Transforming Test Suites into Croissants

Yang Chen
University of Illinois Urbana-Champaign, USA
yangc9@illinois.edu

Alperen Yildiz
Sabanci University, Turkey
alperenyildiz@sabanciuniv.edu

Darko Marinov
University of Illinois Urbana-Champaign, USA
marinov@illinois.edu

Reyhaneh Jabbarvand
University of Illinois Urbana-Champaign, USA
reyhaneh@illinois.edu

ABSTRACT

Software developers often rely on regression testing to ensure that recent changes made to the source code do not introduce bugs. *Flaky tests*, which non-deterministically pass or fail regardless of any change to the code, can negatively impact the effectiveness of the regression testing. While state-of-the-art is advancing the techniques for test-flakiness detection and mitigation, the community is missing a systematic approach for generating high-quality benchmarks of flaky tests to compare the effectiveness of such techniques. Inspired by the power of mutation testing in evaluating the fault-detection ability of tests, this paper proposes Croissant, a framework for injecting flakiness into the test suites to assess the effectiveness of test-flakiness detection tools in finding these tests. Croissant implements 18 flakiness-inducing mutation operators. We designed these operators to allow controlling the non-determinism involved in flakiness, i.e., making many mutants deterministically pass or fail to observe flaky behavior. Our extensive empirical evaluation of Croissant on the test suites of 15 real-world projects confirms the ability of designed mutation operators to generate high-quality mutants, and their effectiveness in challenging test-flakiness detection tools in revealing flaky tests.

CCS CONCEPTS

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

KEYWORDS

Software Testing, Test Flakiness, Fault Injection, Mutation Testing

ACM Reference Format:
Yang Chen, Alperen Yildiz, Darko Marinov, and Reyhaneh Jabbarvand. 2023. Transforming Test Suites into Croissants. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598119

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598119

1 INTRODUCTION

Regression testing is essential to detect potential bugs and ensure the reliability of ever-evolving software. The key idea behind regression testing is to run the test suite against the modified software version to check if a recent change may have introduced a bug. However, *flaky tests*, which non-deterministically pass or fail, can drastically degrade the quality of regression testing [41]. A key reason is that test flakiness happens due to code smells in the *test suite* rather than a bug in the code under test. Consequently, developers may spend a great deal of time debugging recent code changes to search for the root cause of the failure raised by a flaky test, while the root cause is the test code itself.

Test-flakiness detection techniques [3, 16, 30, 34–38, 41, 45, 52, 53, 57, 62, 63, 66, 67, 70] rely on several executions of the test suites (same order or different orders of the tests) to observe flaky behavior or search for some pre-defined or learned anti-patterns in the test code to identify the flakiness promptly. To assess the effectiveness of these techniques at scale, there is a need for a large benchmark containing a diverse set of flaky tests under different contexts. Several attempts have been made to collect or automatically generate such benchmarks. IDoFT [36] are FlaPy [22] are two collections of flaky tests in open-source Java and Python projects, respectively. Despite being rich resources of real-world flaky tests, they have limitations. They do not allow one to control the non-determinism of flaky tests in the benchmark to compare the effectiveness of test-flakiness detection tools in finding hard-to-observe flaky bugs. They are also limited in size by the number of tests found to be flaky. FlakiMe [8] and Flaker [24] automatically inject flakiness into existing test suites to create flaky mutants. FlakiMe is limited to injecting exception statements in tests to create non-deterministic, non-order-dependent (NOD) mutants. Flaker deletes helper statements from tests to make their outcomes order-dependent (OD). Both of these approaches are limited to a few categories of flaky tests, failing to generate a diverse benchmark.

In this paper, we propose Croissant[1], a systematic approach for injecting various types of flakiness into test suites. In the design of Croissant, we overcome two challenges:

(1) Fault Model Construction. Flaky tests can be complex and manifest themselves under peculiar contextual conditions such as test-order execution, race condition, iteration order, etc.; thereby, small syntactic changes or adding exception statements may not substitute or represent flakiness effectively [29]. This complexity demands the design of Croissant's mutation operators based on a fault model of *flaky anti-patterns*. To that end, we first conducted

[1]The approach name is inspired by the croissant pastry famous for its flaky texture.

a semi-systematic study of various sources of information to construct a comprehensive flaky-defect model[2]. Using our defect model, we designed and implemented a total of 18 mutation operators to automatically inject flakiness into JUnit test suites.

(2) Controlling Non-determinism. Effectiveness of a technique such as Croissant on assessing test-flakiness detection tools depends on how well it can control the probability of observing flaky behavior. Depending on the nature of flakiness, i.e., if flakiness happens due to dependency between tests or in isolation, Croissant adjusts the manifestation probability of the mutants to challenge test-flakiness detection tools in finding hard-to-detect flaky tests. For NOD flaky tests, Croissant controls the non-determinism through a threshold value. By changing the threshold value, the chance of observing flaky behavior and finding a specific flaky test also changes. For OD flaky bugs, Croissant controls the non-determinism by adjusting the number of tests that break the order dependency. The higher the number of such tests, the harder it is for a test-flakiness detection tool to find dependent flaky tests.

Controlling non-determinism can also help us with mutation analysis and ensuring the quality of the mutants. For example, suppose we inject NOD tests into a test suite but do not observe test failures in multiple test executions. In general, we cannot determine if the mutation was incorrect (so the test cannot fail) or if more test re-executions are needed to observe flaky behavior. However, if we can control mutants to fail nearly deterministically, it can assure the mutation was correct, but the flakiness was not observed.

This paper makes the following contributions:

- **Flaky anti-patterns:** A comprehensive list of flaky anti-patterns—commonly encountered development practices that make test suites flaky—structured as a defect model. Compared to related work that categorizes flaky tests based on the nature of manifestation [36, 38]—i.e., OD or NOD—our defect model categorizes flakiness based on the root causes, potentially making it easier for practitioners to recognize or debug flaky tests. Our defect model also introduces new categories for flakiness not comprehensively studied by prior work.
- **Mutation operators:** Design of 18 flakiness-inducing mutation operators based on flaky anti-patterns. The ability to control the probability of observing flaky behavior makes Croissant suitable to challenge the ability of test-flakiness detection tools. Compared to related work that introduces test flakiness using limited patterns [24] or through probabilistic exceptions [8], our comprehensive mutation operators are designed based on a defect model, mimicking real test flakiness and leaving similar execution footprints as them. That is, while our operators are probabilistic similar to prior work, we differ by controlling the execution of flaky anti-patterns rather than generic exceptions.
- **Public tool:** We implemented our mutation operators in a tool for Java projects. We call this tool Croissant, the same as our approach. Our implementation is publicly available [64].
- **Empirical evaluation:** An extensive empirical evaluation of Croissant on developer-written test suites of 15 open-source Java projects widely used in research demonstrates that our proposed technique is effective and efficient in generating unique and high-quality mutants to challenge test-flakiness detection tools. Our experiments detected several bugs in iDFlakies [36], a state-of-the-art tool for detecting OD flaky tests in Java projects. All the reported bugs and our pull requests for fixing them were confirmed and accepted by the iDFlakies developers.

[2]We do not claim our defect model to be complete but the most comprehensive to date.

2 DEFECT MODEL

We designed Croissant's mutation operators based on real-world flaky tests and sources of flakiness. Our methodology for collecting a comprehensive list of sources of flakiness consists of three main steps: (1) finding issues related to test flakiness in open-source repositories, (2) identifying the anti-patterns corresponding to identified flakiness, and (3) designing and injecting mutation operators based on these anti-patterns. For the first step, we performed a keyword-based search on GitHub to collect flaky issues. In addition, we crawled Google Scholar to find prior work related to test-flakiness detection and mitigation, and then extracted the information such as commit and issue numbers from those papers. These two resources provided us with a list of issues and commits relevant to flaky tests in JUnit test suites. Next, we carefully studied the code, test suite, and issue comments to identify *flaky anti-patterns*. When present in a test suite, these anti-patterns can result in flaky tests. In the remainder of this section, we explain the details of our search protocol and defect model construction. We present details of mutation operators in Section 3.

