questions will be answered by conducting experiments on the generated test suites. This will mainly involve executing the test suites, counting generated tests categorizing them and carrying out an existing FL tool on the generated code.

To evaluate the first question the experiments were carried out on a system with the following specifications:

- Product: MacBook Pro, 16-inch, 2021
- Operating System: macOS Ventura (Version: Version 13.6.3 (22G436))
- Chip: Apple M1 Max
- Memory: 32 GB
- Python version: Python 3.10

To evaluate the second question the experiments were carried out on a system with the following specifications:

- Operating System: Windows 11 Home 22H2 (Build: 22621.2861)
- CPU: AMD Ryzen 5 5600X 6-Core Processor @ 3.70 GHz
- Memory: 32 GB DDR4 @ 3200 MHz
- Python version: Python 3.10

A complete list of all used packages and their respective version is given in Listing A.1 contained in the appendix.

In the following sections the research questions that are going to be evaluated, the approach on how to evaluate them and the results of the evaluation will be presented. Therefore each of the three questions will first be described, an answering approach will be presented and the results will be shared in detail.

6.1. RQ1: Are the generated test cases flaky?

The first attribute to be evaluated is whether the generated test cases are flaky. Thus one has to check for test cases to yield non-deterministic results when running them multiple times without changing the actual source code. For this different test suites will be generated and then run multiple times without altering them. Following that the generated output depicting the results of the tests will be examined. The collected data from the examination will then be analyzed to answer the question:

RQ 1: Are the generated test cases flaky?

For this one can make use of the fact that each test case within the test suite is uniquely identifiable by a hash contained in its name. For example consider the test case name test_order_dependent_classes_brittle_state_setter_61030948535442d0b9d02d470eacf05f_test ::test_brittle_0 where the hash of interest is 61030948535442d0b9d02d470eacf05f. Now one can run the test suite and obtain the output of the form:

```
test_arithmetical_f120e705af15436fa39b596207eda786 FAILED
rtest_arithmetical_ccced85a639d4223bf67b4318187ae31 FAILED
...
test_combination_41e475efbac848028088f3e58b25db73 FAILED
test_combination_c4307a00a2c74f75bccce3f06d63d90d FAILED
...
test_multiplication_9d2be42da58549d8998ba41e7e15b421 FAILED
test_multiplication_b28112e8fd7c43b699f0890dafc9e5a7 PASSED
...
test_sum_0a4b2d3d60e34a9e92c5c3f1970d85c5 FAILED
test_sum_23dbbc3c583c4510ae09dde4d3f7ac73 PASSED
...
```

Programmatically processing this output for each run of the test suite, one can compare whether one test case failed and succeeded in multiple runs and thus is flaky. This action can be carried out for all test cases and thus one can verify whether the generated test suite is flaky. For example, the first line indicates that the test test_arithmetical_f120e705af15436fa39b596207eda786 failed. The last line indicates that the test test_sum_23dbbc3c583c4510ae09dde4d3f7ac73 succeeded. When re-executing the test suite, one can check whether, for example, test_arithmetical_f120e705af15436fa39b596207eda786 now succeeds and test_sum_23dbbc3c583c4510ae09dde4d3f7ac73 now fails. If that would be the case the respective tests are marked as flaky. This procedure is then repeated for each test case to obtain a result on whether the test suite is flaky.

Results

After evaluating the generated test suites one can name three key observations. For this consider Table A.1 that depicts the test results of a generated test suite containing 540 test cases being executed ten times. The first column shows the test number and a postfix that maps the test case to its respective flakiness category. Test cases are associated with their root cause by a postfix, these are "rand" for randomness, "async_wait" for *async-wait* and "order" for the test order-dependency root cause. Additionally, the test cases that are evaluated especially are indicated with an additional postfix, "arith" stands for test cases that use random arithmetical expressions to become flaky, "comb" for test cases that use combinations of random arithmetical expressions, "state" for state-setter test cases and "polluter" for polluter test cases. Most of the test cases are flaky but there are three minor exceptions the first being intended and the second being a logical consequence to the test case generation. These exceptions are described in the following.

First, when looking at the previously mentioned table, as intended the state-setter and polluter test cases always succeed and thus are never flaky and pass in all runs.

Second, concerning the test cases where multiple randomness flakiness subcategories were combined, in the table depicted as rows with the fist column containing the "comb" postfix, these contain some test cases that never succeed when run ten times. Recalling Section 4.3 and the equation given at the end of the respective section, the explanation for this is simple. Due to the accumulation and calling of multiple flaky methods within the test case, it fails in all executions. Each of the flaky methods called within the test case succeeds with a certain probability p, thus the success probability of the test case is obtained by multiplying the success probabilities of all flaky methods called within that test. Since $0 < p < 1$ the overall success probability of the test case shrinks with each flaky method called. That is why it fails in all executions since its success probability is so low. But when the number of considered test suite runs is increased the number of test cases that never succeed within this flakiness category decreases.

Third, considering the test cases with the "arith" postfix, one can observe that there is also one test case that was non-flaky within this flakiness category, namely test case 304. This is because even though it is unlikely that a test case that is flaky with a certain probability p only fails throughout ten runs it still can happen. It is a so-called anomaly and thus is no threat to the evaluation of the presented approach.

6.2. RQ2: Can `CharmFL` Detect the Flakiness Within the Test Suites?

Closing the evaluation a known FL tool for Python code shall be tested for the generated test suites of this project. The tool is called `CharmFL` [4], was developed in 2021 and can be used as a plug-in for the PyCharm-IDE. To address the test system switch between research questions one and two, it has to be said that `CharmFL` requires a version of the PyCharm-IDE that is not above some threshold version. On Windows, it was easier available, so the test system was simply changed. `CharmFL` uses a Spectrum-Based Fault Localization approach to identify potentially faulty statements, functions or classes. In this case, we want to see whether it detects the generated functions that make the test cases flaky.

