Optimizing Search for Tests Relevant to Order-Dependent Flaky Tests

Suzzana Rafi[1], Md Mahmudul Hasan Pious[1], Shanto Rahman[2], August Shi[2], Wing Lam[1]
[1] George Mason University, Fairfax VA, USA
{srafi,mpious,winglam}@gmu.edu
[2] The University of Texas at Austin, Austin TX, USA
{shanto.rahman,august}@utexas.edu

***Abstract*—A key challenge in regression testing is dealing with flaky tests, which can fail and pass non-deterministically when run on the same version of code. A prominent type of flaky tests is order-dependent flaky tests (OD tests), which pass or fail based on the order in which the tests are run, and the order may often change during testing. These tests' outcomes are affected when previously run tests, called OD-relevant tests, modify the state shared between the tests. Detecting OD-relevant tests is key to understanding and fixing OD tests so their outcomes are not affected by which tests ran before. To detect OD-relevant tests, prior work proposed iFixFlakies, a delta-debugging approach that runs subsequences of a given test list before the OD test to find the minimal sequence of tests that affect its outcome. Prior work also found that this minimal sequence is rarely more than one test, suggesting that future work should focus on detecting just one single OD-relevant test.**

Our intuition is that the default iFixFlakies approach can be more efficient if we use additional heuristics to guide its search. We propose three different types of search options to reduce the overall search time of iFixFlakies: (1) split options that leverage hierarchical, code similarity, and historical information to better form the subsequences for further search, (2) a pick option that prioritizes running a subsequence of tests based on its expected running time, and (3) a run option that can skip running entire sequences of tests. We evaluate our new options on 155 OD tests from 34 open-source Java projects' test suites. We find that some options help reduce the overall running time of the default iFixFlakies that does not use any of our options. Furthermore, we find that the best combination of our options for each OD test type detects OD-relevant tests on average 32.57-42.42% faster than the default iFixFlakies approach, and 49.84-95.96% faster than other baselines.

***Index Terms*—Regression Testing, Debugging Flaky Tests, Order-Dependent Flaky Tests**

I. Introduction

Regression testing is an important part of the software development process, widely practiced in both industry [1]–[5] and open-source [6], but suffers from the presence of flaky tests. *Flaky tests* can both pass and fail when run on the same version of code [7], [8]. Many software development organizations report that flaky tests are one of their biggest problems [1], [9]–[26]. Several studies [8], [21], [27], [28] found one prominent type of flaky tests to be *order-dependent flaky tests* (OD tests), which can pass or fail due to the order in which the tests are run and where this ordering may often be different [29]. OD tests can fail because they depend on some global state [7], such as static variables, files, and databases, but other tests set this state to undesirable values when run beforehand. Prior work referred to tests that determine whether an OD test passes or fails as *OD-relevant tests* [30].

Detecting OD-relevant tests can help developers debug the corresponding OD tests, and prior work proposed iFixFlakies [31], a delta-debugging [32] based approach, to find the smallest sequence of tests that, when run before the OD test, causes it to pass or fail as needed. iFixFlakies *splits* the candidate tests into two subsequences, giving each subsequence the same number of tests. Among the two subsequences, iFixFlakies *picks* the subsequence closest to the OD test in a given order[1] to run first, where both subsequences may be *run* before the OD test to check whether the subsequence preserves the desired pass/fail outcome. If the picked subsequence does not preserve the desired pass/fail outcome, iFixFlakies runs the other subsequence. It then recursively searches whichever subsequence preserves the outcome until the OD-relevant test is found. We refer to each choice that iFixFlakies uses to split, pick, and run tests during its search as an *option*. We call a combination of options as an iFixFlakies *variant*, with the combination of options that iFixFlakies currently uses by default referred to as the *default variant*.

Despite iFixFlakies's usefulness, we find that its three default options may not be properly minimizing its runtime. First, splitting subsequences by the number of tests may not optimally guide the search toward the OD-relevant test, so we use hierarchical information, code similarity, and historical information to improve the split. Second, always running the closest subsequence first may waste time when the other subsequence is faster, so we instead prioritize the subsequence with the lower runtime. Third, when searching for a single OD-relevant test, running both subsequences can be unnecessary, so we propose an option to run only the selected subsequence at each step. It continues searching for that subsequence if it produces the desired outcome; otherwise, it immediately searches the other subsequence without first running it. Using this option only detects OD-relevant tests composed of a single test, even though it may be necessary for multiple tests to run together to affect an OD test's outcome. However,

[1]All OD tests by definition must have at least one order in which the test deterministically passes, and another order in which the test deterministically fails. Prior work [7], [33] has suggested rerunning passing and failing orders to categorize flaky tests as OD.

iFixFlakies's evaluation found that almost always only one test is enough to affect the outcome of the OD test [31], so this option is more practical in the vast majority of cases for detecting the single OD-relevant test; we discuss a case study of our other options for multi-test OD-relevant tests.

We evaluate our options using 155 OD tests from 34 test suites obtained from popular open-source Java projects used in prior work [34]. In theory, we can evaluate up to 20 variants (combinations of options) – five from split options (the default option to split by number of tests, along with one hierarchical, two code similarity, and one historical information based split), two from pick options (pick latter or runtime), and two from run options (run both or only one subsequence). We perform an ablation study to see which of the proposed options has the greatest effect on improving the default variant's detection efficiency of OD-relevant tests. Depending on the type of OD-relevant test, the variants using the best corresponding split options detect OD-relevant tests 7.44-22.63% faster than the default variant. We find that using the proposed pick option by itself does not improve the default variant for any type of OD-relevant test, but that the option does help when it is combined with other options. Lastly, depending on the OD-relevant test type, variants using our run option detect them 13.91-24.13% faster than the default variant. To create the best overall variants, we combine our proposed pick and run options with each of our split options. We find that our best variant for each OD-relevant test type detects them 32.57-42.42% faster than the default variant, and 49.84-95.96% faster than other baselines.

We make the following main contributions in this paper:

- We propose three types of options to an existing approach for detecting OD-relevant tests.
- We perform an ablation study with each of our options on 34 test suites, containing 155 OD tests, from popular open-source projects. We find that using our proposed run option in a variant consistently improves runtime, while split options are most effective when they are combined with our pick and run options.
- We combine our options to form variants, where we find a different variant is best for different types of OD-relevant tests. Our best variant for each type substantially outperforms state-of-the-art approaches. Our implementations of the variants, data, and scripts are publicly available [35].

II. Background

A. Order-Dependent Flaky Test

An order-dependent flaky test (OD test) is a test whose outcome (pass or fail) depends on the execution order of other tests [7], [8], [33]. We define a test-order as a *passing test-order* if the OD test passes when run in that test-order, and as a *failing test-order* if it fails. OD tests can fail in two ways: (1) another test is polluting the shared state when it is run before the OD test, or (2) another test is supposed to set up the required shared state but it runs after the OD test.

TABLE I: Iterations of the iFixFlakies search process. Bold indicates executed tests in each step, gray cells indicate skipped tests, neon marks detected OD test, light neon indicates OD test failed (F), pink indicates the OD test passed (P).

Steps	Test Methods (Class.method)						Result
	C2.m1	C2.m2	C1.m2	C3.m1	C3.m2	C3.m3	F
1	C2.m1	C2.m2	C1.m2	**C3.m1**	**C3.m2**	**C3.m3**	P
2	**C2.m1**	**C2.m2**	**C1.m2**	C3.m1	C3.m2	C3.m3	F
3	C2.m1	C2.m2	**C1.m2**	C3.m1	C3.m2	C3.m3	F

Shi et al. previously provided names for different types of OD tests and their related tests [31]. If an OD test passes when run in isolation but fails after another test, it is called a *victim*, and the test that ran beforehand and "pollutes" the shared state is called a *polluter*. Conversely, if an OD test fails even when run on its own, it is considered a *brittle*, because it relies on another test, called a *state-setter*, to run beforehand and initialize some shared state. Finally, some tests can restore or clean the shared state when placed between a polluter and a victim, preventing the failure. These tests are known as *cleaners*. We refer to polluters, state-setters, and cleaners as *OD-relevant tests*.