Mining Open-Source Repositories. We searched GitHub with the query "`flaky test language:Java label:bug comments:>2 state:closed`" to retrieve issues that: (1) contain the keywords "flaky" and "test", (2) belong to Java projects, (3) are labeled as bugs by developers (confirming the issue to be a bug), (4) have more than two comments (which provides a discussion on the issue that could help us to understand the issue), and (5) are closed (indicating that the issue is resolved). Such issues likely include information that can help understand the reported flaky cases. This search provided 266 issues[3] in April 2022 for further manual investigation.

Collecting Related Work. To study the related work on test-flakiness detection and mitigation for JUnit tests, we searched Google Scholar with the query "`JUnit flaky (test | tests) (source: ACM | IEEE | Elsevier)`"[4]. This search resulted in 145 papers, out of which we removed five replication studies (because they do not introduce new issues to investigate) and two retracted papers. In addition, by analyzing the artifacts of these papers, we identified 64 more GitHub issues for further manual investigation.

Defect Model Construction. From the previous steps, we collected 330 GitHub issues and commits related to JUnit test flakiness. Two paper authors manually investigated these issues to determine common flaky anti-patterns in the code of the involved tests. As a result, we identified 18 unique flaky anti-patterns representing the sources of flakiness and designed mutation operators based on them. Table 1 shows these mutation operators categorized into 17 classes. The classes marked by $*$ implement flakiness not discussed by prior research. The first column of Table 1 shows a coarser-grained categorization of flakiness in our defect model following prior work on classifying flaky tests. OD flaky tests manifest themselves in a specific order of the tests in the test suite due to a shared state between

[3]A repeated search would likely result in a different number.

[4]We only considered the peer-reviewed papers published by ACM, IEEE, or Elsevier.

Table 1: List of flakiness-inducing mutation operators. The classes marked by an asterisk are either not discussed in prior work at all (*) or only a subset of their instances has been identified in prior work (**).

Type	Class	Description	Mutation Operators
OD	Instance Variable Dependency	The shared state is an instance variable	IVD
	Static Variable Dependency*	The shared state is a static variable	SVD
	Third-Party Framework Dependency*	The shared state is through a third-party framework	TPFD
	Cached Status Dependency	The shared state is cached data	CSD
	Database State Dependency	The shared state has items in the database	DSD
	File Permission Dependency	The shared state is a file handle	FPD
	Resource Availability	The shared state is a specific resource	RA
NOD	Memory Dependency	The test requires a large amount of memory that may not exist	MD
	Platform Dependency	The test assumes specific platform properties to run	PD
	System Time Dependency	The test relies on specific system time to execute	STD-V, STD-Z
	Concurrency Timeout Deadlock	Non-deterministic timeouts due to deadlocks	CTD
	Asynchronous Waits	The test makes asynchronous calls but does not wait properly	AW
	Too Restrictive Range	The test relies on a restrictive range that can potentially change	TRR
	Race Condition	Test assertion relies on the data manipulated by multiple threads	RC
ID	Unordered Collection Index Access**	Test converts unordered collection into an ordered collection to access an index	UCIA
	Unordered Collections Conversion**	Test converts the unordered collection into a string	UCC
	Reflection API Misconception	Test assumes deterministic order for the output of reflection APIs	RAM

tests. When a test depends on the order of test execution in the test suite, it is classified as an OD flaky test; otherwise, it is classified as a flaky test that is not order-dependent (NOD). Implementation-dependent (ID) flaky tests are a specific subcategory of NOD flaky tests due to misuse of certain code constructs such as collections.

3 CROISSANT

We next introduce two main algorithms in Croissant for generating flaky mutants. We also briefly discuss each class of mutation operators. The source code and detailed explanation of Croissant's mutants are publicly available [65].

3.1 Mutant Generation and Analysis

3.1.1 ***OD Mutant Generation.*** OD flaky tests occur when two or more tests in the test suite are coupled through a shared state that the developers do not properly manage, e.g., in `tearDown` or `setUp` methods [71]. As a result, if the execution order of tests changes, e.g., due to test prioritization [55] or test parallelization [5], the outcome of tests may also change from pass to fail or the other way round. Tests that can change the outcome based on the shared state are called either *victim* or *brittle* [57]. Victim tests pass when run alone (but can fail when run after some other tests), while brittle tests fail when run alone (but can pass when run after some other tests). A test that changes the shared state for the *victim* test is called *polluter*, while the test that changes the shared state for the *brittle* is called *state-setter*. A victim test passes if executed before the polluter and fails otherwise. In contrast, a brittle test fails if executed before the state-setter and passes otherwise.

Detection of OD flaky tests is even more challenging due to the existence of *cleaners* [57] and a new category of tests, which we call *state-unsetters*. When a cleaner appears between a polluter and a victim, it neutralizes the state change impact, so the victim passes even when run after the polluter. Likewise, when a state-unsetter appears between a state-setter and a brittle, it neutralizes the state change impact, so the brittle fails even when run after the state-setter. Without cleaner or state-unsetter tests, a simple technique that re-executes the tests in one order and its reverse could detect all OD flaky tests. Prior work [57] has introduced the notion of cleaners, but to the best of our knowledge, no prior work discussed state-unsetters. To challenge flaky-detection tools, our mutation operators not only can create a dependency between tests in the test suite but also can inject cleaners and state-unsetters.

Algorithm 1: Injecting OD flakiness

```
Inputs: Original test suite T,
        Set of flaky anti-patterns A,
        Number of cleaner or state-unsetter tests k
Output: Set of mutated test suites X = {T_1, ..., T_m}
1  X ← ∅
2  VB ← identifyVictimsBrittles(T, A)
3  foreach vb_i ∈ VB do
4      if vb_i is victim then
5          p_i, c_i ← createPolluterCleaner(vb_i)
6          T_i ← mutate(T, vb_i, p_i, c_i, k)
7      else
8          s_i, u_i ← createStateSetterUnsetter(vb_i)
9          T_i ← mutate(T, vb_i, s_i, u_i, k)
10     X ← X ∪ {T_i}
11 return X
```

Croissant follows the steps in Algorithm 1 to inject OD flaky tests into test suites. It takes as inputs the original test suite T, a set of flaky anti-patterns A identified by the defect model, and the number of cleaner or state-unsetter tests k to be injected in the test suite. Given these inputs, the algorithm generates a set of mutated test suites X. In the first step, Croissant performs a lightweight static analysis to identify the potential victim or brittle tests that match the anti-patterns in our defect model (Line 2, more details in §3.2.1–§3.2.7). For each vb_i, if the candidate can be a victim (Line 4), Croissant creates polluter test p_i and cleaner test c_i (Line 5). Otherwise, the candidate can be a brittle, so Croissant creates state-setter s_i and state-unsetter u_i tests (Line 8). The p_i or s_i tests modify the shared state according to the type of vb_i, e.g., variable, cache, or file access of specific victim vb_i. On the other hand, c_i or u_i tests reverse the changes made by p_i or s_i. Next, Croissant mutates

T by adding the generated tests into the test suite (Lines 6 and 9) and adds the mutant T_i to the final set of mutants X (Line 10). All classes of OD mutation operators, which we will explain in more detail in Section 3.2, will add new tests to the test suite. Depending on the type of flakiness, these tests will serve as a polluter, cleaner, state-setter, or state-unsetter. Croissant reads the number of cleaners or state-unsetter tests (input parameter k in Algortihm 1) from the config file, and injects them accordingly in the test suite.

3.1.2 ***NOD and ID Mutant Generation.*** NOD and ID flakiness often happen due to misuse or misunderstanding of programming APIs, concurrency problems, etc. Compared to OD flakiness which involves multiple tests, NOD and ID flakiness occur for each test in isolation. As a result, one can detect these tests by re-executing them without shuffling the test execution order. Still, detecting NOD and ID flakiness is challenging when the probability of observing flaky behavior is tiny, e.g., test-flakiness detection tools may re-execute the test suite more than 100 times to get a passing test to fail [38].