Results

Running the tool on the generated test suites there are three key observations to make. First, all flaky functions from the randomness category are detected as such. Second, the tool can not detect the flakiness of the test order-dependent test cases. The reason for this is that the functions themselves are non-flaky and just become flaky as soon as the execution order is randomized. `CharmFL` uses SBFL as its underlying FL technique which runs a test suite for a given software project to identify faulty program elements. In the presented approach the test order-dependent test cases do not call any program element that makes the execution flaky, rather the test cases in combination are flaky. The two major problems with this code that arise when viewing this code from a SBFL standpoint become clear when considering the following generated test cases:

```
global j
j = 914


def test_0_victim():
    global j
    assert (j == 914)


def test_1_polluter():
    global j
    j = 261
    assert (j == 261)


def test_2_victim():
    global j
    assert (j == 914)


def test_3_victim():
    global j
    assert (j == 914)
```

Listing 6.1: Flaky victim-polluter test cases

First, the test cases themselves are not flaky, because test_0_victim, test_2_victim and test_3_victim will only fail when test_1_polluter was run before them, thus only become flaky when run multiple times in random order. Second, they do not call any outer program elements that the SBFL approach could identify as the flakiness cause.

The last observation to state is that `CharmFL` does not detect tests from the *async-wait* category as flaky. It appears that the test case is not even executed properly. The actual reason is not clear, but a reasonable possibility is that tests from that category use asynchronous functions and for them to be executed in `pytest` test cases, these test cases need to be annotated with the @pytest.mark.asyncio annotation. Somehow these annotations are not recognized and thus the whole test case is skipped.

7. Conclusions and Outlook

A novel approach to how to randomly generate flaky test suites was presented. Therefore theoretical background information on related topics was given first. Following that the approach how to randomly generate flaky test suites was elaborated in detail.

For each considered flakiness category, in this case, randomness and test order-dependency, an individual approach on how to randomly generate test cases from this flakiness root cause was presented. First, for the randomness category we worked with random sequences of arithmetical operations that were made flaky by adding some noise to it with a certain probability. Second, for the test order-dependency flakiness category, we worked on defining a state for the respective tests that then was manipulated by the different test cases in a way that only certain execution orders yielded successful results. We extensively used the brittle- and state-setter-pattern and victim- and polluter-pattern defined by Shi et. al. [19] to achieve this.

Following that measures to randomize the generation further were developed, among them are the randomization of variable and state names, combining flakiness categories and subcategories to new flakiness categories, randomizing the test suite composition in a sense that for example the number of tests generated and the number of tests stemming from a certain flakiness category were randomized and non-flaky dummy code was added.

Implementation details were also presented including a validation of the implementation that investigated whether the configurability of the test suite composition worked in the way proposed. Carrying out the validation resulted in discovering that the configuration works as intended.

Last but not least, the presented approach was also evaluated by considering two major research questions. First, *"Are the generated test cases flaky?"* This question can be answered with yes, all test cases that are intended to be flaky are flaky with some minor exceptions that relate to well-known stochastical relations. For example when combining different flakiness categories to new ones each of the flakiness categories results in failing tests with a certain probability, when now combining those the probability with which the respective test cases fail increases up to a point where from around 25 test cases at most two always failed concerning ten individual test suite runs.

Second, the in practice FL tool `CharmFL` was used to detect the faults within the randomly generated test cases. This resulted in discovering a limitation regarding this tool, whereas fault and flakiness due to randomness were detected reliably, flakiness and faults due to test order-dependency or async-wait were not detected as reliably as the other root cause. On the one hand, this might be due to the fact, that in the way we implemented the test order-dependent test cases the test cases themselves are not flaky, when viewed in isolation they are perfectly deterministic. On the other hand, `CharmFL` uses a spectrum-based FL approach that always looks for the cause of the flakiness in some external program element, but in the way the test cases are generated in this approach the test order-dependent test cases do not call any external program elements in contrast to the randomness root caused test cases. Additionally, it seemed like `CharmFL` had problems executing tests from the async-wait category at all. Thus test cases from this category were not detected as flaky.

Outlook

The most substantial limitation of the presented work is that only three of the ten existing flakiness root causes are covered by it. But this bridges the gap to further work that can be done to improve the presented approach. According to Luo et. al. [18] the most common root causes for flakiness are *test order-dependency*, *async-wait* and *concurrency*. As the approach already covers *test order-dependency* and *async-wait*, future work could delve into also adding the other most common root cause, namely *concurrency*. Of course, it can also be beneficial to extend the generation to cover not only the most common flakiness root causes but also the others to be more complete.

To generate tests that are flaky due to the *concurrency* root cause, one could generate multiple test cases, where at least one test calls a concurrent function that initializes some state. All other tests try to access that state without waiting for the initialization test case to finish. This fails if the state is not yet accessible. For this state initialization, various concepts could be used, for example, a simple global variable could be set to some value or a file could be created. This state would then in turn be accessed by the other test cases. Depending on how long the state initialization takes from run to run, different test cases succeed and fail in separate runs.

Besides that, it would be beneficial to extend the generation to generate non-flaky code to make the test suite more like actual test suites from real-world software projects.

A. Appendix

```
astor==0.8.1
click==8.1.7
colorama==0.4.6
iniconfig==2.0.0
numpy==1.26.1
packaging==23.2
pandas==2.1.4
pluggy==1.3.0
pytest==7.4.2
pytest-asyncio==0.23.5
pytest-excel==1.6.0
pytest-json-report==1.5.0
pytest-metadata==3.0.0
pytest-order==1.1.0
pytest-random-order==1.1.0
python-dateutil==2.8.2
pytz==2023.3.post1
shellingham==1.4.0
six==1.16.0
tzdata==2023.4
```

Listing A.1: requirements.txt (used packages in the project)