B. iFixFlakies and Delta-debugging

iFixFlakies [31] detects OD-relevant tests, which it uses to generate fixes, by leveraging delta-debugging [36]. For polluter search, iFixFlakies first takes as input a failing test-order for the OD test. It creates two sequences of tests (e.g., T_1 and T_2) using the tests before the OD test in the given test-order by splitting that sequence of tests that form the prefix in half and always picking the latter sequence to run first (e.g., T_2). If running a sequence produces the desired behavior (e.g., running T_2 before the OD test causes it to fail as expected), then the algorithm searches further down that sequence. Otherwise, iFixFlakies runs the other sequence to check if that one (e.g., T_1) produces the desired behavior. This process continues until it detects a minimal OD-relevant sequence that is sufficient to reproduce the OD test's pass/fail behavior [31], [32].

To illustrate a polluter search, we provide an example in Table I. The table's leftmost column, labeled "Steps" indicates the delta-debugging iteration number. The next six columns represent the tests (m1, m2, m3) alongside the test classes they belong to (C1, C2, C3). These tests come before the victim in each step, and the victim is not shown in the table. The rightmost column ("Result") shows the victim's outcome in each step. When we run the whole sequence, the victim fails, and our goal is to detect a single OD-relevant test that causes the same failure. From the table, we see that iFixFlakies first picks the right-side sequence to explore, executing the tests (C3.m1, C3.m2, C3.m3). As the victim passes when run after the tests in the right-side sequence, the algorithm then inspects the left-side sequence (C2.m1, C2.m2, C1.m2), running its tests before the victim to check for a failure. In the third iteration, this left-side sequence is further split, with iFixFlakies picking the right-side sequence from this new split,

where it runs C1.m2 before the victim, finds the victim to fail, and detects C1.m2 as the polluter for the victim.

The process is similar for state-setter search, except that iFixFlakies takes a passing test-order and searches for a test that causes the brittle to pass. The cleaner search requires a known polluter and victim. For each candidate sequence, iFixFlakies runs the polluter, followed by the candidate sequence and then the victim, and checks whether the victim passes. A passing result indicates that the sequence contains a cleaner that restores the state modified by the polluter.

C. One-by-one, RankF$_L$, and RankF$_O$

Another approach to detect OD-relevant tests is the one-by-one approach proposed by Lam et al. [37], which runs each test with the OD test as a pair. The order in which one-by-one runs each test with the OD test is based on an input passing/failing test-order. Rahman et al. [30] proposed ranking tests based on how likely they are to be OD-relevant tests with two techniques: RankF$_L$ and RankF$_O$. RankF$_L$ analyzes the method bodies of each test with a given OD test using a fine-tuned pre-trained language model (BigBird [38]) to estimate the likelihood that any test is relevant to the OD test. RankF$_O$ relies on historical test-order execution data to compute likelihood scores per test based on how frequently and how closely tests appear before the OD test in past passing and failing runs. Both approaches rank the tests according to their scores and finally run each test in order of the ranked list together with the OD test one-by-one.

D. Example

Figure 1 shows an OD test from `ktuukkan/marine-api`, an open-source GitHub project in our evaluation. The OD test involves the victim test `testConstructor` (Line 13), which passes in isolation but fails after `testRegisterParserWithAlternativeBeginChar` (Line 4), making `testRegisterParserWithAlternativeBeginChar` its polluter. The two tests are in different test classes. `testRegisterParserWithAlternativeBeginChar` modifies shared state by unregistering `VDM-Parser.class` (Line 6), which removes it from the static parsers map (Line 17). The victim test expects this entry. When it is missing, parser creation fails, and the victim throws an exception.

A cleaner is available, namely `testCreateParser` (Line 8). When executed between the polluter and victim, it restores the expected state. Specifically, a setup method in this class (Line 3) calls `reset` on the shared `SentenceFactory` instance (Line 2), repopulating `parsers` (Line 24), including `VDM-Parser.class` (Line 25). As the cleaner is in the same class as the polluter, any test in that class resets the state. However, if the polluter runs last and the next test is the victim, then the state remains polluted and the victim fails. Searching each test one-by-one takes 825.85 seconds to find this cleaner, while iFixFlakies takes 20.61 seconds. Meanwhile, iFixFlakies improved with our proposed options can find the cleaner in 10.00 seconds, giving a 98.79% improvement over one-by-one and 51.48% improvement over the default iFixFlakies. Furthermore, comparing against RankF$_L$ and RankF$_O$, we can achieve an 87.85% and 26.42% improvement over RankF$_L$ and RankF$_O$, respectively.

```
1  public class SentenceFactoryTest {
2    private final SentenceFactory instance =
       SentenceFactory.getInstance();
3    @Before public void setUp() { instance.reset(); }
4    @Test public void
       testRegisterParserWithAlternativeBeginChar() {
5      ...
6      instance.unregisterParser(VDMParser.class);
7    }
8    @Test public void testCreateParser() { ... }
9  }
10 public class AbstractAISMessageListenerTest {
11   private final SentenceFactory sf = SentenceFactory
       .getInstance();
12   private final AISSentence AIS_01 = (AISSentence)
       sf.createParser("VDM");
13   @Test public void testConstructor() { ... }
14 }
15 public final class SentenceFactory {
16   // map containing parser classes
17   private static Map<String, Class<? extends
       SentenceParser>> parsers;
18   public void unregisterParser (Class<? extends
       SentenceParser> parser) {
19     for (String key : parsers.keySet()) {
20       if (parsers.get(key) == parser) {
21         parsers.remove(key);
22         break;
23   }  }  }
24   public void reset() {
25     registerParser(tempParsers, "VDM", VDMParser.
         class);
26     ...
27     parsers = tempParsers;
28 }  }
```

Fig. 1: OD test from `ktuukkan/marine-api` [39].

III. Methodology

To improve the efficiency of iFixFlakies at detecting OD-relevant tests, we propose new search *option*s that enhance different components of the overall search process, focusing on the options to improve the splitting, picking, and running of subsequences. Combining these options forms *variant*s of iFixFlakies, which we believe can reduce the time to find OD-relevant tests over the default variant of iFixFlakies.

A. iFixFlakies Options

Figure 2 shows an overview of the iFixFlakies delta-debugging approach and our proposed options to detect a single OD-relevant test faster. iFixFlakies's delta-debugging is an iterative process, and each iteration operates in three main steps. Given a sequence of tests, the iFixFlakies default variant first splits the sequence into two subsequences. It then runs one of the subsequences to see whether those tests can change the OD test outcome when run before. If that subsequence does not change the OD test outcome, then the default variant runs the other subsequence to check whether running with the other does indeed change the OD test outcome. We propose

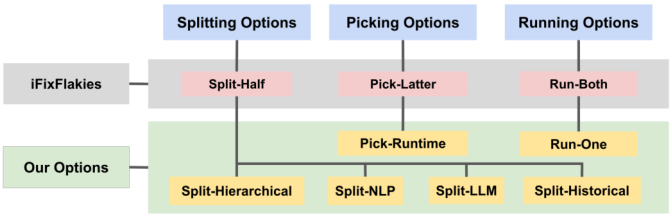


Fig. 2: iFixFlakies overview and our proposed options.

new options that enhance each of these three steps. Overall, when including the default options for each component used currently by iFixFlakies, we have five splitting options, two picking options, and two running options, enabling up to 20 possible variants through their combinations.

B. Splitting Options

1) Split-Half (Default): The default variant uses an option for splitting that splits the sequence into two subsequences with an equal number of tests each, which is the standard delta-debugging approach [32], [36].

2) Split-Hierarchical: We propose an option that leverages the hierarchical relationship between tests, namely how they are organized within test classes. The intuition is to reduce the search space and improve localization efficiency by splitting based on test classes. As there are fewer test classes than individual test methods in a sequence, we reduce the number of iterations necessary to detect the single OD-relevant test. We split the sequence into two subsequences that contain an equal number of test classes (while preserving the original order of test classes from the initial sequence). If the sequence consists of just a single test class, we switch to the default option and split into equal subsequences of tests within that test class.

3) Split-NLP: We define a new splitting option that splits the sequence into two subsequences with equal likelihood of OD-relevant tests being in either one, based on text-based similarity. We rely on TF-IDF [40] to compute similarity. Our intuition is that OD tests and corresponding OD-relevant tests have similar text (e.g., sharing the same global state), so text-based similarity scores can help identify such relationships.

