Ranking Relevant Tests for Order-Dependent Flaky Tests

Shanto Rahman[1], Bala Naren Chanumolu[2], Suzzana Rafi[2], August Shi[1], Wing Lam[2]
[1] The University of Texas at Austin, USA
{shanto.rahman,august}@utexas.edu
[2] George Mason University, USA
{bchanumo,srafi,winglam}@gmu.edu

***Abstract*—One major challenge of regression testing are *flaky tests*, i.e., tests that may pass in one run but fail in another run for the same version of code. One prominent category of flaky tests is order-dependent (OD) flaky tests, which can pass or fail depending on the order in which the tests are run. To help developers debug and fix OD tests, prior work attempts to automatically find *OD-relevant tests*, which are tests that determine whether an OD test passes or fails, depending on whether the OD-relevant tests run before or after the OD test. Prior work found OD-relevant tests by running different tests before the OD test, without considering each test's likelihood of being OD-relevant tests.**

We propose *RankF* to rank tests in order of likelihood of being OD-relevant tests, finding the first OD-relevant test for a given OD test more quickly. We propose two ranking approaches, each requiring different information. Our first approach, *RankF*$_L$, relies on training a large-language model to analyze test code. Our second approach, *RankF*$_O$, relies on analyzing prior test-order execution information. We evaluate our approaches on 155 OD tests across 24 open-source projects. We compare RankF against baselines from prior work, where we find that RankF finds the first OD-relevant test for an OD test faster than the best baseline; depending on the type of OD-relevant test, RankF takes 9.4 to 14.1 seconds on median, compared to the baseline's 34.2 to 118.5 seconds on median.

I. Introduction

Software developers rely on regression testing, the process of rerunning tests after every code change, to check that code changes do not introduce faults [1]. However, regression testing suffers from the presence of *flaky tests*, which are tests that can pass and fail for the same version of code [2]. A flaky-test failure after a change can mislead developers into thinking they introduced a fault in their changes when instead the test could have failed even without those changes. Several studies [2]–[4] found various reasons for flaky tests, with one prominent reason being test-order dependency. *Order-dependent flaky tests* (OD tests) are tests that can pass or fail due to the order in which the tests are run. OD test failures occur when tests depend on some global state, such as static variables, the file system, databases, network, etc. [5], that other tests modify. Developers cannot always enforce a specific test-order in which the OD tests pass, especially if they rely on regression testing techniques, such as test prioritization, selection, and parallelization, which purposefully reorder or run subsets of tests to speed up regression testing [1, 6].

To repair OD tests, recent prior work [7, 8] relies on *OD-relevant tests*, i.e., the tests that modify the global state that the OD test depends on. Finding OD-relevant tests has three main purposes: (1) enable the use of automatic repair techniques [7, 9], (2) enable the use of mitigation techniques [6], and (3) help developers debug and reproduce OD-test failures. Shi et al. [7] proposed the use of delta-debugging [10] to find OD-relevant tests. Follow-up work [11]–[13] proposed finding OD-relevant tests by running pairs of tests one-by-one (OBO), where each test is paired before the OD test. Both approaches can be costly as they rerun tests many times.

We propose *RankF*, a means to rank tests based on their likelihood to be OD-relevant tests for a given OD test. The goal is to rank true OD-relevant tests higher than non-OD-relevant tests, so fewer tests need to be run before finding an OD-relevant test. We propose two different approaches.

Our first approach, *RankF*$_L$, analyzes only the test-method body, i.e., the code directly under a test method, to compute likelihood. RankF$_L$ fine-tunes a pre-trained large-language model (LLM), BigBird [14], to take as input the test-method bodies of a potential OD-relevant test and the OD test to then output a score indicating the likelihood of the potential test being an actual OD-relevant test for the OD test. We train the model on a large dataset of known OD tests and their corresponding OD-relevant tests [12, 15]. While prior work used LLMs to predict which tests are flaky [16], none has explored using LLMs to predict OD-relevant tests, which requires analyzing the relationship between two or more tests.