Viewed Test	1	2	3	4	5	6	7	8	9	10
0_order_state	p	p	p	p	p	p	p	p	p	p
1_order	p	f	p	p	f	f	f	f	p	p
2_order	p	p	p	f	p	p	f	p	p	f
3_order_polluter	p	p	p	p	p	p	p	p	p	p
4_order	f	p	p	f	f	p	p	p	p	f
5_order	f	f	p	p	p	p	p	f	p	p
6_order	f	p	p	f	p	p	p	p	p	p
7_async_wait	f	p	f	f	f	p	f	p	f	f
8_async_wait	f	f	p	f	p	f	f	p	p	p
9_order_state	p	p	p	p	p	p	p	p	p	p
10_order	p	p	p	p	f	p	p	p	f	f
11_async_wait	p	f	f	p	p	p	f	p	f	f
12_order_state	p	p	p	p	p	p	p	p	p	p
13_order	p	f	f	p	p	f	p	f	p	f
14_async_wait	f	p	f	p	f	f	p	f	p	f
15_order_state	p	p	p	p	p	p	p	p	p	p
16_order	p	f	f	f	p	f	f	f	f	f
17_async_wait	p	p	f	f	p	p	p	f	f	f
18_async_wait	p	f	f	p	f	f	f	p	f	p

19_order_polluter	p	p	p	p	p	p	p	p	p	p
20_order	f	f	p	f	f	f	f	f	p	p
21_order	p	p	p	f	p	p	p	f	p	f
22_order	p	p	p	f	f	f	f	p	p	p
23_order	p	f	p	f	f	p	p	f	f	p
24_order_polluter	p	p	p	p	p	p	p	p	p	p
25_async_wait	f	f	f	p	f	f	f	f	f	p
26_rand_arith	f	p	p	f	p	p	p	p	f	p
27_rand_arith	p	f	f	p	f	p	f	p	p	f
28_rand_arith	p	p	p	p	f	p	f	p	f	p
29_rand_arith	p	p	p	p	p	p	p	f	f	p
30_rand_arith	f	p	f	p	p	f	p	f	f	f
31_rand_arith	f	f	p	p	p	p	f	f	f	f
32_rand_arith	f	p	p	f	f	p	f	p	f	f
33_rand_arith	f	p	p	f	f	p	p	p	f	f
34_rand_arith	f	p	f	p	f	f	p	p	f	p
35_rand_arith	p	p	p	f	f	p	p	p	p	f
36_rand_arith	p	p	p	p	p	p	p	f	f	f
37_rand_arith	p	f	p	p	p	p	p	f	p	p
38_rand_arith	p	p	f	f	f	f	f	f	f	f
39_rand_arith	f	f	f	f	f	p	f	p	p	f
40_rand_arith	f	p	f	p	f	p	f	f	p	f
41_rand_arith	p	p	f	p	p	p	p	f	f	p
42_rand_arith	p	f	f	p	f	p	p	f	p	f
43_rand_arith	p	p	f	f	p	p	p	p	f	p
44_rand_arith	f	p	f	f	f	f	p	f	p	p
45_rand_arith	f	p	f	f	p	f	p	f	f	f
46_rand_arith	p	f	p	p	f	p	p	f	f	p
47_rand_arith	p	p	f	f	p	p	f	p	f	f
48_rand_arith	f	p	f	f	f	p	p	p	p	f
49_rand_arith	p	f	f	p	p	f	f	f	f	p
50_rand_arith	p	f	f	p	f	f	f	f	f	p
51_rand_arith	p	f	p	p	p	p	p	f	f	p
52_rand_arith	p	p	p	p	p	p	f	p	f	f
53_rand_arith	p	f	p	f	f	f	f	f	f	p
54_rand_arith	f	f	p	p	p	f	f	p	p	p
55_rand_arith	p	f	p	f	p	f	p	p	f	p
56_rand_arith	p	f	f	f	f	p	p	p	f	p
57_rand_arith	p	p	f	f	p	f	p	f	p	p
58_rand_arith	p	f	f	p	f	f	p	f	f	p
59_rand_arith	f	f	f	p	p	f	f	f	f	f

60_rand_arith	f	p	p	f	p	p	p	f	p	f
61_rand_arith	f	f	p	f	p	p	f	f	f	p
62_rand_arith	f	p	p	f	f	f	p	f	p	p
63_rand_arith	p	p	f	f	p	p	p	f	p	f
64_rand_arith	f	p	p	p	f	f	p	f	p	p
65_rand_arith	p	p	f	p	p	p	p	f	f	p
66_rand_arith	p	f	f	p	f	f	p	f	p	f
67_rand_arith	p	f	f	p	f	p	f	f	p	f
68_rand_arith	p	f	p	p	f	f	f	f	p	p
69_rand_arith	p	p	p	f	f	f	f	p	f	p
70_rand_arith	p	p	f	p	f	f	f	f	p	f
71_rand_arith	f	p	f	f	p	f	p	p	f	p
72_rand_arith	f	f	f	f	f	f	f	f	f	f
73_rand_arith	p	f	f	f	p	f	p	f	f	f
74_rand_arith	p	f	f	f	p	p	f	f	f	p
75_rand_arith	p	f	f	p	p	p	p	p	p	p
76_rand_arith	f	p	p	p	p	f	f	p	p	p
77_rand_arith	f	p	f	p	f	f	f	f	f	p
78_rand_arith	f	f	f	f	p	p	p	f	p	p
79_rand_arith	p	f	f	f	f	p	p	p	f	f
80_async_wait	f	f	p	p	p	p	f	p	f	p
81_order	p	f	f	f	f	f	p	p	f	p
82_order_polluter	p	p	p	p	p	p	p	p	p	p
83_order	f	f	f	p	f	f	p	f	f	p
84_order	f	p	f	p	f	f	f	p	p	f
85_order_state	p	p	p	p	p	p	p	p	p	p
86_order	f	p	f	p	p	f	p	p	p	f
87_order_state	p	p	p	p	p	p	p	p	p	p
88_order_polluter	p	p	p	p	p	p	p	p	p	p
89_order	f	p	f	p	p	f	p	p	p	f
90_order	p	p	f	p	p	f	p	f	p	p
91_order	p	p	f	p	p	f	p	f	p	p
92_order_polluter	p	p	p	p	p	p	p	p	p	p
93_async_wait	f	p	p	f	f	p	f	f	f	p
94_order	f	f	p	p	p	p	f	p	f	f
95_order	f	f	f	p	f	f	p	p	p	p
96_order_state	p	p	p	p	p	p	p	p	p	p
97_order_state	p	p	p	p	p	p	p	p	p	p
98_order	f	p	f	p	f	p	p	f	p	p
99_order	f	p	f	p	f	f	p	f	f	f
100_order_state	p	p	p	p	p	p	p	p	p	p