The process begins by computing the cumulative TF-IDF scores for the entire test suite $T = \{t_1, t_2, \ldots, t_n\}$, where each test t has a TF-IDF score with respect to the OD test, denoted as $\text{TfIdf}(t, \text{OD})$. After calculating these scores, we split the test list into two subsequences T_L (left) and T_R (right) while preserving the original order of tests in the sequence. To create a balanced split, we first compute the total cumulative score for the entire sequence:

$$S(n) = \sum_{k=1}^{n} \text{TfIdf}(t_k, \text{OD})$$

We then incrementally add tests to the left subsequence T_L until the running sum $S(i) = \sum_{k=1}^{i} \text{TfIdf}(t_k, \text{OD})$ reaches half of the total score. This split preserves the order of the original sequence and ensures both subsequences are balanced in terms of cumulative textual similarity to the OD test.

4) Split-LLM: With a similar intuition as Split-NLP, we can also compute similarity using LLMs, which can capture code similarities based on semantics. Specifically, we reuse the same fine-tuned LLM and likelihood scores produced by RankF_L [30]. Our intuition is that these scores capture deeper contextual relationships that go beyond textual overlap, such as shared functional intent or similar state setup. We split tests the same way as Split-NLP, except using the RankF_L scores.

5) Split-Historical: Another way to compute scores representing likelihood of a test being an OD-relevant test to an OD test is through historical test-order analysis. Using the same insights as prior work, RankF_O [30], we can compute likelihood scores based on how frequently and how closely each test appears before the OD test in previously observed failing and passing test-orders. We use the Plus-One heuristic together with the combined class strategy as recommended in the prior study to compute scores [30], and we split the tests same way as Split-NLP, except using the RankF_O scores.

C. Picking Options

1) Pick-Latter (Default): Delta-debugging can choose to run either subsequence first, as they should be "equal" in likelihood of containing the desired OD-relevant test. The default variant always chooses the right subsequence first, i.e., the subsequence of tests that are closer to the OD test in the original sequence [31]. The intuition is that the OD-relevant test that causes the OD test outcome to change is likely to reside in the same class as the OD test, as tests within the same class are more closely related and thus more likely to "interfere" with each other. While this deterministic choice simplifies the search process, it can be suboptimal in cases where the right subsequence does not contain the OD-relevant test, resulting in extra runtime.

2) Pick-Runtime: Given that the goal is to reduce the overall runtime, we propose picking the subsequence that takes less time to run, based on individual test runtimes collected from the previous run. If the two subsequences are "equal" to each other (based on the splitting options' heuristics from Section III-B), then picking to run the one with less runtime should decrease overall runtime, especially if an OD-relevant test can be found in the faster-running subsequence.

D. Running Options

1) Run-Both (Default): In the default variant's search process, if the subsequence it first picks to run in each iteration does not result in the desired OD test outcome, then it always runs the other subsequence to check that this other subsequence does have that outcome, meaning it contains the OD-relevant test. This "run and check" is in case of running this other subsequence also does not result in the desired outcome – meaning the "OD-relevant test" is actually a sequence of multiple tests, at which point, delta-debugging will create other subsequences to find multi-test OD-relevant tests [36], [41].

2) Run-One: However, prior work found that the vast majority of OD tests have outcomes affected by only a single OD-relevant test [30], [31], so they recommend enhanced search processes that target finding just a single test as the OD-relevant test. Thus, we propose a new option that optimizes the search process by needing to only run one subsequence per iteration, the first one picked. If that one does not result in the desired OD test outcome, we immediately assume the other subsequence must contain the OD-relevant test, and we begin the next iteration immediately using the other subsequence without needing to first run it. This option explicitly assumes the majority case when there is only single OD-relevant tests, and our evaluation shows that it substantially reduces the runtime to detect these OD-relevant tests. If a developer suspects that assumption to not hold, they can forgo the Run-One option to run both subsequences, like the default variant.

E. Algorithm of Using Proposed Options

Figure 3 shows the pseudo-code of our proposed options to identify a polluter. Note that if searching for state-setters or cleaners, the algorithm would be the same except the desired outcome would be a "pass" instead of a "fail," as a state-setter causes a brittle to pass, and a cleaner causes a victim-polluter pair to pass. The core function, `search`, takes as input the current list of tests, the splitting option (`split_tests` or `split_scores`), and the side-picking function (`pick_side`). The `split_pick` function applies the selected splitting option to divide the current test sequence into two complementary subsequences and then invokes the selected picking option to determine which subsequence should be executed first.

Line 16 performs the split. If using `split_tests`, then tests are divided evenly by count, while using `split_scores` divides the tests as to minimize the difference between the total similarity scores of the two resulting subsequences. The `pick_runtime` at Line 24 always selects the subsequence with the smaller aggregated runtime, ensuring that the next iteration explores the faster side first. The `pick_side` parameter allows the algorithm to use either the default `pick_latter` option or the proposed `pick_runtime` option. In Figure 3, the illustrated variants pass `pick_runtime` to `search`. To further reduce redundant runs, the `run_one` function enforces the *Run-One option* that executes only one subsequence per iteration instead of both. Depending on whether the executed subsequence passes or fails, Line 18 selects the subsequence for the next iteration. If the sequence passes, the search continues on the other subsequence since the executed sequence can be ignored. If it fails, the search continues on the executed subsequence. Line 19 then recursively invokes `search` on the selected subsequence, repeating this process until the polluter is isolated. The `getruntime(t)` function returns the runtime of a test `t`, and `runtime(seq)` computes the total runtime of a sequence `seq`. The `run(seq)` function executes a sequence `seq` followed by the OD test and reports whether it passes or fails. The remaining helper functions

```
1  # --- Helper functions ---
2  es = dict() # test names to scores
3  def runtime(seq): # total runtime
4    return sum(getruntime(t) for t in seq)
5  # --- Split sequence and pick one side ---
6  def split_pick(elements, split_func, pick_func):
7    left, right = split_func(elements)
8    sequence = pick_func(left, right)
9    other = right if sequence == left else left
10   return sequence, other
11 # --- Searching algorithm ---
12 def search(elements, test_or_score, pick_side):
13   if len(elements) <= 1:
14     return elements if run(elements)==fail else None
15   # split and pick side
16   sequence, other = split_pick(elements,
       test_or_score, pick_side)
17   res = run_one(sequence) # Run-One: skip other side
18   next_elements = other if res == pass else sequence
19   val = search(next_elements, test_or_score,
       pick_side)
20   return val
21 # --- Pick options ---
22 def pick_latter(l, r): # default Pick-Latter option
23   return r
24 def pick_runtime(l, r):
25   return l if runtime(l) <= runtime(r) else r
26 # --- Run-One option ---
27 def run_one(seq): # run only one side
28   return run(seq)
29 # --- Variants using split options ---
30 def split_hierarchical(classtest):
     # Split-Hierarchical
31   elements = classtest.keys()
32   es = {key: 1 for key in elements}
33   found_class = search(elements, split_tests,
       pick_runtime)
34   es = {key: 1 for key in get_tests(found_class)}
35   return search(get_tests(found_class), split_tests,
       pick_runtime)
36 def split_nlp(classtest): # Split-NLP
37   elements = classtest.values()
38   es = {key: getnlpscore(key) for key in elements}
39   return search(elements, split_scores, pick_runtime
       )
40 def split_llm(classtest): # Split-LLM
41   elements = classtest.values()
42   es = {key: getllmscore(key) for key in elements}
43   return search(elements, split_scores, pick_runtime
       )
44 def split_historical(classtest): # Split-Historical
45   elements = classtest.values()
46   es = {key: getrfoscore(key) for key in elements}
47   return search(elements, split_scores, pick_runtime
       )
```

Fig. 3: Pseudo-code of our proposed variants when searching for polluters. We apply *Run-One* by executing only one side per iteration (`run_one`), and we apply *Pick-Runtime* via `pick_runtime`. Splitting uses either Split-Half by count (`split_tests`) or score-balanced splitting (`split_scores`).

assign scores (`setele`), compute NLP-, LLM-, and historical information-based scores (`getnlpscore`, `getllmscore`, and `getrfoscore`, respectively), and retrieve tests in a class (`get_tests`). The `split_nlp`, `split_llm`, and `split_historical` functions apply score-balanced splitting based on textual (TF-IDF), semantic (LLM embedding)

TABLE II: Details of our evaluation subjects. "V." represents Victim, "P.-V." represents Polluter-Victim Pair.