Our second approach, *RankF*$_O$, analyzes the test results of previous test-order execution information to compute the likelihood of each test being OD-relevant tests. Developers may already have several different test-orders that they ran before, either as part of regression testing, where tests were run in different test-orders in the past [17], or as part of an OD test detection process that runs tests in different test-orders to search for existing OD tests [5, 18]–[20]. RankF$_O$ uses several heuristics to compute likelihood scores for each test based on how often each one appears before the OD test in the passing test-orders and failing test-orders as well as how far away each test is from the OD test in these test-orders.

Both RankF$_L$ and RankF$_O$ have the same goal of ranking tests based on likelihood of being OD-relevant tests. However, the approaches represent different use scenarios. RankF$_O$ is

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00178

useful if developers already have some results from different test-orders. If developers do not have such information, they can use RankF$_L$, which operates just on test code.

We evaluate RankF$_L$ and RankF$_O$ on a dataset of known OD tests from prior work [12]. This dataset contains OD tests along with the corresponding OD-relevant tests for each OD test. In total, there are 155 OD tests from 34 modules across 24 open-source projects in this dataset, and we use it to measure how highly each approach ranks a true OD-relevant test. To evaluate efficiency in an actual use scenario, we measure how long the approaches take to both rank tests and then run them in a ranked order to confirm finding a true OD-relevant test. We compare against prior baselines for finding OD-relevant tests, OBO [11]–[13], and delta-debugging [7]. For the majority of our subjects, either RankF$_L$ or RankF$_O$ is faster than all of the baselines at finding OD-relevant tests – achieving a range of median times, depending on type of OD-relevant test, from 9.4 to 14.1 seconds, compared to the 34.2 to 118.5 seconds achieved by the baselines.

We make the following contributions in this paper:

- We propose two new approaches for ranking tests based on their likelihood of being OD-relevant tests for an OD test: (1) RankF$_L$ uses an LLM to rank tests based on test code, and (2) RankF$_O$ analyzes test results from different test-orders to rank tests based on their positions relative to a given OD test in those test-orders. Developers can find OD-relevant tests faster using these rankings.
- Our evaluation shows our approaches are faster at finding OD-relevant tests than prior baselines. Our implementations, scripts, and data are publicly available [21].

II. Background

An *order-dependent flaky test* (OD test) is a flaky test whose outcome depends on the test-order in which it is run [2,5,18]. We refer to a test-order in which the OD test passes as a *passing test-order* and a test-order in which it fails as a *failing test-order*. The reason that an OD test fails in a failing test-order is because it shares some global state with some other test, where that other test either (1) runs before the OD test in the failing test-order and "pollutes" the shared state, leading to the OD test to fail when run afterwards, or (2) runs after the OD test in the failing test-order and is not setting up a shared state required by the OD test to pass. Shi et al. provided names for these different types of OD tests as well as their corresponding OD-relevant tests [7]. An OD test that passes when run on its own but fails when run after some other test is a *victim*; the test that "pollutes" the shared state in the failing test-order of a victim is a *polluter*. An OD test that fails when run on its own is a *brittle*. A brittle fails when run on its own because it relies on some other test to set up its initial state; this other test is a *state-setter*, and a brittle will pass when run after it. Finally, Shi et al. found that, for the case of victims and polluters, there are other tests in the test suite that can actually reset the shared state when these other tests are run after the polluter but before the victim, resulting in the victim passing. These other tests are referred to as *cleaners* for a polluter-victim pair. Shi et al. relied on these cleaners to automatically repair victims [7]. We collectively refer to polluters, state-setters, and cleaners as *OD-relevant tests*.

Shi et al. previously proposed iFixFlakies [7], which can find OD-relevant tests. Given a victim/brittle and a failing/passing test-order for the test, respectively, iFixFlakies identifies a polluter/state-setter for the OD test, respectively, via delta-debugging [10] the tests that come before the OD test in the test-order. (An OD test may need more than one other test to run before the OD test for it to fail/pass, but Shi et al. found this scenario to be very rare [7].) Lam et al. later proposed detecting OD-relevant tests in a simpler way by running each test one-by-one with the OD test to check whether the OD test's outcome changes [11]. This one-by-one search is also effective for finding cleaners for a given polluter-victim pair, by running each test in-between the polluter and victim; iFixFlakies also uses this one-by-one strategy to find cleaners [7]. Both delta-debugging and the one-by-one search depend on the given test-orders, potentially running for a long time depending on where the OD-relevant tests are within the test-orders. Our evaluation shows these approaches often run many (irrelevant) tests before finding any OD-relevant test.