101_order_state	p	p	p	p	p	p	p	p	p	p
102_order	p	p	f	p	p	f	f	p	f	p
103_order_state	p	p	p	p	p	p	p	p	p	p
104_order	f	f	p	f	f	f	f	f	f	f
105_order_state	p	p	p	p	p	p	p	p	p	p
106_async_wait	p	f	f	f	f	f	p	p	p	f
107_async_wait	p	f	p	p	p	p	p	p	p	f
108_order	p	p	f	f	f	p	p	f	p	f
109_order	p	f	p	f	f	p	f	f	p	f
110_order_polluter	p	p	p	p	p	p	p	p	p	p
111_order	f	p	f	f	f	p	p	p	p	f
112_order_polluter	p	p	p	p	p	p	p	p	p	p
113_order	f	f	p	f	f	p	f	f	p	f
114_order	f	f	p	p	p	p	f	p	p	f
115_async_wait	p	p	f	f	p	f	f	p	p	f
116_async_wait	f	p	f	p	p	p	p	p	p	p
117_order	f	f	p	f	f	f	f	f	p	p
118_order_state	p	p	p	p	p	p	p	p	p	p
119_async_wait	f	f	f	p	f	f	f	p	p	p
120_order	p	p	f	f	p	f	f	p	p	f
121_order	p	p	p	f	p	f	f	p	p	f
122_order_polluter	p	p	p	p	p	p	p	p	p	p
123_async_wait	p	f	p	p	p	p	f	f	f	p
124_order	p	p	f	f	p	p	f	p	p	p
125_order_polluter	p	p	p	p	p	p	p	p	p	p
126_order	f	p	f	f	f	p	p	f	f	f
127_order	f	p	f	p	f	p	p	f	f	f
128_order	f	p	f	f	p	p	p	p	f	f
129_async_wait	f	f	f	f	p	p	f	p	p	p
130_order_state	p	p	p	p	p	p	p	p	p	p
131_order	p	f	p	f	p	f	p	f	f	p
132_order	p	p	f	p	p	f	f	p	p	p
133_order	p	p	f	f	f	f	f	p	p	p
134_order	p	f	f	p	p	f	f	p	p	p
135_order_polluter	p	p	p	p	p	p	p	p	p	p
136_order	f	p	p	p	f	p	f	p	f	p
137_order_state	p	p	p	p	p	p	p	p	p	p
138_order	p	f	p	p	p	f	p	f	f	p
139_order	p	f	f	f	p	f	p	f	p	p
140_order_polluter	p	p	p	p	p	p	p	p	p	p
141_order	f	f	p	f	p	f	p	p	p	f

142_async_wait	p	f	p	f	f	p	f	f	p	f
143_order_polluter	p	p	p	p	p	p	p	p	p	p
144_order	f	p	p	f	p	f	p	p	f	f
145_order	p	p	f	p	p	p	p	f	p	p
146_order_polluter	p	p	p	p	p	p	p	p	p	p
147_order	f	p	p	p	p	p	p	p	p	p
148_order	f	p	f	f	p	p	p	p	p	p
149_async_wait	f	p	f	f	f	f	p	p	f	p
150_order	p	p	p	f	f	f	p	p	p	f
151_order	p	f	p	f	p	p	p	f	p	f
152_order_polluter	p	p	p	p	p	p	p	p	p	p
153_order	f	f	f	f	p	p	p	f	f	p
154_async_wait	f	p	f	f	p	f	p	f	p	p
155_order_state	p	p	p	p	p	p	p	p	p	p
156_order	p	p	p	f	p	f	p	f	f	f
157_async_wait	p	p	p	p	f	f	p	p	f	f
158_async_wait	p	p	f	f	p	f	f	p	f	f
159_order_state	p	p	p	p	p	p	p	p	p	p
160_order	p	p	p	p	f	p	f	p	p	p
161_order	f	f	p	p	f	f	f	p	p	p
162_order	f	p	p	p	f	f	f	p	f	p
163_order	f	p	p	f	f	f	f	f	f	p
164_order_state	p	p	p	p	p	p	p	p	p	p
165_order_state	p	p	p	p	p	p	p	p	p	p
166_order_state	p	p	p	p	p	p	p	p	p	p
167_order	p	p	f	f	f	f	p	p	f	f
168_order	p	p	f	p	p	p	p	p	p	p
169_order	p	p	f	f	f	f	p	p	f	f
170_order_polluter	p	p	p	p	p	p	p	p	p	p
171_order_state	p	p	p	p	p	p	p	p	p	p
172_order	p	f	f	f	f	f	f	f	p	p
173_async_wait	f	p	f	p	f	f	f	p	p	f
174_async_wait	f	f	f	f	f	p	p	f	f	p
175_order	f	f	f	f	f	p	f	f	p	f
176_order_state	p	p	p	p	p	p	p	p	p	p
177_order	p	f	p	p	f	f	p	p	f	p
178_order_polluter	p	p	p	p	p	p	p	p	p	p
179_order_polluter	p	p	p	p	p	p	p	p	p	p
180_order	f	f	f	f	p	f	f	p	f	p
181_order	f	f	f	f	p	p	f	f	f	p
182_order	f	p	f	f	p	p	f	f	f	p