ID	Project	Number of Tests	OD	V.	P.-V.
M1	Activiti/Activiti	42	21	17	115
M2	Apache/Struts	61	22	22	77
M3	alibaba/fastjson	4781	2	0	0
M4	apache/dubbo	106	3	3	8
M5	apache/dubbo	499	1	1	0
M6	apache/dubbo	5	1	1	0
M7	apache/dubbo	58	2	2	2
M8	apache/dubbo	65	2	1	4
M9	apache/dubbo	21	2	2	13
M10	apache/hadoop	388	1	1	1
M11	c2mon/c2mon	5	1	1	0
M12	ctco/cukes	14	1	1	1
M13	doanduyhai/Achilles	233	1	0	0
M14	dropwizard/dropwizard	74	1	1	2
M15	elasticjob/elastic-job-lite	479	2	2	4
M16	fhoeben/hsac...	251	1	0	0
M17	hexagonframework/spring...	48	4	0	0
M18	jhipster/jhipster-registry	53	2	2	2
M19	kevinsawicki/http-request	163	25	25	24
M20	ktuukkan/marine-api	925	12	12	12
M21	openpojo/openpojo	588	3	3	12
M22	spring-projects/spring-boot	1734	2	2	2
M23	spring-projects/spring-boot	735	6	6	0
M24	spring-projects/spring-boot	584	1	0	0
M25	spring-projects/spring-boot	219	1	0	0
M26	spring-projects/spring...	10	2	2	2
M27	spring-projects/spring-ws	61	2	2	2
M28	tbsalling/aismessages	44	2	2	0
M29	vmware/admiral	478	1	0	0
M30	vmware/admiral	302	1	1	0
M31	wikidata/wikidata-toolkit	49	3	0	0
M32	wikidata/wikidata-toolkit	23	2	2	0
M33	wildfly/wildfly	81	21	20	0
M34	zalando/riptide	40	1	1	0
Total		**13219**	**155**	**135**	**283**

similarity, and historical test order analysis, respectively. In contrast, `split_hierarchical` first splits by test class and then searches tests within the selected class.

IV. Experimental Setup

Our evaluation addresses the following research questions:

RQ1. How well do individual options contribute towards variants at detecting OD-relevant tests?

RQ2. How efficient are variants that combine our options at detecting OD-relevant tests?

A. Dataset

We use an existing dataset from prior work by Rahman et al. [30] that contains reproduced OD tests and the corresponding OD-relevant tests from the original dataset by Wei et al. [34]. The dataset also provides the test method bodies, test runtimes, and 10 passing and failing test-orders that Rahman et al. used for ranking OD-relevant tests and evaluating RankF$_L$ and RankF$_O$. We exclude two cleaner cases because our evaluation assumes an available passing test-order in which the known polluter precedes the victim. Such an order is required for delta-debugging to search the tests between the polluter and victim for a cleaner. If no such order exists, then the victim is either before the polluter and is passing in this order, or after the polluter and is failing in this order. In either of these cases, iFixFlakies must perform additional executions before it can start delta-debugging by placing subsets of the remaining tests between the known polluter and victim until it finds a passing polluter-before-victim order. For simplicity, our evaluation contains only cleaner cases for which at least one of the provided test-orders is a passing polluter-before-victim order. After these exclusions, our final dataset contains 155 OD tests and their respective OD-relevant tests across 34 modules. Table II provides details, including the number of tests, OD tests, and OD-relevant tests in each module.

B. Baselines

For comparison, we use four baselines: (1) iFixFlakies' default variant [31], (2) RankF$_L$ [30], (3) RankF$_O$ [30], and (4) a one-by-one search (OBO) [42]. For OBO, we test candidate tests sequentially in the order in which they appear in each provided test-order until the first OD-relevant test is detected. We apply OBO to the same 10 passing or failing test-orders used for the other approaches.

C. Simulation Setup

As we have complete information on known OD tests and corresponding OD-relevant tests from prior work [30], [34], along with each test's runtime, we can simulate the total time needed to run each variant. To obtain each test's runtime, prior work [30] ran each module's full test suite using 10 random test orders, repeated each order three times, and averaged each test's runtime across the 30 runs. All these tests belong to projects that build using the Maven build system, which uses Maven Surefire to run the tests. Each time tests are run, an overhead is incurred from starting/ending the Surefire process used for running tests. The prior work obtained the average Surefire overhead from these runs by subtracting the total test runtime reported by Surefire from the total time required to run the test suite. We reuse the test-orders and scores from prior work [30]: 10 passing or failing test-orders per OD test for RankF$_L$, and 10 sets of 20 historical test-orders per module for RankF$_O$. 10 sets are used to obtain an average as different historical test orders can affect RankF$_O$'s performance, and the recommended default is to use 20 orders in each set as prior work found RankF$_O$ to be effective with just 20 orders. For any sequence of tests, we know whether the OD test passes or fails by checking the positions of OD-relevant tests relative to the OD test without needing to run the tests. We estimate the total time to run all tests in a sequence by summing the average runtimes of the tests.

D. Evaluation Metrics

We measure time in seconds for all RQs. The estimated time to detect an OD-relevant test includes: (1) the cumulative runtime of all tests executed in each candidate sequence (i.e., subsequences of the original test-order), (2) the runtime of the OD test is appended at the end of each sequence to check whether an OD-relevant test is present, and (3) Surefire overhead incurred when running a sequence. When using

the Split-NLP and Split-LLM options, we also include the time required to compute similarity scores. We compute these scores once per input test-order and reuse them across all sequence runs for that order.

$$\text{Total Runtime} = \sum_{i=1}^{n} \Big(\Big(\sum_{t \in \text{Seq}_i} \text{Runtime}_t \Big) + \text{Runtime}_{\text{ODTest}} + \text{Overhead}_{\text{Surefire}} \Big) + \text{ScoreComputationTime}$$

where $\sum_{t \in \text{Seq}_i} \text{Runtime}_t$ is runtime of all tests in the i-th sequence, $\text{Runtime}_{\text{ODTest}}$ is the OD test runtime, $\text{Overhead}_{\text{Surefire}}$ is the overhead incurred from running Surefire, n is the total number of sequence executions, and ScoreComputationTime is time taken to compute NLP- and LLM-based scores. All our NLP- and LLM-based results include this one-time cost.

All averages in Tables III, IV, and V are weighted averages, because different modules may contain a different number of OD tests, and weighting ensures that larger modules contribute proportionally to the result. All experiments use Ubuntu 22 LTS machines, with i7 8-core processors and 32 GB RAM.

V. Evaluation

A. RQ1: Efficiency of Using Individual Options

We conduct experiments to evaluate the efficiency in using each of our options individually in isolation, comparing them against the default variant. Tables III-V show the seconds to find the polluter, state-setter, and cleaner, respectively, for all of our baselines, variants that use our options individually (forming an ablation study of our options), and variants that combine multiple options.

In our ablation study, the Run-One option helps achieve the lowest runtime for detecting two of the three OD-relevant test types (polluters and cleaners), making it the most efficient standalone option to help variants in most cases. This option shows that avoiding redundant executions is generally an efficient way to reduce runtime. However, Run-One does not always help reduce the runtime the most. For finding state-setters, splitting options, especially Split-NLP and Split-Historical, helps more than Run-One by creating more efficient splits that reduce the number of search iterations. Run-One cannot always compensate for an inefficient split. When using the default half-split, the test set shrinks slowly, which leads to many delta-debugging iterations, and even using Run-One did not reduce runtime the most. In these cases, a better split can be more efficient than Run-One alone. For finding state-setters, some splitting options produce more efficient splits, resulting in fewer iterations, thus running faster. For example, for M25, using Split-NLP helps identify the state-setter in three iterations and using Split-Hierarchical helps identify it in four, whereas using Split-Half requires seven iterations.