A. Example

Figures 1 and 2 illustrate simplified versions of an example brittle OD test, and its state-setter OD-relevant test, respectively. The brittle, `testDubboProtocolWithMina`, and state-setter, `testRpcFilter`, are from the `apache/dubbo` project [22], which has over 40000 stars on GitHub and is an easy-to-use Web and RPC framework that provides multiple language implementations for different services (e.g., communication, security).

The brittle fails because it explicitly specifies the server is `mina` in its URL at Line 3, which tells the code to use the `mina` server for communication. Both the brittle and state-setter have a `protocol` (sets up a `DemoService` to be accessed over the network) and `proxy` (handles calls to the network) variable, which are separate class instances. Using these variables, Line 3 exports the service and initiates the server (based on the URL and configuration). We find that this method call performs various server initialization logic, including for the `mina` server, when no server information is specified in the URL. However, as Line 3 contains the server information in `addParameter(Constants.SERVER_KEY, "mina")`, the code skips the `mina` server initialization and directly tries to use the server. If the `mina` server is not already initiated, skipping initialization results in an exception.

When state-setter `testRpcFilter` runs before this brittle, we find that `protocol.export` is called without specifying any server at Line 12, which initializes various servers, including `mina`. Once this state-setter starts the `mina` server, the brittle passes when run afterwards.

Finding this state-setter among all tests can be time-consuming, as there are 65 total tests in the test suite. For example, we can use the one-by-one strategy [11]–[13] by running each test before the brittle to see which one makes

```
1  public void testDubboProtocolWithMina() {
2   DemoService service = new DemoServiceImpl();
3   protocol.export(proxy.getInvoker(service, ...,
      URL.valueOf("dubbo://127.0.0.1:9010/").
      addParameter(Constants.SERVER_KEY, "mina")));
4   ...
5  }
```

Fig. 1: Example brittle from the `DubboProtocolTest` class in the module `dubbo-rpc/dubbo-rpc-dubbo` of the `apache/dubbo` project [22].

```
10 public void testRpcFilter() {
11  DemoService service = new DemoServiceImpl();
12  protocol.export(proxy.getInvoker(service, ...,
      URL.valueOf("dubbo://127.0.0.1:9010/")));
13  ...
14 }
```

Fig. 2: Example of a state-setter for the brittle in Figure 1. `testRpcFilter` is from the `RpcFilterTest` class and the same module and project as the brittle.

the brittle pass. In our experiments, we can find the state-setter after running 15.5 tests (averaged from using the tests positioned before the brittle in 10 different passing test-orders). On the other hand, if we first rank the tests based on likelihood of being OD-relevant tests, such as by leveraging a LLM (Section III-A), we can find the state-setter after running only 6.5 tests, on average. If we instead rank tests based on analyzing test outcomes from a large number of test-orders (Section III-B), we can find the state-setter after running only 1.0 test, on average. Developers can spend less time to find OD-relevant tests by first ranking tests and then running them in the ranked order, taking, on average, 23.5 and 1.9 seconds using the LLM and test-order-analysis approach, respectively. Meanwhile, the one-by-one baseline takes 60.9 seconds, on average, while the delta-debugging baseline proposed by Shi et al. [7] takes 419.0 seconds, on average.

III. Ranking OD-Relevant Tests

Given a known OD test, the goal is to rank the set of other tests in terms of their likelihood of being an OD-relevant test for the given OD test. First, we determine the type of OD test, which according to Shi et al. [7], we can determine whether an OD test is a victim or brittle by running it by itself (Section II). In the case of a victim, we search for a polluter. For a brittle, we search for a state-setter. For a victim and a found polluter for that victim, we search for a cleaner corresponding to the given victim/polluter pair.