183_order	f	f	p	f	p	p	f	f	f	p
184_rand_comb	p	f	f	p	f	f	f	p	p	p
185_rand_comb	f	f	f	f	f	f	p	f	f	f
186_rand_comb	f	f	f	f	f	f	f	f	f	f
187_rand_comb	f	p	f	f	f	f	f	f	f	f
188_rand_comb	f	f	f	f	f	f	f	f	f	f
189_rand_comb	f	f	f	f	f	f	p	f	f	f
190_rand_comb	f	f	f	f	f	f	f	f	f	f
191_rand_comb	f	f	f	f	f	f	f	f	f	f
192_rand_comb	f	f	f	f	f	f	f	f	f	f
193_rand_comb	f	f	p	f	p	p	f	f	p	f
194_rand_comb	f	f	f	f	f	f	f	f	f	f
195_rand_comb	f	p	f	p	p	p	f	p	p	p
196_rand_comb	f	f	f	f	f	f	f	p	f	f
197_rand_comb	f	f	f	p	f	f	f	f	f	f
198_rand_comb	p	f	f	f	f	f	p	f	f	p
199_rand_comb	f	f	p	f	f	f	p	f	f	f
200_rand_comb	p	p	p	p	p	p	f	p	p	p
201_rand_comb	f	f	f	f	f	f	f	f	f	f
202_rand_comb	f	f	f	f	f	f	f	f	f	f
203_rand_comb	f	f	f	f	f	f	f	f	f	f
204_rand_comb	f	f	f	p	f	f	p	f	f	f
205_rand_comb	f	p	f	p	p	p	f	p	p	p
206_rand_comb	f	f	f	f	f	f	f	f	f	f
207_rand_comb	f	f	f	f	f	f	f	p	f	f
208_rand_comb	f	f	f	f	f	f	f	f	f	f
209_rand_comb	f	f	f	f	f	f	f	f	p	f
210_rand_comb	p	f	f	f	f	f	f	f	f	f
211_rand_comb	f	f	f	f	f	f	f	f	f	f
212_rand_comb	f	f	p	f	f	f	f	f	f	p
213_rand_comb	f	f	f	f	f	f	f	f	f	f
214_rand_comb	p	p	f	f	p	f	f	p	p	f
215_rand_comb	f	f	f	f	f	f	f	f	f	f
216_rand_comb	f	f	f	f	f	f	f	f	f	f
217_rand_comb	f	f	f	f	f	f	f	p	f	f
218_rand_comb	f	f	f	f	f	f	f	f	f	f
219_rand_comb	f	f	f	f	f	f	f	f	f	f
220_rand_comb	f	f	f	f	f	f	f	f	f	f
221_rand_comb	f	f	f	f	f	f	f	f	f	f
222_rand_comb	f	f	f	f	f	f	f	f	f	f
223_rand_comb	f	f	f	f	f	f	f	f	f	f

224_rand_comb	f	f	f	f	f	f	f	f	f	f
225_rand_comb	f	f	f	f	f	f	f	f	f	f
226_rand_comb	f	f	f	f	p	f	p	f	f	p
227_rand_comb	f	f	f	f	f	f	f	f	f	f
228_rand_comb	f	f	f	f	f	f	f	f	f	f
229_rand_comb	f	f	f	f	f	f	f	f	f	p
230_rand_comb	f	f	f	f	f	f	f	f	f	f
231_rand_comb	f	f	f	f	f	f	f	f	f	f
232_rand_comb	f	f	p	f	p	p	f	p	f	p
233_rand_comb	f	f	f	p	f	f	f	f	f	p
234_rand_comb	f	f	f	f	f	f	f	f	f	f
235_rand_comb	f	f	f	f	f	f	f	f	f	f
236_rand_comb	f	f	f	f	f	f	f	f	f	f
237_rand_comb	f	f	f	f	f	f	f	f	f	f
238_order_polluter	p	p	p	p	p	p	p	p	p	p
239_order	f	f	f	p	p	p	f	f	p	p
240_order	f	f	f	p	p	f	p	f	f	f
241_order	f	p	f	f	f	p	p	f	f	f
242_order_polluter	p	p	p	p	p	p	p	p	p	p
243_order	f	p	f	p	f	f	p	f	p	p
244_order	f	p	f	f	f	f	f	f	f	p
245_order_state	p	p	p	p	p	p	p	p	p	p
246_order	f	p	f	f	p	f	p	p	p	p
247_order_state	p	p	p	p	p	p	p	p	p	p
248_async_wait	f	f	p	p	p	p	p	f	p	f
249_order_state	p	p	p	p	p	p	p	p	p	p
250_order	p	p	f	f	f	p	f	p	f	f
251_async_wait	f	p	f	f	f	f	p	p	f	f
252_order_polluter	p	p	p	p	p	p	p	p	p	p
253_order	f	f	f	f	f	f	p	p	p	p
254_order	p	f	f	p	p	p	p	p	f	f
255_order	p	f	f	p	f	p	p	p	p	f
256_order	p	f	f	p	p	f	f	p	p	p
257_order	p	f	f	p	f	p	p	p	f	p
258_order_polluter	p	p	p	p	p	p	p	p	p	p
259_order	f	p	f	p	f	p	p	p	f	p
260_order_state	p	p	p	p	p	p	p	p	p	p
261_order	f	p	f	f	f	f	p	p	f	p
262_order_state	p	p	p	p	p	p	p	p	p	p
263_order_state	p	p	p	p	p	p	p	p	p	p
264_order	p	p	f	p	f	f	f	f	f	p