RQ1: *Run-One is generally the strongest individual option, followed closely by the new splitting options.*

B. RQ2: Efficiency of Variants Combining Multiple Options

We also evaluate which variant that combines multiple options is the most efficient at detecting an OD-relevant test. We construct our variants by combining the Run-One and Pick-Runtime options with all split options: Split-Half, Split-Hierarchical, Split-NLP, Split-LLM, Split-Historical, resulting in five variants in total.

From Tables III–V, we observe that using Pick-Runtime alone is not as efficient in consistently reducing runtime. However, when we combine it with our Run-One option, the resulting variant becomes substantially more efficient. From Tables III–V, the column Pick-Runtime under the "Ablation Study" header shows results of the variant combining Split-Half, Pick-Runtime, and Run-Both options, while the column Split-Half under the "Our Variants" header shows the results of the variant combining Split-Half, Pick-Runtime, and Run-One option (so they differ only by the run option while both using the default Split-Half option). The variant using Pick-Runtime with Run-One achieves up to 36.14% speedup over Pick-Runtime without it, across all OD-relevant test types. For example, when finding polluters, the average runtime decreases from 42.52 seconds to 33.46 with Run-One, and further to 31.96 after adding Pick-Runtime. The same pattern holds for state-setters and cleaners, showing that Run-One works better with Pick-Runtime. As we only run one of the subsequences per iteration with the Run-One option, picking the one with less runtime makes a bigger overall impact. Figure 4 compares each split option with and without Run-One and Pick-Runtime, showing that adding both reduces runtime by 15.90–35.70% across OD-relevant test types.

When we combine the picking and running options with more informed splitting options, the runtime improves further. Across polluters, state-setters, and cleaners, we find that Split-Hierarchical, Split-NLP, and Split-Historical options consistently help achieve the best performance, with only some small differences among them. For example, for cleaner detection, Split-Hierarchical and Split-Historical have nearly identical runtimes (22.88 and 22.99 seconds). These results suggest that any of the three splitting options is effective in practice. Across polluters, state-setters, and cleaners, they achieve 32.59%–42.42% speedup over the best baseline and provide up to 28.22% improvement over a variant using the default Split-Half. Because detecting OD-relevant tests is required before debugging and repair, these improvements directly reduce the runtime of the overall workflow to repair OD tests.

We observe that using Split-LLM leads to worse results than when using the other splitting options, including the default Split-Half. However, we also observe that the main cost comes the large inference time. If we separate out the inference time, using Split-LLM now achieves an improvement of 5.19%-11.56%, with a weighted average of 33.16 seconds for finding polluters, 74.60 seconds for finding state-setters and 24.49 seconds for finding cleaners. However, using Split-LLM under these conditions still leads to results worse than variants using our other proposed options, indicating that the current

TABLE III: Runtime for finding polluters in seconds. Bold indicates the best result within each group.

	Baseline				Ablation Study of Our Options						Our Variants				
ID	iFixFlakies's Default Variant	RankF$_L$	RankF$_O$	OBO	Split-Hierarchical	Split-NLP	Split-LLM	Split-Historical	Pick-Runtime	Run-One	Split-Half	Split-Hierarchical	Split-NLP	Split-LLM	Split-Historical
M1	29.99	32.96	**18.77**	31.30	27.94	31.12	41.53	33.37	31.99	**23.49**	24.38	23.81	22.00	28.37	**21.34**
M2	**13.63**	15.05	14.32	31.11	**9.04**	13.57	17.02	13.95	13.97	9.50	9.06	**8.53**	8.90	11.97	8.57
M4	16.74	144.85	**13.86**	77.81	14.82	14.25	18.60	16.80	16.18	**12.34**	11.00	10.46	8.48	13.04	**7.72**
M5	43.43	40.49	**5.54**	430.37	34.12	**22.39**	48.91	39.16	53.52	43.11	20.67	16.62	**13.62**	24.01	18.40
M6	**2.53**	334.48	2.68	4.56	**1.77**	2.49	2.55	1.77	2.53	1.77	**1.77**	1.77	2.37	2.55	1.77
M7	13.08	**4.66**	6.84	42.16	16.47	18.01	14.55	13.01	13.37	**9.33**	8.61	10.81	11.92	9.52	**8.49**
M8	265.85	**59.03**	176.24	209.28	338.05	440.27	308.47	326.35	311.44	**236.15**	**241.06**	245.56	351.02	248.57	273.26
M9	5.34	100.25	**4.10**	5.66	5.28	5.95	6.39	5.40	4.02	**3.59**	3.61	3.75	3.79	4.83	**2.85**
M10	500.71	**8.44**	49.21	758.66	405.57	499.06	550.60	436.67	509.56	**337.51**	285.65	317.82	295.51	313.91	**224.98**
M11	20.56	29.99	**15.66**	36.15	24.76	21.80	11.05	**10.96**	15.86	15.86	15.86	15.71	21.00	11.05	**10.96**
M12	4.60	**2.28**	2.48	12.18	5.69	4.19	**3.91**	5.29	6.24	4.12	3.96	4.11	3.47	**3.46**	5.12
M14	24.37	84.06	**21.25**	95.03	**18.24**	34.48	29.75	28.00	32.84	20.39	13.78	**12.96**	18.25	16.72	14.60
M15	**620.72**	2679.20	5614.95	14670.85	607.66	560.25	631.05	559.43	726.86	**505.77**	496.29	406.41	442.90	488.14	**403.95**
M18	69.53	215.62	**35.95**	124.52	51.07	55.91	59.77	54.40	76.56	**44.51**	44.22	49.38	37.79	40.33	**33.54**
M19	13.11	100.83	**2.23**	89.70	**3.47**	15.70	16.02	12.25	13.64	8.97	**8.92**	9.30	11.40	12.22	10.03
M20	**20.96**	203.53	136.86	402.77	**12.09**	21.42	32.99	22.19	21.42	13.45	13.24	**13.15**	14.62	27.28	16.21
M21	**43.97**	286.50	100.84	489.92	44.02	45.72	54.39	128.42	42.88	**31.24**	26.30	**25.13**	27.95	35.36	37.69
M22	**154.51**	3662.60	17697.50	5987.41	179.94	**119.96**	182.48	124.82	207.59	129.99	124.83	124.39	**92.74**	152.37	96.72
M23	138.37	2773.34	**13.57**	5529.71	182.49	98.89	166.69	**93.79**	205.76	126.09	120.18	117.36	84.04	150.18	**74.59**
M26	15.72	25.40	**15.01**	24.89	13.28	24.46	17.83	16.45	17.24	**11.71**	**11.40**	14.01	18.58	13.94	12.39
M27	13.65	20.49	**5.08**	48.46	14.66	10.14	13.57	13.37	14.44	**9.63**	9.64	10.10	**7.39**	10.37	9.72
M28	4.00	2.02	**0.90**	20.91	5.19	3.18	6.24	**2.77**	4.62	3.65	3.64	3.76	2.76	5.75	**2.40**
M30	225.67	332.25	**6.07**	913.24	252.68	318.72	247.69	**90.93**	323.72	225.67	198.99	164.98	236.00	214.60	**104.42**
M32	4.65	3.03	**1.32**	21.03	**3.48**	4.33	7.65	4.70	5.98	4.23	4.09	**2.97**	3.63	6.97	3.66
M33	21.38	58.72	**14.79**	75.28	16.57	23.92	22.17	16.15	23.29	**14.23**	13.54	15.39	16.00	15.39	**10.23**
M34	23.31	40.84	**6.87**	131.16	39.93	**11.83**	33.50	18.36	35.75	23.31	22.78	24.88	**11.60**	32.50	18.63
Avg.	**42.52**	288.79	376.54	660.42	40.16	42.12	49.28	39.35	50.05	**33.46**	31.96	30.78	30.79	37.50	**26.69**

TABLE IV: Runtime for finding state setters in seconds. Bold indicates the best result within each group.