We propose two different approaches, RankF$_L$ and RankF$_O$, for computing the likelihood of tests as OD-relevant tests, each one relying on different information. For a given OD test, each approach ranks all other tests as candidate OD-relevant tests, giving each one a score representing its likelihood. Then, we iterate through each candidate OD-relevant test according to the ranking to confirm whether the test is a true OD-relevant test for the given OD test. To confirm, we run just the candidate test along with the OD test. If the OD test is a victim, we run the candidate before the victim to see whether the victim fails. If the OD test is a brittle, we run the candidate before the brittle to see whether the brittle passes. In the case of searching for cleaners, we require both a victim and polluter, and we run the candidate in-between the polluter and victim to see whether the victim passes, which normally fails when run right after the polluter. The search stops once the process confirms a candidate test to be a true OD-relevant test. Prior work found that finding just one OD-relevant test is often enough to debug or repair an OD test [7]. This search becomes more efficient when the ranking is better (i.e., the real OD-relevant tests are ranked earlier), as fewer tests need to be run before the confirmation of a true OD-relevant test. Note that an OD-relevant test can also be an OD test, but this situation is rare – only one such reported case in prior work [7]. If there is a test that is both a victim and polluter, then we will find the polluters for this test as a victim, then separately find it later as a polluter for another victim. (We carry out a similar process for brittles and state-setters.)

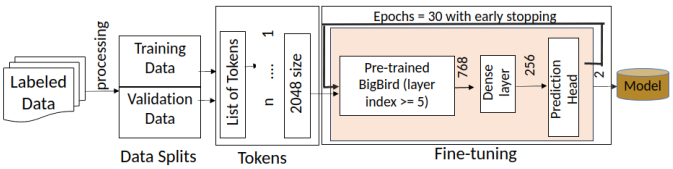

Fig. 3: Overview architectural diagram of RankF$_L$ training.

A. *RankF$_L$*

RankF$_L$ leverages a large language model (LLM) to compute likelihood scores for OD-relevant tests. Given an OD test, a user can use RankF$_L$ to compute a likelihood score for each of the other tests to be an OD-relevant test. We build separate models for scoring polluters for victims, state-setters for brittles, and cleaners for victim/polluter pairs.

To train a model, we use a training dataset of labeled tuples of tests. Each tuple consists of either a victim, brittle, or victim/polluter pair, depending on the type of OD test the model is specialized for, along with some other tests in the test suite. The tuple has a positive label if the other test is a true OD-relevant test for the victim, brittle, or victim/polluter pair, and negative otherwise. Figure 3 shows the training process for creating a RankF$_L$ model.

1) BigBird: We use BigBird-roberta-base [23] as the pre-trained LLM that we fine-tune and build the model from. BigBird was constructed based on the Transformer neural architecture [24], with a focus on understanding long sequences and can process a large number of tokens, especially in comparison to other LLMs [25]. Processing many tokens is especially important for our problem, as we need to analyze the code from multiple tests, as opposed to just a single test that is used for other similar LLM-based tasks [16, 26, 27].

2) Extracting Features from Test Code: For a given OD test (or polluter-victim pair), we obtain all of its candidate tests and use each one to form a tuple with the given OD test. For each candidate and OD test, we use srcML [28] to extract

the signature of the test method and the test method body. We also eliminate any code comments from the method body.

We provide each tuple (consisting of the signature and body of a candidate test and OD test/polluter-victim pair) with a positive or negative label to the model for training. We utilize BigBird's built-in tokenizer that uses SentencePiece with Byte-Pair Encoding (BPE) [29] to create a sequence of tokens representing this aggregated code. We also utilize `batch_encode_plus` [30] from the tokenizer to encode the token sequences into numerical representations (token ID and attention mask) in a batch, outputting a vector of numbers, where each number represents a token.

While BigBird can process at most 4096 tokens, setting it to always process this many tokens can introduce unnecessary token padding and memory usage that negatively impacts the model's performance. In our evaluation dataset, over 99% of the tuples have fewer than 2048 tokens. Hence, we set the maximum token length to 2048. Similar to prior work, if the number of tokens is over this limit, we truncate [16, 26].