Algorithm 2 shows Croissant's approach for injecting NOD and ID flaky tests into test suites. It takes as inputs the original test suite T, a set of flaky anti-patterns A, and a threshold value tr that determines the probability of observing flakiness. Given these inputs, Algorithm 2 generates a set of NOD and ID mutants X. The algorithm first employs a simple static analysis to identify unit tests that invoke specific APIs corresponding to different NOD or ID anti-patterns (Line 2). Depending on the API invocation and whether it relates to NOD or ID anti-patterns (details in §3.3–§3.4.3), the algorithm mutates f_i into a NOD or ID mutant (Line 4). Croissant controls the non-determinism in NOD and ID mutants through a threshold value tr, ranging from 0 to 1. Specifically, the probability of failure for NOD mutants is exactly tr; for ID mutants, it is at most tr. The threshold value can be adjusted dynamically during mutant run-time execution, and the mutant checks it against a randomly generated number. The mutated code will be executed if the random value is smaller than the threshold. Consequently, a lower threshold value makes it harder for naïve flakiness detection tools to pinpoint the injected flaky tests. After mutating the test suite, the algorithm adds the mutated test suite T_i to the final set of mutants (Line 6). Unlike OD mutant generation, NOD and ID mutation operators only modify existing tests to introduce flakiness in the test suite. That said, the implementation of Croissant contains some *template tests*, which can be added to test suites if certain NOD and ID mutation operators do not apply to them. After adding these template tests, Croissant can mutate them into flaky tests.

3.1.3 ***Mutation Analysis and Debugging.*** Mutation analysis can be challenging in the presence of non-determinism. For example, suppose Croissant injects a NOD mutant using Algorithm 2, where we do not observe a test failure after re-executing the mutated test suite multiple times. In that case, we cannot easily conclude whether the mutation was ineffective (i.e., the test is not flaky at all) or if the flaky behavior was not observed (although the test is indeed flaky). To enable such analysis and debug potential issues, Croissant enables the control of the *non-determinism* associated with flaky tests in a fully *deterministic* fashion, helping users to control mutants to pass or fail more deterministically. To that end, Croissant generates *helpers* as a byproduct of the mutation, which are the versions of the generated mutants that are more likely to fail. Comparing the execution results of helpers[5] and the original test suite for each generated mutant allows us to evaluate if the flakiness injection was successful.

Algorithm 2: Injecting NOD and ID flakiness

Inputs: Original test suite T,
Set of flaky anti-patterns A,
Threshold value tr
Output: Set of mutated test suites $X = \{T_1, \ldots, T_m\}$

```
1 X ← ∅
2 F ← getCandidates(T, A)
3 foreach f_i ∈ F do
4     T_i ← mutate(T, f_i, tr)
5     X ← X ∪ {T_i}
6 return X
```

Croissant generates OD helpers by discarding c_is or u_is from mutants, i.e., helpers include no cleaner or state-unsetter tests. It also modifies the order of test execution to reverse the test order in the original test suite. As a result, if the mutation generates a polluter/victim test pair, the original test suite passes while the helper fails. On the other hand, if the mutation generates a state-setter/brittle test pair, the original test suite fails while the helper passes. For NOD and ID mutants, Croissant generated helpers by setting the threshold value to 1, making them fail.

3.2 OD Mutation Operators

Our empirical study identified *seven* classes of OD flaky tests, depending on the kind of shared state that couple polluter/cleaner/victim and state-setter/state-unsetter/brittle tests. We next explain the example flaky tests and corresponding mutation operators mimicking them. The code snippets used to demonstrate how mutants are injected only show a high-level overview of the code in Croissant.

3.2.1 ***Instance Variable Dependency (IVD).*** IVD flakiness happens when developers define instance variables to be shared among different test methods inside a test class[6].

```
public void pTest() { // Polluter
    instanceVar = CHANGED_STATUS;
}
public void cTest() { // Cleaner
    instanceVar = DEFAULT_STATUS;
}
public void vTest() { // Victim
    // specific assertion depends on the type of instance variable
    assertEquals(DEFAULT_STATUS, instanceVar);
}
```

Issue 592 in project elasticjob [17] is a real-world example of such a case, where polluters do not shut down the shared instance; if they run before the victim that assumes the instance to be shut down, the victim fails. The snippet above shows the templates of IVD mutants in Croissant, where the victim test `vTest` has a dependency with the polluter test `pTest` through `instanceVar`.

3.2.2 ***Static Variable Dependency (SVD).*** SVD flaky tests happen when a polluter changes static variables that are used later by a victim. Issue 4384 in project nacos [50] demonstrates an instance of

[5] Each helper is a test suite.
[6] This feature is available in JUnit 5 but was not available in JUnit 4.

such bugs in the real world. The snippet below shows an example SVD mutant, where the victim `vTest` has a dependency with the polluter `pTest` through a static variable `field`.

```
public class ClassWithStaticVariable {
    FieldClass fieldClass = new FieldClass();
    static class FieldClass {
        static int field = defaultValue;
    }

    public void pTest() { // Polluter
        fieldClass.field = diffValue;
    }
    public void cTest() { // Cleaner
        fieldClass.field = defaultValue;
    }
    public void vTest() { // Victim
        assertEquals(defaultValue, fieldClass.field);
    }
}
```

3.2.3 ***Third-Party Framework Dependency (TPFD).*** Another form of dependency between tests could be through third-party libraries such as Mockito. Developers often mock methods rather than running them; e.g., they use Mockito to define what a specific method should return for a given input. After test execution, mocks can be queried to see what methods were called and how many times. Not resetting the state of Mockito can create an implicit dependency between tests that use Mockito. The issue 182 in project alien4cloud [31] shows an example TPFD flaky test. The snippet below shows how Croissant generates such tests[7].

```
public void pTest() { // Polluter
    Mockito.when(mockedClass.stub(input1).thenReturn(output1);
    Mockito.when(mockedClass.stub(input2).thenReturn(output2);
}
public void cTest() { // Cleaner
    Mockito.reset(mockedClass);
}
public void vTest() { // Victim
    Mockito.when(mockedClass.stub(input).thenReturn(output);
    verify(mockObject, times(1)).add());
    // If polluter executes first, verify(..., times(3)) is true
}
```

3.2.4 ***Cached Status Dependency (CSD).*** Dependencies such as shared cache lead to flakiness if a polluter modifies the cache used by a victim. Issue 1165 in project spring-ws [58] is a real-world of such flakiness. The snippet below shows an example CSD mutant in Croissant, where the victim `vTest` assumes a clean cache status and fails if the polluter `pTest` runs first unless the cleaner `cTest` runs before the victim to clean the shared cache. Croissant uses Caffeine to create cache objects [4].

```
public void pTest() { // Polluter
    cache.add(cachedObject);
}
public void cTest() { // Cleaner
    cache.invalidateAll();
}
public void vTest() { // Victim
    assertNull(cache.getIfPresent(cachedObject));
}
```

[7]The current implementation of Croissant only supports Mockito. Users can extend the template to support other mocking libraries.

3.2.5 ***Database State Dependency (DSD).*** Relying on other tests to populate a shared database or failing to reset the state of the database in `tearDown` results in DSD flaky tests. A real-world example of such a test is reported in the issue 8 in project spring-data-ebean [33]. The snippet below shows an example DSD mutation operator in Croissant. The shared `item` between these tests is not in the database unless state-setter `sTest` inserts it. If the brittle `bTest` runs in isolation, it fails; if the state-setter runs before the brittle, it passes. However, if the state-unsetter `utest` runs before brittle, the shared item is deleted, which makes the brittle fail again.

```
public void bTest() { // Brittle
  PreparedStatement query = "SELECT * FROM table WHERE ID=item.ID";
  assertTrue(query.execute());
}
public void sTest() { // State-Setter
  PreparedStatement insert = "INSERT INTO table item";
  insert.execute();
}
public void uTest() { // State-Unsetter
  PreparedStatement delete = "DELETE FROM table WHERE ID=item.ID";
  delete.execute();
}
```

3.2.6 ***File Permission Dependency (FPD).*** For FPD, the shared state between tests is a file. With such a dependency, the polluter may modify the permission of a file the victim later attempts to access. Consequently, access to the file may fail. The issue 484 in project Wikidata-Toolkit is a real-world example [32], where the victim `testMwRecentCurrentDumpFileProcessing` fails since the polluter changes the permission to the file `wdtk-dumpfiles` to read-only. The snippet below shows the template FPD mutant, where the polluter `pTest` changes the access permission of a shared file to non-writable, which leads the victim `vTest` to fail.

```
public void pTest() { // Polluter
    file.setWritable(false);
}
public void cTest() { // Cleaner
    file.setWritable(true);
}
public void vTest() { // Victim
    file.write();
}
```

3.2.7 ***Resource Availability (RA).*** A unit test that assumes the existence of a certain resource, e.g., a file, can become a brittle test. It can fail if run in isolation but pass if run after another test that creates the required resource. The snippet below shows the template RA mutant in Croissant. At the start, the filesystem is cleared before all tests run, so `FileA` does not exist. If brittle `bTest` runs first, it fails unless state-setter `sTest` runs before the brittle. However, if the state-unsetter `uTest` runs before the brittle, the file resource is unavailable, which causes the brittle to fail again.

```
public void bTest() { // Brittle
    assertTrue(File.exist(FileA)); // FileA does not exist at first
}
public void sTest() { // State-Setter
    File.create(FileA);
}
public void uTest() { // State-Unsetter
    File.delete(FileA);
}
```

3.3 NOD Mutation Operators

Our empirical study identified seven classes of NOD flaky tests depending on the type of APIs they misuse. We next explain some flaky tests and their corresponding mutation operators.