	Baseline				Ablation Study of Our Options						Our Variants				
ID	iFixFlakies's Default Variant	RankF$_L$	RankF$_O$	OBO	Split-Hierarchical	Split-NLP	Split-LLM	Split-Historical	Pick-Runtime	Run-One	Split-Half	Split-Hierarchical	Split-NLP	Split-LLM	Split-Historical
M1	23.80	33.82	**5.50**	103.05	27.36	26.65	29.99	**15.72**	32.42	20.48	20.83	19.98	23.64	26.55	**13.03**
M3	**155.01**	265.25	1140.50	737.94	130.89	157.92	215.87	162.92	126.87	**124.46**	104.75	**98.85**	107.50	171.81	104.21
M9	356.26	46.06	**1.90**	60.74	**252.71**	288.53	354.29	313.64	337.66	256.85	224.50	**153.57**	159.13	225.72	205.08
M13	**20.53**	168.94	68.40	1062.41	**11.54**	22.94	27.24	18.75	15.03	14.77	9.80	9.24	10.97	13.48	**8.02**
M16	29.32	286.66	**3.70**	489.53	31.23	20.68	35.74	**19.47**	37.60	24.80	23.70	25.61	16.15	30.90	**14.97**
M17	**25.69**	32.01	70.00	45.51	**7.88**	31.57	25.83	24.73	27.95	17.87	**17.92**	20.63	20.89	18.95	17.94
M24	215.53	2434.24	**13.30**	3284.96	278.81	**113.72**	281.85	119.78	349.97	215.53	215.10	188.67	**113.42**	281.32	119.37
M25	327.96	37.72	**13.30**	1146.39	429.14	**167.73**	428.19	179.05	542.30	327.96	327.69	297.95	**167.58**	427.78	178.91
M29	**121.17**	342.20	446.90	1538.35	131.12	**45.78**	140.93	119.88	154.20	120.66	109.36	115.06	**39.90**	120.46	111.03
M31	9.31	**3.25**	7.40	3.35	**5.50**	8.87	10.14	8.86	6.31	7.58	5.69	5.94	6.31	6.97	**5.65**
M33	19.39	**5.36**	26.80	47.16	**10.18**	23.45	20.04	17.82	19.29	14.18	12.49	**11.16**	16.92	14.65	11.85
Avg.	**81.30**	206.24	158.98	485.49	78.20	**62.91**	98.69	65.13	98.51	69.99	65.21	58.96	**46.81**	83.04	49.92

LLM-based scores are not yet informative enough to guide splitting efficiently. Wilcoxon signed-rank tests with effect sizes confirm that these improvements over the best baseline are statistically significant across all three OD-relevant test types. Detailed results of variants without inference time and statistical tests are on our website [35].

RQ2: *We find that the three best splitting options (Split-Hierarchical, Split-NLP, and Split-Historical) exhibit comparable speedups when used with our Run-One and Pick-Runtime options. Developers should use a Split-Historical variant for polluters, Split-NLP variant for state-setters, and Split-Hierarchical variant for cleaners.*

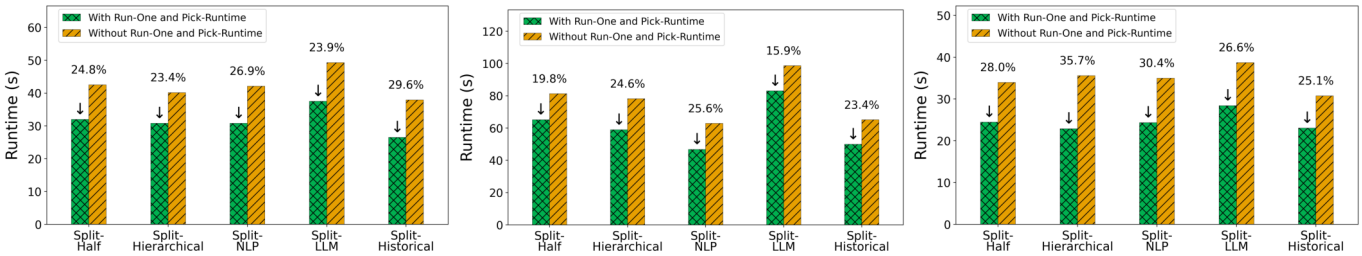
C. Case Studies of RQ Results

Analysis of why Split-NLP variant performs better than Split-Hierarchical variant for some cases. Our Split-NLP and Split-Hierarchical variants perform similarly for polluters and cleaners, but Split-NLP is substantially better for finding a few subjects' state-setters. From Figure 5, we see an OD test and its corresponding OD-relevant test for M25. Both tests belong to the same class and share many identical tokens (e.g., `webDriver`, `"/html"`, and `element.getText()`), which results in a high TF-IDF score of 0.69. This high score resulted in Split-NLP placing the state-setter alone in one subsequence and all remaining tests in the other. As the test runtime of the state-setter is less than the combined runtime of the other tests, using the Pick-Runtime option picks the correct subsequence and detects the OD-relevant test in one iteration. Although the brittle and state-setter belong to the same class, Split-Hierarchical splits the total classes in half and ultimately required seven iterations, as the subject has 13 classes and 2-6 tests each. We find that variants using Split-Hierarchical can perform poorly if the number of classes is large and each class contains only a few tests, as the initial class-level search adds extra iterations without substantially reducing the search space. Developers should avoid using the Split-Hierarchical option when their project contains a very large number of test classes, with only a small number of tests per class ($\sim$2–6), as the class structure provides little benefit in guiding the search in such cases.

Analysis of RankF$_O$'s Performance. To better understand

TABLE V: Runtime for finding cleaners in seconds. Bold indicates the best result within each group.

	Baseline				Ablation Study of Our Options						Our Variants				
ID	iFixFlakies's Default Variant	RankF$_L$	RankF$_O$	OBO	Split-Hierarchical	Split-NLP	Split-LLM	Split-Historical	Pick-Runtime	Run-One	Split-Half	Split-Hierarchical	Split-NLP	Split-LLM	Split-Historical
M1	**26.56**	85.99	32.81	65.91	25.19	26.68	24.82	23.74	24.98	**18.88**	19.21	17.66	18.71	20.25	**16.98**
M2	**14.44**	47.54	36.05	64.30	11.20	11.53	9.72	13.46	8.95	**7.82**	7.61	6.93	**6.55**	6.75	6.92
M4	10.43	43.20	**1.92**	50.74	13.77	12.86	21.73	9.38	15.84	**9.05**	7.65	7.10	9.73	8.65	**6.67**
M7	10.52	**9.34**	9.36	30.43	7.99	10.66	13.99	10.52	11.00	**6.92**	6.48	**6.29**	7.45	6.49	6.96
M8	256.32	4.12	**2.45**	17.32	332.16	323.72	439.87	**237.83**	417.19	256.32	197.37	192.67	214.55	213.02	**186.99**
M9	10.59	8.53	**4.46**	15.33	10.49	8.36	4.75	**4.24**	5.30	5.28	5.20	4.50	4.42	3.82	**2.07**
M10	560.25	2097.42	**204.62**	1577.06	729.34	680.11	862.06	544.38	825.94	**503.86**	397.21	**137.53**	389.91	404.83	306.72
M12	2.58	**1.34**	1.44	10.80	2.97	3.14	2.58	2.22	2.60	**1.83**	2.16	2.16	2.40	**1.80**	1.80
M14	13.41	**5.50**	6.49	30.87	15.88	16.41	19.95	12.64	19.18	**11.34**	11.13	11.53	13.63	11.97	**10.79**
M15	490.88	585.83	**123.93**	2083.24	628.55	536.96	608.31	451.58	584.16	**449.62**	**450.23**	476.18	466.73	463.98	485.16
M18	**78.73**	90.48	108.11	313.88	62.52	56.86	**37.53**	38.23	42.97	40.20	40.14	37.78	29.64	33.46	**20.00**
M19	12.75	64.61	**2.84**	131.36	12.75	13.24	19.36	11.49	13.97	**8.89**	8.84	**8.80**	9.93	8.83	9.31
M20	**20.59**	70.49	21.36	825.85	14.81	19.60	36.85	18.85	12.32	**11.51**	11.61	11.02	**10.05**	10.96	10.57
M21	**33.76**	216.51	336.43	480.08	35.82	34.46	51.84	34.13	34.24	**24.31**	21.34	**21.25**	21.39	21.87	21.32
M22	**233.63**	5948.81	476.70	12953.25	183.94	209.91	171.92	210.97	132.07	**127.17**	124.26	119.41	**114.97**	121.25	115.63
M26	9.66	6.39	**4.50**	13.50	10.06	13.13	**6.56**	9.66	8.73	6.84	8.31	9.08	**5.27**	6.71	8.31
M27	7.37	3.55	**2.00**	21.03	9.05	9.18	12.49	**6.43**	10.53	7.04	7.32	6.99	7.54	7.95	**6.67**
Avg.	**33.94**	125.75	45.62	242.67	35.59	34.96	38.69	30.70	35.10	**25.75**	24.44	**22.88**	24.32	28.39	22.99