265_order	f	p	p	f	p	p	f	p	p	p
266_order_state	p	p	p	p	p	p	p	p	p	p
267_rand	p	f	p	f	p	p	p	f	f	f
268_rand	p	f	p	f	p	p	p	f	p	p
269_rand	p	p	f	p	p	f	p	p	f	f
270_rand	p	f	f	p	p	f	p	f	p	f
271_rand	f	p	f	f	p	f	p	f	f	p
272_rand	p	f	p	f	f	f	f	p	p	f
273_rand	f	p	f	f	p	f	p	f	f	f
274_rand	f	p	p	f	f	p	f	f	f	f
275_rand	f	f	f	f	p	p	p	f	f	p
276_rand	p	p	f	p	f	f	f	f	p	p
277_rand	f	p	f	p	f	p	f	f	f	p
278_rand	f	p	p	p	p	f	p	p	p	p
279_rand	f	p	f	f	p	p	p	f	f	p
280_rand	p	p	f	f	p	p	f	p	f	p
281_rand	f	p	p	f	p	f	f	f	p	p
282_rand	f	p	p	f	f	p	p	p	p	p
283_rand	f	f	f	p	f	p	p	p	p	f
284_rand	p	f	f	p	p	p	p	p	p	p
285_rand	p	p	p	f	f	p	f	p	f	p
286_rand	p	f	f	p	p	p	f	f	f	f
287_rand	p	p	p	p	p	p	p	p	p	f
288_rand	p	p	p	p	p	f	f	p	p	f
289_rand	p	p	f	p	p	f	p	p	f	f
290_rand	f	f	p	f	p	f	f	p	p	p
291_rand	f	p	f	p	f	f	f	p	f	p
292_rand	p	f	f	f	f	f	f	p	f	f
293_rand	f	p	f	f	f	f	p	p	p	f
294_rand	f	f	f	f	f	f	f	f	f	f
295_rand	p	p	f	f	p	p	f	p	f	f
296_rand	p	p	f	f	p	p	p	f	p	f
297_rand	p	f	f	p	p	f	p	f	p	p
298_rand	p	f	f	p	f	f	p	p	p	p
299_rand	p	p	p	p	p	p	f	f	f	p
300_rand	f	p	f	f	f	f	f	f	p	p
301_rand	f	f	p	p	p	f	f	f	f	p
302_rand	p	p	p	p	f	f	p	p	p	f
303_rand	f	f	f	f	f	p	p	p	f	p
304_rand	f	p	f	f	f	f	p	p	p	p
305_rand	p	p	p	f	p	p	p	f	f	f

306_rand	p	p	p	p	f	f	p	p	p	f
307_rand	f	p	p	f	p	p	f	f	p	f
308_rand	f	f	p	p	f	f	f	p	p	f
309_rand	f	p	f	p	p	f	p	f	p	f
310_rand	f	f	p	f	p	p	p	f	f	f
311_rand	p	f	f	p	p	f	p	f	p	p
312_rand	p	f	p	f	p	f	p	f	p	f
313_rand	p	p	f	p	f	f	p	f	p	p
314_rand	f	p	p	f	p	f	p	f	f	f
315_rand	p	f	p	p	p	f	p	p	p	p
316_rand	p	p	f	f	p	p	p	f	p	p
317_rand	f	p	f	p	f	f	f	p	f	f
318_rand	p	p	f	p	f	p	p	p	f	f
319_rand	p	p	p	p	p	f	f	p	p	f
320_rand	f	p	p	p	p	p	p	f	p	p
321_order_polluter	p	p	p	p	p	p	p	p	p	p
322_order	f	f	f	f	f	f	p	f	p	f
323_order	f	f	p	p	f	f	p	f	f	f
324_order_polluter	p	p	p	p	p	p	p	p	p	p
325_order	f	p	p	p	p	p	p	p	f	p
326_order	f	f	f	f	f	f	f	f	f	f
327_order	f	f	p	p	p	f	f	p	f	p
328_order	f	f	p	f	f	p	p	f	f	f
329_order	p	p	p	f	f	p	f	p	p	p
330_order	p	p	p	f	f	f	p	p	f	p
331_order_polluter	p	p	p	p	p	p	p	p	p	p
332_order	f	p	p	f	p	f	p	p	f	p
333_order	f	p	p	f	f	f	p	p	f	p
334_order	f	p	f	p	f	f	f	p	p	f
335_order_state	p	p	p	p	p	p	p	p	p	p
336_async_wait	p	f	p	p	p	p	f	p	p	f
337_async_wait	p	f	p	p	p	f	p	p	f	p
338_async_wait	f	f	p	f	p	p	f	f	p	f
339_rand	p	p	p	p	f	f	f	p	p	f
340_rand	p	p	p	p	f	p	f	p	f	f
341_rand	p	f	f	p	f	f	p	p	p	p
342_rand	f	f	f	f	p	f	f	p	f	f
343_rand	p	f	f	p	f	p	p	f	p	p
344_rand	f	f	f	f	f	p	f	f	f	f
345_rand	p	f	p	f	f	p	p	f	f	p
346_rand	p	f	f	p	p	p	f	f	f	f

347_rand	f	f	p	f	p	p	p	p	p	f
348_rand	f	f	f	p	p	f	f	p	p	f
349_rand	p	f	p	p	f	f	f	f	f	f
350_rand	f	f	f	f	f	p	f	f	p	f
351_rand	f	p	f	f	p	p	f	f	f	p
352_rand	f	f	f	f	p	p	f	f	f	p
353_rand	f	p	f	p	p	f	f	p	f	f
354_rand	f	p	p	p	f	f	f	f	f	f
355_rand	p	p	p	p	p	p	f	p	f	p
356_rand	p	f	f	p	p	p	f	f	f	f
357_rand	f	f	p	f	f	p	p	f	p	p
358_rand	f	p	f	f	p	p	p	f	f	p
359_rand	f	p	p	f	p	f	p	p	f	f
360_rand	p	p	p	p	f	p	f	p	p	f
361_rand	f	p	p	p	p	f	f	p	p	f
362_rand	f	f	f	p	p	f	f	p	p	f
363_rand	f	f	f	f	p	f	f	p	f	p
364_rand	p	f	p	p	f	f	f	p	f	f
365_rand	p	p	p	p	f	f	p	p	p	p
366_rand	f	p	f	p	p	p	f	p	f	f
367_rand	f	f	p	p	p	f	f	p	f	f
368_rand	f	p	f	p	f	p	p	f	f	p
369_rand	f	f	f	f	p	p	p	p	f	f
370_rand	f	f	p	f	p	p	f	p	f	p
371_rand	f	p	p	p	p	f	f	f	f	f
372_rand	f	p	f	p	f	f	p	p	f	p
373_rand	p	p	p	f	f	p	f	f	p	f
374_rand	p	p	f	f	f	f	f	p	p	f
375_rand	f	f	f	f	f	f	p	p	p	f
376_rand	p	f	p	p	f	f	f	p	f	p
377_rand	p	p	p	p	p	f	p	p	p	f
378_rand	f	p	p	f	f	p	f	p	p	p
379_rand	f	p	p	f	p	f	p	f	f	f
380_rand	p	p	f	p	f	f	f	p	p	f
381_rand	f	f	f	p	p	f	f	f	f	p
382_rand	f	f	f	f	p	f	p	p	f	p
383_rand	f	p	p	f	f	f	f	f	p	p
384_rand	f	f	p	f	p	f	p	p	p	f
385_rand	f	p	p	f	p	f	p	p	f	f
386_rand	f	p	f	f	p	f	p	p	f	p
387_rand	p	p	f	p	p	p	f	f	p	p