3.3.1 ***Memory Dependency (MD).*** MD flaky tests require a specific amount of memory that may or may not be available, e.g., due to the frequency of garbage collection and memory usage of other tests. Consequently, the test passes if enough memory is available and fails otherwise.

```
public void MDTest() { // NOD test
    System.gc();
    long totalMemory = getTotalMemory();
    System.gc();
    long usedMemory = getTotalMemory() - getFreeMemory();
    assertEquals(totalMemory, usedMemory);
}
```

3.3.2 ***Platform Dependency (PD).*** Platform-dependent flakiness occurs when a test assumes certain properties about the running platform, including the availability of local ports. Consequently, if it runs on a different platform that does not satisfy the assumption, e.g., during continuous integration, test outcome can differ. The code snippet below shows the template PD mutant in Croissant, which makes a test rely on the availability of a specific port. The test passes if the port is available but fails otherwise.

```
public void PDTest() { // NOD test
    int PORT = 6380; // hard-coded port
    assertTrue(isPortAccessible(PORT));
}
```

3.3.3 ***System Time Dependency (STD).*** The outcome of STD flaky tests depends on the timezone of the execution environment or a specific timestamp. Thus, running the test at a different time, e.g., with daylight saving, can trigger flakiness. We implement two types of STD flaky tests, namely STD-Z and STD-V. The flakiness in the former is due to the time-zone difference, while the time values in the latter result in the flakiness.

1) Time-zone-dependency (STD-Z) flaky tests. The snippet below shows how Croissant injects STD-Z flakiness, where a test assumes one specific timezone, but the actual timezone differs.

```
public void STDZTest() { // NOD test
    SimpleDateFormat dateFormat;
    dateFormat.setTimeZone(TimeZone.getTimeZone("UTC"));
    int date1 = dateFormat.parse("2022-11-22 10:13:55");
    dateFormat.setTimeZone(TimeZone.getDefault());
    int date2 = dateFormat.parse("2022-11-22 10:13:55");
    assertEquals(date1, date2);
}
```

2) Timestamp-Value-dependency (STD-V) flaky tests. Such flakiness is caused when a test assertion depends on the system timestamp. An example real-world issue is 1935 in project hutool [25], where an assertion compares the current time with a precision of a second.

```
public void STDVTest() { // NOD test
    long timeStamp1 = System.currentTimeMillis();
    Thread.sleep(1L);
    long timeStamp2 = System.currentTimeMillis();
    assertEquals(timeStamp1, timeStamp2);
}
```

When a time object is created, the current timestamp is saved. If there is a delay between the creation time and the assert time, the test fails. The code snippet below shows an example of STD-V mutant generated by Croissant.

3.3.4 ***Concurrency Timeout Deadlock (CTD).*** Another root cause of test flakiness is timeouts, which happen non-deterministically due to concurrency. For example, a test may make some calls and wait for some time to get the return value. If the wait time is not long enough to ensure the return, different executions of the test may show flaky behavior. Also, threads may get stuck in a deadlock in some executions while returning successfully in others, resulting in test flakiness.

```
@Test(timeout=5)
public void CTDTest() { // NOD test
    Object lock1 = new Object();
    Object lock2 = new Object();
    Thread thread1 = new Thread() {
        public void run() {
            synchronized (lock1) {
                Thread.sleep(100);
                synchronized (lock2) {}
            }
        }
    };
    Thread thread2 = new Thread() {
        public void run() {
            synchronized (lock2) {
                Thread.sleep(100);
                synchronized (lock1) {}
            }
        }
    };
    thread1.run();
    thread2.run();
}
```

3.3.5 ***Asynchronous Wait (AW).*** When a test makes an asynchronous call and does not wait properly to get the returned result, it leads to test flakiness. A real-world example is issue 3683 in project retrofit [54]. The snippet below shows how Croissant injects AW flaky tests, with `CountDownLatch` to mimic multi-threaded effects.

```
public void AWTest() { // NOD test
    CountDownLatch latch = new CountDownLatch(1);
    Thread thread = new CountDownThread(latch);
    thread.start();
    assertTrue(latch.await(1000, TimeUnit.MILLISECONDS));
}
```

3.3.6 ***Too Restrictive Range (TRR).*** If the test design does not consider some valid output values, actual outputs can be outside the assertion range. A restrictive range for test assertions causes such flakiness. Croissant injects TRR flakiness by mimicking an extremely restricted range, similar to the code snippet below.

```
public void TRRTest() { // NOD test
    boolean output = true;
    int restrictRange = getAcceptableRange();
    if (restrictRange >= 0 || restrictRange <= 0) {
        output = false;
    }
    assertTrue(output);
}
```

3.3.7 ***Race Condition (RC).*** Multi-threading may cause test flakiness due to the seemingly non-deterministic behavior of thread interleaving. For example, the test outcome may depend on a variable shared by multiple threads in a non-thread-safe manner, where race conditions result in non-determinism.

```
public void RCTest() throws IOException { // NOD test
    ArrayList<Integer> list = new ArrayList<>();
    list.add(0);
    for (int i = 0; i < 1000; i++) {
        Thread thread = new Thread(new NonSafeThread(list));
        thread.start();
    }
    assertEqual(1000, list.get(0));
}
class NonSafeThread implements Runnable {
    ArrayList<Integer> list;
    NonSafeThread(ArrayList<Integer> var2) {
        this.list = var2;
    }
    list.set(0, list.get(0) + 1);
}
```

3.4 ID Mutation Operators

Implementation-dependent (ID) flaky tests are a subcategory of NOD flaky tests in which the flakiness is due to an incorrect assumption on some implementation-specific API, e.g., the order of *unordered collections* [23]. For example, `Map` and `Set` collections in Java do not provide any order for iteration. We separated ID from NOD, as some test-flakiness detection and prevention techniques may focus on IDs [23, 49] but not general NODs. The remainder of the section discusses mutation operators that mimic different misuses of unordered collections.

3.4.1 ***Unordered Collection Index Access (UCIA).*** Java does not allow direct indexing of unordered collections. Some tests bypass this constraint by converting an unordered collection into an ordered collection. However, this conversion does not preserve the index of items deterministically, which can lead to test flakiness. The issue 4717 in project druid [14] shows a real-world example. Croissant generates UCIA mutants by converting one unordered collection into two lists and checking if the items at index 0 are the same, as shown in the code snippet below.

```
public void UCIATest() { // ID test
    HashMap<> map = getAMap();
    List<> list1 = new ArrayList<>(map.values());
    List<> list2 = new ArrayList<>(map.values());
    assertEquals(list1.get(0), list2.get(0));
}
```

3.4.2 ***Unordered Collections Conversion (UCC).*** UCC flaky tests happen when developers convert unordered collections such as `Map` to `String` to use for comparison in assertions. Due to the non-deterministic order of elements in unordered collections, the actual string generated may or may not match the expected string.

```
public void UCCTest() { // ID test
    HashMap<> map = getAMap();
    Set<> set = new HashSet<>(map.values());
    assertEquals(set.toString(), set.toString());
}
```

Croissant generates UCC mutants by converting an unordered collection into a string twice and checking if the two strings match.