(a) Polluter (b) State Setter (c) Cleaner

Fig. 4: Runtime comparison of each split option with and without Pick-Runtime and Run-One. Lower runtime is better.

```
// Brittle test
public void shouldBeADifferentWebClient() {
  this.webDriver.get("/html");
  WebElement element = this.webDriver.findElement(By
    .tagName("body"));
  assertThat(element.getText()).isEqualTo("Hello");
  try {
    ReflectionTestUtils.invokeMethod(
      previousWebDriver, "getCurrentWindow");
    fail("Did not call quit()");
  } catch (NoSuchWindowException ex) {
    // Expected behavior
  }
  assertThat(previousWebDriver).isNotNull().
    isNotSameAs(this.webDriver);
}
// State-setter test
public void shouldAutoConfigureWebClient() {
  this.webDriver.get("/html");
  WebElement element = this.webDriver.findElement(By
    .tagName("body"));
  assertThat(element.getText()).isEqualTo("Hello");
  WebMvcTestWebDriverIntegrationTests.
    previousWebDriver = this.webDriver;
}
```

Fig. 5: Example of a brittle test and its state-setter test.

how our variants outperform state-of-the-art RankF$_O$, we examine one of our subjects in detail. For M22, there are an average of 812.2 tests before the victim. The polluter appears before the victim eight times without a cleaner, but it also appears before the victim seven times with a cleaner in between. In the remaining five test-orders, the polluter appears after the victim. For each time the polluter appears before the victim but a cleaner is also in-between the polluter and the victim, RankF$_O$ will lower the polluter's score, so the true polluter ends up with a low score. Table III shows that our variants reach up to 92.91% speedup over RankF$_O$, showing how poor scoring from historical information can affect RankF$_O$. However, we construct variants that can achieve better results without even requiring this historical information.

Multi-test OD-relevant tests. Among our six proposed options, only Run-One is not applicable to multi-test OD-relevant tests due to its underlying assumption (Section III-D). Although prior work [31] found that the majority of OD-relevant tests are single tests, they identified one known case where a victim fails only when multiple polluters run before it. For this case, we compare the default variant with our best polluter variant that does not require historical information (combining Split-Hierarchical and Pick-Runtime). In this case, this variant still performs better, achieving a 9.73% speedup over the default variant. In summary, when we apply our variant to the one known case of a multi-test polluter, we find that our proposed variant does outperform the default variant.

VI. Threats to Validity

Our results may not generalize to all projects, as we rely on a previous dataset from Wei et al. [34] and Rahman et al. [30], which focuses on OD tests from popular open-source Java projects. The specific test environments, build systems, and configurations in this dataset may not reflect broader contexts, potentially limiting generalizability, although these datasets were constructed using tests taken from large and popular

open-source projects, providing some level of representativeness. Our use of simulation to evaluate the runtime of different approaches may also limit reproducibility. Reusing the same dataset simulating the runtime of approaches does enable us to more easily compare with prior work [30]. To mitigate the threat of simulations, we validate the simulated runtimes of the default variant and our best polluter variant (combining Split-Historical, Pick-Runtime, and Run-One) on one polluter per module. Actual executions show that our best variant is 41.72% faster than the default variant, consistent with the simulated speedup of 36.11% for the same polluters. Also, our NLP and LLM options rely on heuristics, which assume a correlation between textual similarity and test relevance, an assumption that may not hold across different subjects and thus warrants further validation. Finally, our comparisons mainly rely on descriptive statistics, e.g., average runtimes, which may be affected by runtime outliers and differences in the number of OD tests across modules. We mitigate this threat by reporting weighted averages and complementing them with Wilcoxon signed-rank tests and effect sizes.

VII. Related Work

Detecting OD tests. Luo et al. first conducted an empirical study on flaky tests in open-source projects [8]. Tests can be flaky for many reasons and flakiness due to test order is found to be one of the most prevalent reasons. Knowing the type of flaky test is essential for helping developers debug and reproduce flaky test failures [43]–[45]. Also, knowing the flaky test's type enables the use of targeted repair approaches, such as iFixFlakies [31], which is specifically designed to fix OD tests. To detect OD tests, prior research has employed methods such as generating random test orders [7], [33], [46], [47], analyzing code changes [48], systematically creating test orders [34], [49], and conducting code analysis [50], [51].

Detecting OD-relevant tests. Detecting OD tests does not identify the corresponding OD-relevant test, i.e., the test responsible for triggering order dependencies, namely polluter, state-setter, or cleaner. Detecting the OD-relevant test is important for repairing OD tests [31], [52]. iFixFlakies finds OD-relevant tests using delta-debugging [31], which searches for OD-relevant tests by dividing tests that come before a victim/brittle in a failing/passing test-order, respectively, and finding the smallest set of tests that cause the failure/passing of a test. Although this approach is effective at detecting OD-relevant tests when an OD test depends on multiple tests, this iterative method requires rerunning tests many times, which can be expensive. Our proposed options reduce redundant executions and overall runtime to detect OD-relevant tests, whether they are single tests or consist of multiple tests.

Since prior work [31] found the vast majority of OD tests have only a single OD-relevant test that changes its outcome when run beforehand, Lam et al. [42] later leveraged this finding to propose running each test *one-by-one (OBO)* before an OD test to find all OD-relevant tests for that OD test. More recently, Rahman et al. proposed approaches that rank other tests based on their likelihood of being OD-relevant tests, leveraging large language models (RankF$_L$) and historical run information (RankF$_O$), running tests one-by-one in order of these rankings as a means to speed up the search further [30]. In this paper, we compare against both approaches.

Delta-debugging and its variants. Our proposed improvements are generally centered around better adapting the traditional delta-debugging approach to more efficiently search for OD-relevant tests. Delta-debugging has been a foundational approach to minimize failure-inducing inputs proposed by Zeller and Hildebrandt [36]. It systematically partitions inputs, and tests each subset to find the smallest failing input. While effective, delta-debugging can be inefficient when dealing with structured or hierarchical inputs. Probabilistic Delta Debugging (ProbDD) [53] improves upon delta-debugging by estimating the likelihood of each input fragment causing failure and prioritizing those with higher probability. However, in our setting, the Run-One option immediately discards the non-failing part once a failure is observed, forcing its probability to zero, and preventing ProbDD from updating its model. Thus, ProbDD offers no additional benefit in our setting. Other work, such as Hierarchical Delta Debugging (HDD) [54] and Perses [55], focuses on tree-structured or syntax-guided reduction, which does not directly apply to our linear test-order domain. More recently, weight-based delta-debugging [56] has been explored; our TF–IDF similarity weighting follows a similar intuition, but uses a relevance-aware signal more suitable for OD-relevant test detection.

VIII. Conclusion

In this paper, we revisit the problem of efficiently detecting OD-relevant tests for order-dependent flaky tests, a major source of unreliable test outcomes in regression testing. Building on the foundation of iFixFlakies, we identified three types of options that can reduce unnecessary executions and improve search time: (i) split options that create better sequences using hierarchical information, NLP, LLM, and historical information, (ii) a Pick-Runtime option that prioritizes lower-runtime sequences, and (iii) a Run-One option that skips unnecessary sequence runs.

Our evaluation on 155 OD tests from 34 test suites shows that our improvements independently contribute to faster detection of OD-relevant tests, with Run-One providing the most consistent gains across all types of OD-relevant tests. Splitting options based on hierarchical structure and textual similarity further improve detection efficiency over the default splitting option used by iFixFlakies. When combining the options, our best variant achieves up to 42.42% faster detection than iFixFlakies, as well as up to 90.76% and 92.91% faster detection than prior state-of-the-art approaches RankF$_L$ and RankF$_O$, respectively.