388_rand	f	f	f	p	f	f	p	p	p	f
389_rand	f	f	p	p	p	p	p	f	f	p
390_rand	f	p	p	p	p	p	p	p	f	f
391_rand	f	p	f	p	f	f	p	f	f	p
392_rand	p	f	p	f	p	f	p	p	p	p
393_async_wait	f	f	f	f	f	p	p	p	p	p
394_async_wait	f	p	f	p	p	p	f	p	p	f
395_order	f	f	p	f	f	f	f	p	f	f
396_order_state	p	p	p	p	p	p	p	p	p	p
397_order	p	p	p	f	f	p	f	p	f	p
398_order_polluter	p	p	p	p	p	p	p	p	p	p
399_order_polluter	p	p	p	p	p	p	p	p	p	p
400_order	f	p	f	f	f	p	f	p	f	p
401_async_wait	f	p	p	p	f	f	p	f	p	f
402_order	f	p	p	f	p	f	f	p	p	f
403_order_state	p	p	p	p	p	p	p	p	p	p
404_async_wait	f	p	f	p	p	p	p	p	f	p
405_order	f	p	p	p	p	p	f	f	p	f
406_order_state	p	p	p	p	p	p	p	p	p	p
407_order	p	p	p	f	p	f	p	p	p	f
408_order	p	p	p	p	p	f	p	f	f	f
409_order	p	p	p	p	p	f	p	f	f	f
410_order_polluter	p	p	p	p	p	p	p	p	p	p
411_async_wait	f	f	p	p	f	p	f	f	p	f
412_order	p	p	p	f	p	p	f	p	p	p
413_order	p	p	p	f	p	p	f	f	f	p
414_order_polluter	p	p	p	p	p	p	p	p	p	p
415_order	f	p	p	f	p	p	f	p	p	p
416_order	f	f	f	f	p	f	p	p	f	f
417_order_state	p	p	p	p	p	p	p	p	p	p
418_async_wait	f	f	f	p	f	f	p	f	p	p
419_order_state	p	p	p	p	p	p	p	p	p	p
420_order	p	f	f	p	p	f	f	f	p	f
421_async_wait	f	f	p	p	f	p	p	f	f	p
422_async_wait	f	f	p	f	f	p	f	f	p	f
423_order	f	f	p	p	p	p	p	f	p	f
424_order_state	p	p	p	p	p	p	p	p	p	p
425_order_state	p	p	p	p	p	p	p	p	p	p
426_order	p	f	p	f	f	p	f	p	p	p
427_order_polluter	p	p	p	p	p	p	p	p	p	p
428_order	f	p	f	f	p	p	f	p	f	f

429_order	f	p	f	f	p	p	p	p	p	p
430_order	f	f	f	p	f	f	f	p	p	p
431_order	p	p	f	f	p	p	f	f	f	f
432_order	p	f	f	p	f	f	f	p	f	p
433_order	p	f	f	p	p	f	f	p	f	f
434_order_polluter	p	p	p	p	p	p	p	p	p	p
435_async_wait	f	f	p	f	p	p	f	p	f	f
436_order	f	p	f	f	f	p	f	f	f	p
437_order_state	p	p	p	p	p	p	p	p	p	p
438_order	p	p	f	f	f	f	p	p	p	f
439_order	f	p	p	f	f	f	f	f	f	p
440_order_state	p	p	p	p	p	p	p	p	p	p
441_order_state	p	p	p	p	p	p	p	p	p	p
442_async_wait	f	p	f	p	f	p	f	f	f	f
443_order	p	p	f	f	f	f	p	p	p	p
444_order	p	p	f	f	p	f	p	p	f	p
445_order_polluter	p	p	p	p	p	p	p	p	p	p
446_order_state	p	p	p	p	p	p	p	p	p	p
447_order	p	f	f	p	f	p	p	f	p	f
448_order_polluter	p	p	p	p	p	p	p	p	p	p
449_order	f	f	f	p	p	f	f	p	f	p
450_order	f	p	f	f	f	p	f	f	f	f
451_order	f	p	f	f	f	p	f	p	f	p
452_order	p	f	p	p	f	f	f	f	p	f
453_order	p	p	p	p	p	f	f	f	f	f
454_order_polluter	p	p	p	p	p	p	p	p	p	p
455_order	p	p	f	p	f	f	p	p	p	f
456_order_polluter	p	p	p	p	p	p	p	p	p	p
457_order	f	f	f	p	p	f	f	f	p	f
458_order	f	p	f	f	p	f	f	f	p	p
459_async_wait	f	f	p	f	p	f	p	p	f	f
460_order	p	p	f	p	f	f	f	f	f	f
461_order_polluter	p	p	p	p	p	p	p	p	p	p
462_order	f	p	p	p	f	p	p	p	f	f
463_order	f	p	f	p	f	f	p	p	f	p
464_order	f	p	p	p	f	f	f	f	f	p
465_async_wait	f	p	f	p	p	p	p	p	p	f
466_order_state	p	p	p	p	p	p	p	p	p	p
467_order	f	p	p	f	p	f	p	f	f	p
468_order	f	p	f	p	p	f	p	f	p	p
469_order_state	p	p	p	p	p	p	p	p	p	p