3.4.3 ***Reflection API Misuse (RAM).*** Developers use reflection APIs to inspect methods or classes at runtime, but the results of such APIs are non-deterministic. As a result, assuming a specific order is incorrect. The issue 480 in project commons-lang [39] demonstrates a real-world RAM flaky test, which assumes a specific order of elements returned by the reflection method `getDeclaredFields`. The snippet below shows Croissant's technique for generating RAM flaky tests. The test uses `getMethods` from the same class twice and checks if the first returned elements are the same.

```
public void RAMTest() { // ID test
    Method[] methods1 = classA.getMethods();
    Method[] methods2 = classA.getMethods();
    assertEquals(methods1[0], methods2[0]);
}
```

4 EVALUATION

To evaluate the effectiveness of Croissant, we investigate the following research questions:

RQ1: Quality of the Mutants. To what extent the designed mutation operators are successful in making test suites flaky? What is the percentage of mutants that are non-compilable (test or whole test-suite cannot be compiled) or non-executable (test(s) permanently fails due to exception caused by mutation)?

RQ2: Effectiveness of the Mutation. To what extent do the mutants challenge the state-of-the-art test-flakiness detection tools?

RQ3: Comparison with Other Techniques. How prevalent are our mutants compared to the alternative mutation testing approach, FlakiMe [8]? To what extent can FlakiMe challenge test-flakiness detection tools compared to Croissant?

RQ4: Performance. What are the performance characteristics of the proposed technique?

4.1 Experimental Setup and Data Availability

Test-Flakiness Detection Tools. To analyze Croissant's mutants, we used state-of-the-practice Maven Surefire [40] and two state-of-the-research tools for Java: NonDex [23] and iDFlakies [36]. Surefire is the default Maven plugin for running unit tests. One feature it offers is re-running *failed* tests when `rerunFailingTestsCount` parameter is set to a value greater than 0. If the test outcome differs in multiple executions, Surefire raises *flaky failure* or *flaky error*. As a result, Surefire can only detect NOD flaky tests. NonDex is a tool for detecting ID and NOD flaky tests by randomly exploring different behaviors of certain APIs during test execution. iDFlakies detects OD flaky tests by reordering and rerunning tests in the test suite; it can also detect NOD tests as a byproduct of rerunning tests. We were not able to use learning-based techniques such as FlakeFlagger [2], FlakyVocabulary [53], and Flakify [19], due to the limitations of these techniques. Specifically, these tools can only extract features from the source code, while Croissant generates binary files for mutants[8]. We choose bytecode manipulation in the current implementation of Croissant, because it is faster and does not require expensive compilation to generate executable mutants, which can reduce the cost of mutation testing.

Subjects. To collect subjects, we searched GitHub for Java repositories with the following properties: (1) popular and well-maintained

[8]Decompilation of the binary files resulted in non-compilable test suites.

Table 2: The breakdown of the number of mutants generated by Croissant for each subject test suite.

Project	Total	IVD	SVD	TPFD	CSD	DSD	FPD	RA	MD	PD	STD-V	STD-Z	AW	CTD	RC	TRR	UCIA	UCC	RAM
cli	1003	20	20	28	28	28	28	28	378	28	23	28	51	28	28	175	28	28	28
codec	2532	50	50	59	65	65	59	65	1088	57	57	62	114	59	59	452	57	57	57
crypto	530	19	19	19	19	19	19	19	136	17	17	29	34	17	17	79	17	17	17
csv	1058	22	22	25	26	26	25	26	440	25	25	26	50	25	25	195	25	25	25
email	601	13	13	18	18	14	18	18	218	15	15	14	15	29	14	125	15	15	15
fileupload	410	13	13	13	13	13	13	13	127	13	13	14	26	13	13	61	13	13	13
graph	700	21	21	23	23	23	23	23	228	21	21	21	42	21	21	105	21	21	21
jsoup	1765	30	30	32	32	30	32	32	816	31	31	32	62	32	32	418	31	31	31
marine	3650	79	79	79	79	78	79	79	1372	77	73	79	150	77	77	962	77	77	77
math	1219	33	33	33	33	33	33	33	406	32	32	32	64	31	31	264	32	32	32
monitoring	552	16	16	16	16	16	16	16	203	17	17	34	17	34	17	67	17	17	17
scxml	1688	52	52	52	52	52	52	52	499	52	52	52	107	52	52	302	52	52	52
text	3556	82	82	82	86	86	80	86	1499	80	80	84	160	77	77	675	80	80	80
unix4j	716	48	48	51	51	45	48	44	180	14	14	14	28	14	14	61	14	14	14
xmlgraphics	1835	61	60	61	61	61	61	61	502	61	61	62	123	61	61	295	61	61	61
Total	21815	559	558	591	602	589	586	595	8092	540	567	530	1074	538	538	4236	540	540	540

Table 3: The number of flaky tests for each subject test suite using FlakiMe and Croissant (NE = Non-Executable).

Project	#tests	#FlakiMe mutants	#Croissant mutants (NE)
cli [6]	438	324	1003 (0)
codec [7]	1336	865	2532 (0)
crypto [9]	121	92	530 (0)
csv [10]	326	435	1058 (0)
email [18]	139	190	601 (0)
fileupload [20]	78	83	410 (0)
graph [21]	131	138	727 (27)
jsoup [28]	1136	1035	1765 (0)
marine [43]	995	1097	3697 (47)
math [46]	375	328	1267 (48)
monitoring [48]	118	108	552 (0)
scxml [56]	239	261	1688 (0)
text [59]	1153	1182	3556 (0)
unix4j [60]	136	453	716 (0)
xmlgraphics [69]	196	205	1835 (0)
Total	6302	6796	21937 (122)

(> 200 stars, recent commits within six months, > 100 closed issues, and < 20 open issues tagged as bugs); (2) has > 100 existing JUnit tests; (3) has size > 5 KLoC; (4) is written in Java 8 (requirement of iDFlakies); and (5) with test suites using only JUnit 4 or JUnit 5, and not the mix (requirement of iDFlakies). From these repositories, we excluded those that we could not compile and those for which FlakiMe could not generate mutants. To eliminate the impact of original flakiness on the evaluation results, we used Surefire and NonDex first to ensure the initial test suites did not have obvious NOD and ID flakiness. For the cases that passed the first step, we ran iDFlakies to check for obvious OD flaky tests. Table 3 (first column) shows our 15 subjects.

Mutant Generation. We use Croissant for the generation of flaky mutants. The implementation of Croissant is publicly available [64] as a stand-alone tool that takes the source code of test suites as an input and generates a mutated test suite bytecode as an output. To pinpoint tests of interest in test suites, i.e., those that should be modified or can act as a victim/brittle in the test suites, Croissant employs lightweight flow-sensitive analysis using Soot [61]. The current implementation supports both first- and higher-order mutation injection. However, all the experiments have been performed on first-order mutation injection. We also compare Croissant with FlakiMe [8], which injects exception statements that can be executed with some probabilities, making the test suites flaky. FlakiMe can generate more mutants than the number of tests in the test suite because it mutates tests as well as methods such as `setUp` in test classes.

4.2 RQ1: Quality of the Mutants

To measure the quality of Croissant's mutation operators, we check the extent to which it creates executable flaky test suites. Table 3 shows the summary results of applying Croissant to subject test suites. In total, Croissant generated 21,937 mutants for all the subjects, of which only 122 (< 0.5%) were non-executable (last column in Table 3). To validate that the generated mutants induce flakiness, we executed mutant helpers (§3.1.3) and the original test suites. If the pass/fail outcome of these two test suites differed, we confirmed that flakiness injection was successful. This validation process confirmed that all the executable mutants (> 99.5% of the generated mutants) made the original test suite flaky. Due to the design of our mutation operators and ensuring that mutants represent flaky behavior, Croissant produces no equivalent mutant. Similarly, given the uniqueness of flaky anti-patterns concerning specific tests in the test suite, Croissant will never generate duplicate mutants. These results confirm that Croissant is a viable technique for injecting various flakiness types into test suites.

4.3 RQ2: Effectiveness of the Mutation

To evaluate the effectiveness of Croissant in challenging flakiness-detection tools, we applied Surefire and NonDex on NOD and ID mutants, and iDFlakies on OD mutants. We configure Surefire with *rerunFailingTestsCount=5*, NonDex with *nondexRuns=5*, and iDFlakies with *rounds=5* repeating 5 runs with different random seeds.