Acknowledgements

We thank Dana Richards, Didier Ishimwe, and Talank Baral for comments on this work. We acknowledge support from NSF grants nos. CCF-2145774 and CCF-2338287, and the Jarmon Innovation Fund.

REFERENCES

[1] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming Google-scale continuous testing," in *ICSE SEIP: 39th International Conference on Software Engineering, Software Engineering in Practice Track*, 2017.

[2] J. Micco, "The state of continuous integration testing at Google," in *ICST: 10th International Conference on Software Testing, Verification and Validation*, 2017.

[3] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in *ICSE SEIP: 41st International Conference on Software Engineering, Software Engineering in Practice Track*, 2019.

[4] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *ICSE: 37th International Conference on Software Engineering*, 2015.

[5] A. Shi, S. Thummalapenta, S. K. Lahiri, N. Bjorner, and J. Czerwonka, "Optimizing test placement for module-level regression testing," in *ICSE: 39th International Conference on Software Engineering*, 2017.

[6] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of Java projects using continuous integration," in *ESEC/FSE: 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[7] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *ISSTA: 23rd International Symposium on Software Testing and Analysis*, 2014.

[8] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *FSE: 22nd Symposium on the Foundations of Software Engineering*, 2014.

[9] E. Kowalczyk, K. Nair, Z. Gao, L. Silberstein, T. Long, and A. Memon, "Modeling and ranking flaky tests at Apple," in *ICSE SEIP: 42nd International Conference on Software Engineering, Software Engineering in Practice Track*, 2020.

[10] "Expedia – fixing flaky time based unit tests," 2026, https://medium.com/expedia-group-tech/fixing-flaky-time-based-unit-tests-176accf5096e.

[11] "A machine learning solution for detecting and mitigating flaky tests," 2026, https://eng.fitbit.com/a-machine-learning-solution-for-detecting-and-mitigating-flaky-tests.

[12] "Flakiness dashboard HOWTO," 2026, http://www.chromium.org/developers/testing/flakiness-dashboard.

[13] J. Micco, "Continuous integration at Google scale," 2026, https://www.slideshare.net/JohnMicco1/2016-0425-continuous-integration-at-google-scale.

[14] C. Ziftci and J. Reardon, "Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at Google scale," in *ICSE SEIP: 39th International Conference on Software Engineering, Software Engineering in Practice Track*, 2017.

[15] E. Wendelin, "Introducing flaky test mitigation tools," 2026, https://blog.gradle.org/gradle-flaky-test-retry-plugin.

[16] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? Automatic cause analysis for test alarms in system and integration testing," in *ICSE: 39th International Conference on Software Engineering*, 2017.

[17] B. Harry, "How we approach testing VSTS to enable continuous delivery," 2026, https://blogs.msdn.microsoft.com/bharry/2017/06/28/testing-in-a-cloud-delivery-cadence.

[18] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *ISSTA: 28th International Symposium on Software Testing and Analysis*, 2019.

[19] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *ICSE: 42nd International Conference on Software Engineering*, 2020.

[20] "Test verification," 2026, https://developer.mozilla.org/en-US/docs/Mozilla/QA/Test_Verification.

[21] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *ESEC/FSE: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[22] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds," in *ESEC/FSE: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[23] "Netflix automation talks - Test automation at scale," 2026, https://youtu.be/FrBN94gUn_I?t=764.

[24] "Flaky tests (and how to avoid them)," 2026, https://engineering.salesforce.com/flaky-tests-and-how-to-avoid-them-25b84b756f60.

[25] P. Gochenour and R. Andre, "How to deal with flaky Java tests," 2026, https://wiki.saucelabs.com/display/DOCS/How+to+Deal+with+Flaky+Java+Tests.

[26] P. Sudarshan, "No more flaky tests on the Go team," 2026, http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.

[27] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *TOSEM*, 2021.

[28] S. Rahman, S. Dutta, and A. Shi, "Understanding and improving flaky test classification," in *OOPSLA: Object-Oriented Programming Systems, Languages, and Applications*, 2025.

[29] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *ISSTA: 29th International Symposium on Software Testing and Analysis*, 2020.

[30] S. Rahman, B. N. Chanumolu, S. Rafi, A. Shi, and W. Lam, "Ranking relevant tests for order-dependent flaky tests," in *ICSE: 47th International Conference on Software Engineering*, 2025.

[31] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE: 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[32] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *SIGSOFT Software Engineering Notes*, 1999.

[33] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "iDFlakies: A framework for detecting and partially classifying flaky tests," in *ICST: 12th International Conference on Software Testing, Verification and Validation*, 2019.

[34] A. Wei, P. Yi, T. Xie, D. Marinov, and W. Lam, "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests," in *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*, 2021.

[35] "Optimizing search for tests relevant to order-dependent flaky tests," 2026, https://sites.google.com/view/optimizing-od-search/home.

[36] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *TSE*, 2002.

[37] W. Lam, S. Winter, A. Astorga, V. Stodden, and D. Marinov, "Understanding reproducibility and characteristics of flaky tests through test reruns in Java projects," in *ISSRE: 31st International Symposium on Software Reliability Engineering*, 2020.

[38] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang *et al.*, "Big bird: Transformers for longer sequences," *Advances in neural information processing systems*, 2020.

[39] "Java Marine API," 2026, https://github.com/ktuukkan/marine-api.

[40] A. Singhal, "Modern information retrieval: A brief overview," *IEEE Data Engineering Bulletin*, 2001.

[41] A. Zeller, "Yesterday, my program worked. Today, it does not. Why?" in *ESEC/FSE: 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999.

[42] W. Lam, S. Winter, A. Wei, T. Xie, D. Marinov, and J. Bell, "A large-scale longitudinal study of flaky tests," in *OOPSLA: Object-Oriented Programming Systems, Languages, and Applications*, 2020.

[43] S. Rahman, A. Massey, W. Lam, A. Shi, and J. Bell, "Automatically reproducing timing-dependent flaky-test failures," in *ICST: 17th International Conference on Software Testing, Verification and Validation*, 2024.

[44] A. Akli, G. Haben, S. Habchi, M. Papadakis, and Y. Le Traon, "Flaky-Cat: Predicting flaky tests categories using few-shot learning," in *AST: 4th International Conference on Automation of Software Test*, 2023.

[45] S. Rahman, A. Baz, S. Misailovic, and A. Shi, "Quantizing large-language models for predicting flaky tests," in *ICST: 17th International Conference on Software Testing, Verification and Validation*, 2024.

[46] R. Wang, Y. Chen, and W. Lam, "iPFlakies: A framework for detecting and fixing Python order-dependent flaky tests," in *ICSE Demo: 44th International Conference on Software Engineering, Demonstrations Track*, 2022.

[47] M. Gruber, S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of flaky tests in Python," in *ICST: 14th International Conference on Software Testing, Verification and Validation*, 2021.

[48] C. Li and A. Shi, "Evolution-aware detection of order-dependent flaky tests," in *ISSTA: 31st International Symposium on Software Testing and Analysis*, 2022.

[49] C. Li, M. M. Khosravi, W. Lam, and A. Shi, "Systematically producing test-orders to detect order-dependent flaky tests," in *ISSTA: 32nd International Symposium on Software Testing and Analysis*, 2023.

[50] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe Java test acceleration," in *ESEC/FSE: 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[51] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *ICST: 11th International Conference on Software Testing, Verification and Validation*, 2018.

[52] C. Li, C. Zhu, W. Wang, and A. Shi, "Repairing order-dependent flaky tests via test generation," in *ICSE: 44th International Conference on Software Engineering*, 2022.

[53] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang, "Probabilistic delta debugging," in *ESEC/FSE: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[54] G. Misherghi and Z. Su, "HDD: Hierarchical delta debugging," in *ICSE: 28th International Conference on Software Engineering*, 2006.

[55] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *ICSE: 40th International Conference on Software Engineering*, 2018.

[56] X. Zhou, Z. Xu, M. Zhang, Y. Tian, and C. Sun, "WDD: Weighted delta debugging," in *ICSE: 47th International Conference on Software Engineering*, 2025.