470_order_state	p	p	p	p	p	p	p	p	p	p
471_order	p	f	p	f	p	p	f	p	f	p
472_order	p	p	f	p	f	p	p	p	p	f
473_order_state	p	p	p	p	p	p	p	p	p	p
474_async_wait	p	p	f	f	p	f	p	f	p	f
475_order_polluter	p	p	p	p	p	p	p	p	p	p
476_order	f	p	f	f	p	p	p	f	p	p
477_order	f	p	p	f	p	f	p	f	f	f
478_order	p	f	f	f	f	f	f	f	p	f
479_order_polluter	p	p	p	p	p	p	p	p	p	p
480_order	f	p	f	f	f	f	f	f	p	p
481_async_wait	f	p	f	f	f	p	p	p	f	p
482_async_wait	f	f	f	p	f	f	f	p	p	p
483_order	f	f	f	p	p	p	f	f	f	f
484_order_state	p	p	p	p	p	p	p	p	p	p
485_order_state	p	p	p	p	p	p	p	p	p	p
486_order	p	p	f	p	f	p	p	p	f	p
487_order	p	p	f	f	p	f	f	p	f	f
488_order_state	p	p	p	p	p	p	p	p	p	p
489_order	p	f	f	p	p	p	f	p	f	f
490_order_state	p	p	p	p	p	p	p	p	p	p
491_async_wait	p	f	f	f	p	f	f	f	p	f
492_order	f	f	p	f	p	p	f	p	p	f
493_order	f	f	p	p	f	f	p	f	p	f
494_order_state	p	p	p	p	p	p	p	p	p	p
495_order_state	p	p	p	p	p	p	p	p	p	p
496_order_state	p	p	p	p	p	p	p	p	p	p
497_order	p	p	p	p	f	f	p	f	p	p
498_async_wait	p	p	p	p	p	p	p	f	f	p
499_async_wait	p	f	f	f	f	p	f	f	f	p
500_async_wait	p	p	p	p	p	f	p	f	p	p
501_order	p	f	p	p	p	p	p	p	p	f
502_order_polluter	p	p	p	p	p	p	p	p	p	p
503_order	f	p	p	p	p	p	p	p	f	f
504_async_wait	f	p	f	p	f	f	f	p	f	p
505_async_wait	p	f	f	f	f	p	p	f	p	p
506_async_wait	f	f	f	p	f	p	p	f	f	p
507_order_polluter	p	p	p	p	p	p	p	p	p	p
508_order	f	f	p	f	f	f	f	p	p	f
509_order_state	p	p	p	p	p	p	p	p	p	p
510_order	p	p	p	p	f	p	p	f	p	p

511_async_wait	f	p	p	p	p	p	p	f	p	p
512_order	p	f	f	p	p	p	p	f	f	p
513_order_polluter	p	p	p	p	p	p	p	p	p	p
514_order	p	f	p	p	f	f	f	f	p	f
515_order_polluter	p	p	p	p	p	p	p	p	p	p
516_order	f	p	p	p	p	f	f	f	p	f
517_order	f	f	p	f	f	f	f	f	f	f
518_order	p	f	p	p	p	p	p	f	p	p
519_order	p	f	p	p	p	p	p	f	p	f
520_order_polluter	p	p	p	p	p	p	p	p	p	p
521_order_polluter	p	p	p	p	p	p	p	p	p	p
522_order	f	p	p	f	p	f	f	f	p	f
523_async_wait	p	f	f	p	p	p	f	f	f	f
524_async_wait	p	f	p	p	p	f	f	p	f	f
525_order	f	p	p	f	f	p	f	p	f	f
526_order_state	p	p	p	p	p	p	p	p	p	p
527_order	p	f	p	p	p	p	p	p	p	f
528_order	p	f	f	p	p	f	p	f	f	f
529_order	p	p	p	p	p	f	p	p	p	f
530_order_polluter	p	p	p	p	p	p	p	p	p	p
531_order	f	p	p	p	p	p	p	p	f	f
532_order	p	p	p	f	f	p	p	p	f	p
533_order_polluter	p	p	p	p	p	p	p	p	p	p
534_order	p	f	f	p	p	f	p	p	f	p
535_order_polluter	p	p	p	p	p	p	p	p	p	p
536_order	f	f	p	p	p	f	p	f	f	p
537_async_wait	f	p	p	p	f	p	p	f	f	p
538_order	p	f	f	f	f	f	p	p	f	f
539_order_polluter	p	p	p	p	p	p	p	p	p	p

Table A.1.: Table depicting test suite execution results for each test case

Bibliography

[1] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[2] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[3] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[4] Q. Idrees Sarhan, A. Szatmári, R. Tóth, and Beszédes, "Charmfl: A fault localization tool for python," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 114–119.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[6] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[7] M. Renieres and S. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, 2003, pp. 30–39. [Online]. Available: http://dx.doi.org/10.1109/ASE.2003.1240292

[8] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 467–477. [Online]. Available: https://doi.org/10.1145/581339.581397

[9] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006, pp. 39–46.

[10] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[11] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[12] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond genprog," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300159

[13] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–11. [Online]. Available: https://doi.org/10.1145/3180155.3180233

[14] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *Journal of Systems and Software*, vol. 171, p. 110825, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302193

[15] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "On the impact of flaky tests in automated program repair," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 295–306.

[16] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in python," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 148–158.

[17] G. Darbord, A. Etien, N. Anquetil, B. Verhaeghe, and M. Derras, "A Unit Test Metamodel for Test Generation," in *International Workshop on Smalltalk Technologies*. Lyon, France: Stephane Ducasse and Gordana Rakic, Aug. 2023. [Online]. Available: https://hal.science/hal-04219649

[18] 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*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653. [Online]. Available: https://doi.org/10.1145/2635868.2635920

[19] 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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 545–555. [Online]. Available: https://doi.org/10.1145/3338906.3338925