NOD mutants. We introduce thresholds for NOD mutants to control the possibility of test failures. Higher threshold leads to a higher possibility of test failures. We vary the threshold tr from 0.1 to 1. When $tr = 0$, tests always pass. As tr increases, tests have more chance to fail (but can still pass during some of the 5 reruns), so more tests behave as flakes. After achieving the highest point of flakiness, tests have a much higher possibility to fail

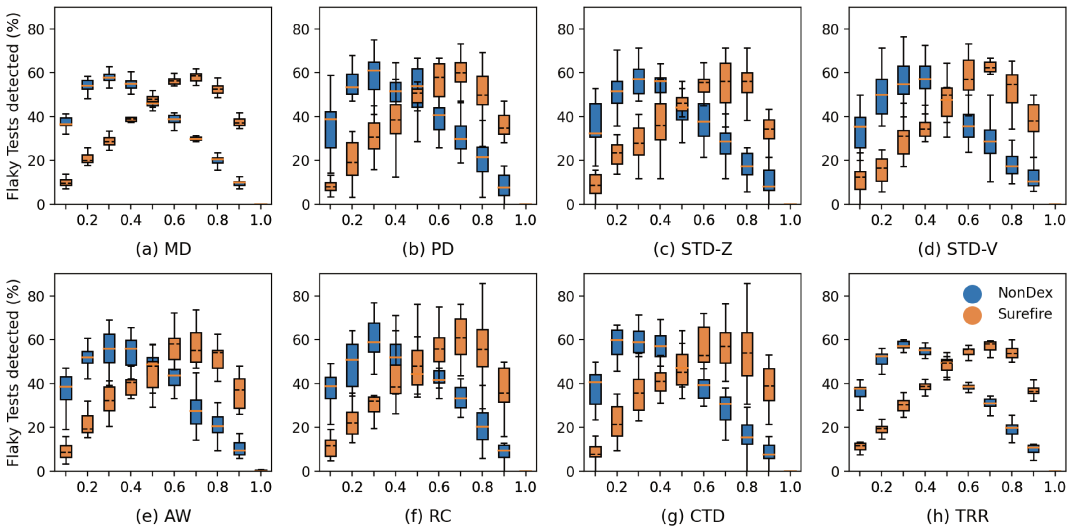

Figure 1: Effectiveness of NOD mutants in challenging NonDex and Surefire running with thresholds from 0.1 to 1 with the increment value 0.1 (tr introduced in Section 4.3)

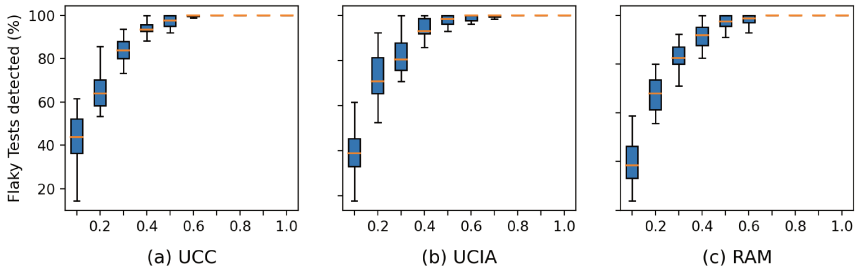

Figure 2: Effectiveness of ID mutants in challenging NonDex running with thresholds from 0.1 to 1, with the increment value 0.1 (tr introduced in Section 4.3)

during all reruns, so more and more mutants are categorized as `Failure` or `Error`. For $tr = 1$, tests always fail, resulting in no flakiness. Figure 1 shows the percentages of flaky tests detected by Surefire and NonDex as the threshold tr changes. Each box plot represents values for each of 15 projects, where each value is a percentage of all generated flaky tests detected by the used tool. The trends of both Surefire and NonDex detection first increase and then decrease, which is expected. However, we observe the turning points of Surefire and NonDex detection to greatly differ. Surefire achieves the highest percentage of flaky tests when $tr \in [0.6, 0.8]$, while NonDex achieves the highest percentage when $tr \in [0.2, 0.4]$. When $tr < 0.5$, NonDex works more effectively than Surefire. We further calculated flakiness detection formulas $tr(1 - tr^5)$ for Surefire and $(1-tr)(1-(1-tr)^5)$ for NonDex. The trends in Figure 1 are consistent with the probabilities calculated by formulas. In practice, we can expect that flaky tests do not fail too often (with tr most likely less than 0.5); otherwise, tests that fail too often would lead to broken builds and likely be addressed by developers.

ID mutants. NonDex is designed to detect wrong assumptions on Java APIs with implementation-specific behavior, targeting flaky ID tests. It specifically checks for the invocation of such APIs in tests and examines different behaviors, e.g., different order of unordered data structures to ensure there is no wrong assumption. With the threshold tr increasing, the ID mutants have a higher chance of executing flakiness, thereby being detected by NonDex. Figure 2 shows the percentages of flaky tests detected by NonDex and Surefire when they are run on ID mutants. With tr increasing, NonDex can detect more flaky tests and, in all cases, achieves 100% when $tr \in [0.6, 1.0]$. (Note that the trends differ for NOD and ID flaky tests.) However, Surefire cannot detect any of the ID flaky tests when tr varies from 0 to 1.

OD mutants. Test suites with more cleaners/state-unsetters should make it harder for iDFlakies to detect victims/brittles. To confirm this behavior, we varied the number of cleaners/state-setters in the test suite from 0 to 50. Figure 3 demonstrates the result of this experiment. From these results, iDFlakies can always detect nearly 100% victims/brittles when the number of cleaners/state-unsetters is sent to 0. By increasing this number, the ability of iDFlakies to detect flaky tests degrades, such that with 50 cleaners/state-unsetters, it can only detect about 10% of the flaky tests.

The results on the effectiveness of Croissant in challenging different tools show similar trends, regardless of the type of flakiness. This is mainly because the current dynamic flakiness-detection tools need to observe the flaky behavior first and then confirm the flakiness. As researchers develop smarter tools for flakiness detection that involve deeper analysis of the test suites, they can

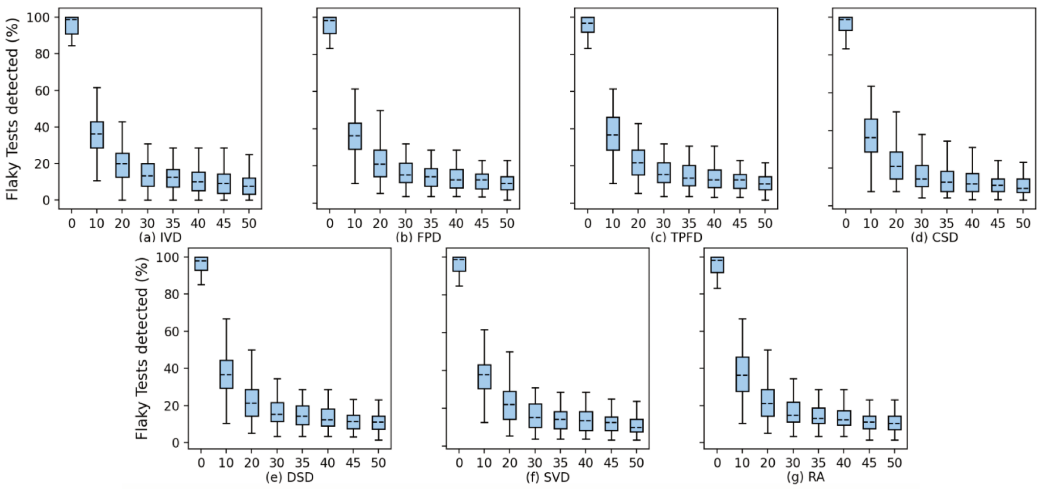

Figure 3: Effectiveness of OD mutants in challenging iDFlakies running with the number of cleaners from 0 to 50

choose which mutation operators to use, showing the superiority of their tool, regardless of the probability of observing flakiness.

Mutation operators that change file systems (RA and FPD), memory (MD), and database (DSD) are specifically important since these resources are always used in real-world development. These operators also helped us find diverse bugs in iDFlakies. Specifically, in the initial phase of our experiments, we observed that increasing the number of cleaners in these mutants does not consistently degrade the detection of victims. Our further investigation revealed two non-deterministic bugs in iDFlakies: (1) iDFlakies collects test methods by name, and not by signature. In the presence of helper methods with the same name as tests, iDFlakies ignores the test if it collects the helper method first. Consequently, the list of methods does not always include all tests consistently, resulting in different results regardless of the change in the number of cleaners and state-unsetters; (2) iDFlakies uses an unordered collection when collecting the test orders in the original test suite, resulting in non-determinism, resulting a mismatch when comparing the original and shuffled order. Our fixes to these issues [1], which were accepted by the developers of iDFlakies, resolved the inconsistency in the reported results.

4.4 RQ3: Comparison with Other Techniques

In this RQ, we evaluate (1) how the mutants generated by Croissant differ from those of prior techniques and (2) to what extent such techniques can challenge test-flakiness detection tools. To that end, we qualitatively and quantitatively compare the mutants generated by Croissant and FlakiMe for three aspects listed below.

Flaky test type. Compared to Croissant that generates all types of flakiness—OD, NOD, and ID—FlakiMe generates only NOD flaky tests. Consequently, one cannot use FlakiMe to evaluate flakiness detection and mitigation tools designed for ID and OD flaky tests. Furthermore, we investigate the effectiveness of FlakiMe NOD mutants in challenging Surefire and NonDex. As demonstrated by Figure 4, FlakiMe's mutants are similar to Croissant's mutants in showing that re-execution of tests poses challenges to detecting infrequent NOD flaky tests. The detection rate follows a similar

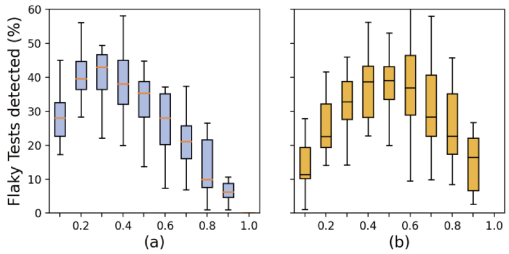

Figure 4: Effectiveness of the FlakiMe mutants in challenging (a) NonDex and b) Surefire with flake rate from 0.1 to 1

trend. However, for the same *threshold* and *flake rate* value, the average detection rate of FlakiMe's mutants is *lower* than Croissant's. The reason is that FlakiMe computes the *flake rate*—probability to observe flaky behavior—as $FR_n \times P_{flakiness}$, where $P_{flakiness}$ is a predefined value computed by a prediction model, and FR_n is similar to the threshold value tr in Croissant. Hence, *flake rate* value in FlakiMe is always smaller than the *threshold* value in Croissant, resulting in a lower chance to manifest flakiness. Furthermore, computing the exact probability of failure in FlakiMe mutants is challenging. In Croissant, we can control the probability to trigger flakiness through tr and k values. In contrast, for FlakiMe, this depends on the number of times a test reaches the threshold checks. FlakiMe adds such checks in various methods in test classes, and these methods may be executed more than once (e.g., a test method may call setUp and tearDown methods).

Flakiness signature. Croissant can inject 18 unique patterns of flaky bugs into test suites. These mutants are designed based on an extensive study of the issues and commits of open-source Java projects; thereby, their footprints are *more similar* to real flaky tests than FlakiMe's mutants, which are created by adding only a conditional exception statement, always resulting in exception failure as a footprint. As a result, while FlakiMe mutants represent *flaky test outcome*, they do not represent *flaky test behavior*. This difference is specifically important for benchmarking because different flakiness detection and mitigation techniques may focus on the particular properties of flaky tests. For example, another way to detect RC

flaky tests could involve a data-flow analysis, while a lightweight static analysis could detect potential FPD flaky tests. Without a fine-grained benchmark similar to Croissant, one cannot truly distinguish the extent to which these sophisticated techniques work better than rerun techniques such as Surefire and iDFlakies.

Mutation location. Croissant uses a lightweight static analysis to identify the potential location of flakiness injection, depending on specific flaky anti-pattern. On the other hand, FlakiMe relies on a vocabulary-based machine-learning model to identify the injection location. Specifically, FlakiMe mutates a test in the subject test suite if there is a high similarity between its vocabulary with previously identified flaky tests in real-world settings. As a result of this difference, Croissant can determine all the applicable locations for a specific class of mutation operators in a given test suite, generating a more diverse set of flaky tests as a benchmark. While a bigger set of mutants adds to the cost of mutation testing, it can create a more representative set of mutants.

4.5 RQ4: Performance for Mutant Generation

To answer this research question, we evaluated the time required for Croissant to mutate subject test suites. We ran the experiments on a CPU cluster with 2.6 GHz Intel Xeon(R) Platinum 8171M processor and 16 GB DDR3 RAM. Overall, it takes 0.35 seconds on average (min=0.19, max=0.57) to mutate a test suite with a single mutation operator and 8 minutes on average (min=1, max=21) to create all the mutants for subject test suites. The time required for assessing test-flakiness detection tools is unique to their underlying approach and is independent of Croissant's design choices.

5 RELATED WORK

Our research is related to prior work on mutation testing as well as approaches aimed to characterize, detect, or mitigate flaky tests.

Mutation Testing. Mutation testing has been widely used in testing programs written in different languages [11, 12, 12, 13, 42, 47], as well as testing program properties such as specifications [44], memory usage [68], and energy consumption [27]. The most closely related work to Croissant on injecting flakiness into test suites are FlakiMe [8] and Flaker [24].

FlakiMe creates NOD flaky tests by adding exception statements into the tests. The probability of observing flaky behavior depends on the combination of *test flakiness probability*, which is provided by a prediction model integrated into FlakiMe, and *nominal flake rate*, which is a parameter that users can adjust. Similar to FlakiMe, Croissant's flakiness rate can be determined by users (threshold value) to control the experiments and investigate the effectiveness of test-flakiness detection tools. However, Croissant's mutation operators are designed based on real-world flaky tests. Consequently, the footprint of our mutants is more similar to actual flaky tests compared to FlakiMe's mutants. This property is important for benchmarking flaky tests, which can have usages beyond evaluating test-flakiness detection tools discussed in this paper.

Flaker injects OD flaky tests into test suites by removing the helper statements in cleaners and state-setter tests. Croissant's OD mutation operators are superior to Flaker's in two ways. First, the flakiness behavior of our mutation operators can be adjusted by changing the number of cleaners and state-setters. Second, removing helper statements, i.e., voiding the impact of cleaners and state-setters, makes detecting OD flaky tests easier because the only remaining dependency is now between polluter/victim or polluter/brittle tests, and flakiness behavior can be observed in more test-suite permutations.

Test Flakiness. Test flakiness could be frustrating in software development [51]. To help developers, several techniques have been proposed to characterize [15, 16, 34, 35, 37, 38, 41], detect [3, 30, 36, 45, 52, 53, 62, 67, 70], or mitigate [57, 63, 66] flaky tests. The related work characterizing flaky tests has investigated and analyzed flaky tests in traditional programs, probabilistic programming systems, machine learning projects, and mobile apps. These techniques aim to understand the nature of flaky tests and what makes their detection of them challenging. Consequently, they characterize flaky tests at a very high level, e.g., OD, NOD, or ID. In contrast, Croissant categorizes flaky tests based on the unique flaky anti-patterns. Such fine-grained categorization can better describe flaky tests and distinguish test-flakiness detection tools in detecting specific classes of tests.

Croissant is orthogonal to test-flakiness detection and mitigation body of work. That is, Croissant can generate the evaluation benchmark to fairly compare such techniques with prior work and identify the strengths and weaknesses of newer techniques. Furthermore, the capability of injecting Croissant's mutants into given JUnit test suites also enables large-scale evaluation for such tools. Compared to IDoFT [26], which is a distributed/community effort to construct a dataset of real-world flaky tests, Croissant can inject flaky tests into any given test suite in JUnit 4 or JUnit 5.

6 CONCLUDING REMARKS

Test flakiness can drastically impact the effectiveness of regression testing and the quality of the software products. Despite recent advancements in the detection and mitigation of test flakiness, the research is still in its infancy. Specifically, the community lacks systematic approaches or benchmarks for fairly comparing such techniques. We proposed Croissant, a framework to inject realistic flaky tests into test suites. The current version of Croissant offers 18 high-quality mutation operators. The experimental results show that these mutation operators can indeed challenge test-flakiness detection tools to further design better heuristics and algorithms for reliably pinpointing flaky tests.

We believe that Croissant offers several directions for future work. While the focus of this paper is using flaky anti-patterns for assessing test-flakiness detection tools, one can use the defect model and anti-patterns for two other purposes: (1) developing a static analysis tool to find these anti-patterns in test suites and (2) curating a realistic, high-quality dataset for training machine learning models to detect flaky tests without the need for re-execution. To that end, one direction for future research can be an empirical study that measures the correlation between real flaky tests and Croissant mutants, and compare how different or similar their footprints are.

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CCF-1763788, CCF-1956374, and CCF-2238045, and grants from IBM and C3.ai. We also acknowledge support for research on flaky tests from Google and Meta.

REFERENCES

[1] [n. d.]. Issue 51 iDFlakies. https://github.com/idflakies/iDFlakies/issues/51.
[2] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. [n. d.]. FlakeFlagger: Predicting flakiness without rerunning tests. https://doi.org/10.1109/ICSE43902.2021.00140
[3] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 433–444.
[4] Caffeine 2023. Caffeine caching library. https://github.com/ben-manes/caffeine.
[5] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. 2017. Test suite parallelization in open-source projects: A study on its usage and impact. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 838–848.
[6] cli 2023. cli Project. https://github.com/apache/commons-cli
[7] codec 2023. https://github.com/apache/commons-codec
[8] Maxime Cordy, Renaud Rwemalika, Adriano Franci, Mike Papadakis, and Mark Harman. 2022. FlakiMe: Laboratory-controlled test flakiness impact assessment. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 982–994.
[9] crypto 2023. crypto Project. https://github.com/apache/commons-crypto
[10] csv 2023. csv Project. https://github.com/apache/commons-csv
[11] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. 2001. Interface mutation: An approach for integration testing. *IEEE transactions on software engineering* 27, 3 (2001), 228–247.
[12] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of Android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–10.
[13] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2017. Mutation operators for testing Android apps. *Information and Software Technology* 81 (2017), 154–168.
[14] druid4717 [n. d.]. Issue 4717 druid. https://github.com/alibaba/druid/pull/4717.
[15] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. 2020. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 211–224.
[16] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: The developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 830–840.
[17] elasticjob592 [n. d.]. Issue 592 elasticjob. https://github.com/apache/shardingsphere-elasticjob/pull/592.
[18] email 2023. email Project. https://github.com/apache/commons-email
[19] Sakina Fatima, Taher A Ghaleb, and Lionel Briand. 2022. Flakify: A black-box, language model-based predictor for flaky tests. *IEEE Transactions on Software Engineering* (2022), 1912–1927.
[20] fileupload 2023. fileupload Project. https://github.com/apache/commons-fileupload
[21] graph 2023. graph Project. https://github.com/apache/commons-graph
[22] Martin Gruber and Gordon Fraser. 2023. FlaPy: Mining Flaky Python Tests at Scale. *arXiv preprint arXiv:2305.04793* (2023).
[23] Alex Gyori, Ben Lambeth, August Shi, Owolabi Legunsen, and Darko Marinov. 2016. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 993–997.
[24] Sarra Habchi, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2021. On the Use of Mutation in Injecting Test Order-Dependency. *arXiv preprint arXiv:2104.07441* (2021).
[25] hutool1935 [n. d.]. Issue 1935 hutool. https://github.com/dromara/hutool/pull/1935.
[26] idoft 2023. International Dataset of Flaky Tests (IDoFT). https://github.com/TestingResearchIllinois/idoft.
[27] Reyhaneh Jabbarvand and Sam Malek. 2017. μdroid: An energy-aware mutation testing framework for Android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 208–219.
[28] jsoup 2023. jsoup Project. https://github.com/jhy/jsoup
[29] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.
[30] Tariq M King, Dionny Santiago, Justin Phillips, and Peter J Clarke. 2018. Towards a Bayesian network model for predicting flaky automated tests. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 100–107.
[31] l1 [n. d.]. Issue 182 alien4cloud. https://github.com/alien4cloud/alien4cloud/pull/182.
[32] l2 [n. d.]. Issue 484 Wikidata-Toolkit. https://github.com/Wikidata/Wikidata-Toolkit/pull/484/files.
[33] l3 [n. d.]. Issue 8 spring-data-ebean. https://github.com/hexagonframework/spring-data-ebean/pull/8.
[34] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–111.
[35] Wing Lam, Kıvanç Muşlu, Hitesh Sajnani, and Suresh Thummalapenta. 2020. A study on the lifecycle of flaky tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1471–1482.
[36] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 312–322.
[37] Wing Lam, Stefan Winter, Angello Astorga, Victoria Stodden, and Darko Marinov. 2020. Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 403–413.
[38] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. 2020. A large-scale longitudinal study of flaky tests. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–29.
[39] lang480 [n. d.]. Issue 480 commons-lang. https://github.com/apache/commons-lang/pull/480.
[40] Qingzhou Luo. 2014. https://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html
[41] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653.
[42] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.
[43] marine 2023. marine Project. https://github.com/ktuukkan/marine-api
[44] Evan Martin and Tao Xie. 2007. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*. 667–676.
[45] Maximiliano A Mascheroni and Emanuel Irrazabal. 2018. Identifying key success factors in stopping flaky tests in automated REST service testing. *Journal of Computer Science and Technology* 18, 02 (2018), e16–e16.
[46] math 2023. math Project. https://github.com/apache/commons-math
[47] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. Efficient JavaScript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 74–83.
[48] monitoring [n. d.]. monitoring Project. https://github.com/google/java-monitoring-client-library
[49] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael D. Ernst. 2021. Verifying Determinism in Sequential Programs. In *ICSE*. 37–49.
[50] nacos [n. d.]. Issue 4384 nacos. https://github.com/alibaba/nacos/pull/4384.
[51] Owain Parry, Gregory M Kapfhammer, Michael Hilton, and Phil McMinn. 2022. Surveying the developer experience of flaky tests. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 253–262.
[52] Suzette Person and Sebastian Elbaum. 2015. Test analysis: Searching for faults in tests (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 149–154.
[53] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. 2020. What is the vocabulary of flaky tests?. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 492–502.
[54] retrofit3683 [n. d.]. Issue 3683 retrofit. https://github.com/square/retrofit/pull/3683.
[55] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.
[56] scxml [n. d.]. scxml Project. https://github.com/apache/commons-scxml
[57] August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. 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*. 545–555.
[58] spring [n. d.]. Issue 1165 spring-ws. https://github.com/spring-projects/spring-ws/pull/1165.
[59] text 2023. text Project. https://github.com/apache/commons-text
[60] unix4j 2023. unix4j Project. https://github.com/tools4j/unix4j
[61] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[62] Roberto Verdecchia, Emilio Cruciani, Breno Miranda, and Antonia Bertolino. 2021. Know you neighbor: Fast static prediction of test flakiness. *IEEE Access* 9

(2021), 76119–76134.
[63] Ruixin Wang, Yang Chen, and Wing Lam. 2022. iPFlakies: A Framework for Detecting and Fixing Python Order-Dependent Flaky Tests. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 120–124.
[64] website [n. d.]. CROISSANT website. https://github.com/Intelligent-CAT-Lab/Croissant
[65] website-mutants [n. d.]. List of proposed mutation operators. https://sites.google.com/view/croissant-website/mutation-operators
[66] Anjiang Wei, Pu Yi, Zhengxi Li, Tao Xie, Darko Marinov, and Wing Lam. 2022. Preempting flaky tests via Non-Idempotent-Outcome tests. In *International Conference on Software Engineering (ICSE'22)*. 1730–1742.
[67] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2021. Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 270–287.
[68] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. 2017. Memory mutation testing. *Information and Software Technology* 81 (2017), 97–111.
[69] xmlgraphics 2023. xmlgraphics Project. https://github.com/apache/xmlgraphics-commons
[70] Pu Yi, Anjiang Wei, Wing Lam, Tao Xie, and Darko Marinov. 2021. Finding polluter tests using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 46, 3 (2021), 37–41.
[71] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 385–396.

Received 2023-02-16; accepted 2023-05-03