3) Fine-Tuning Models: We fine-tune the BigBird model to distinguish between positive and negative-labeled tuples from the dataset. As BigBird is a large model, fine-tuning all of its layers is computationally expensive. Therefore, we freeze the first five layers (lower-level), meaning their weights remain unchanged during training, and fine-tune the remaining seven layers of the model [31]. The output of each BigBird model is a vector of length 768. We feed this new vector as an input to a dense layer that outputs a vector of length 256, which we found to be the most effective based on some preliminary experiments. We send this 256-length vector to a prediction head, which leads to an output layer of two neurons. We use ReLu [32] as an activation function for each neuron. We add dropout layers of 0.4 to eliminate some neurons randomly from the network during the training phase to avoid overfitting [33]. We use the softmax function [34] to convert the output layer's logits into the probability of a positive or negative label based on the results of the output layer. Finally, we use the AdamW optimizer [35] for parameter optimization and NLLLoss (Negative Log Likelihood Loss) [36] to compute the loss from predicted log-probabilities. The result of this training is a model that, given token representations of a tuple of tests, outputs two scores: positive class score and negative class score, reflecting the likelihood of a positive and negative label, respectively. Section III-C further details how these scores are used.

We fine-tune the pre-trained model for 30 epochs to enhance the model's performance. Given all the tuples from a training set (with a balanced set of both positive and negative labeled tuples), we randomly set aside 10% of them for validation. For each epoch, we evaluate the model's validation loss. After each epoch, if the model achieves the lowest validation loss, we proceed to update the weights of the model for the next epoch using this current optimal model's weights. Additionally, we utilize early stopping, which is configured to halt training if there is no improvement after 10 consecutive epochs, to mitigate overfitting. We ultimately produce the best model with the minimum validation loss among the epochs [37]. To preserve determinism in our results, we set the seed for the random number generator during testing and inference [21]. We find that our model provides deterministic outputs when we run inference with it on the same input 100 times.

B. RankF$_O$

RankF$_O$ leverages execution information of tests in different test-orders to compute scores for likely OD-relevant tests. Given an OD test and the execution result of the test suite in different test-orders, RankF$_O$ outputs a likelihood score for each test being an OD-relevant test for the given OD test.

If the OD test is a victim, then a polluter must be one of the tests that run before the victim in a failing test-order. Conversely, if the OD test is a brittle, then a state-setter must be one of the tests that run before the brittle in a passing test-order. Intuitively, the more often we see the same tests that come before the victim/brittle in a failing test-order/passing test-order, respectively, the more likely that test is the polluter/state-setter for the victim/brittle, respectively. Similar to RankF$_L$, RankF$_O$ also computes two scores for each test: the positive class score for the likelihood to be an OD-relevant test and negative class score for the likelihood to not be an OD-relevant test. RankF$_O$ computes these scores based on occurrence of each test and the position of each test when the OD test is passing and failing. For example, for a victim, depending on whether it is failing or passing in a given test-order, we increment the appropriate score, i.e., positive class scores for failing test-order and negative class scores for passing test-order. The more test-orders processed, the more confident RankF$_O$ is that the scores reflect the likelihood of tests being true OD-relevant tests for the OD test. We propose five different heuristics for scoring tests:

Plus One (+1): This heuristic gives a score of $+1$ for each test before an OD test in a given test-order. The intuition is that a test that frequently appears before a victim/brittle when the OD test is failing/passing, respectively, is more likely to be an OD-relevant test than a test that never appeared before the OD test (e.g., if a particular test is always running before a victim when it fails, then this test is likely a polluter).

#Methods (#M): This heuristic uses the number of tests before an OD test (ot) in a test-order to give a score of $1/indexOf(ot)$ to each test before an OD test, where $indexOf$ returns the index of a specific test in a given test-order. The intuition is that when a victim/brittle fails/passes, respectively, the test-orders where the number of tests before the OD test is small should have more weight on the likelihood of the tests being OD-relevant tests than test-orders in which there were more tests before the OD test.

Distance (D): This heuristic takes the distance between a given test (gt) and the OD test (ot) in a test-order, and gives a score of $1/(indexOf(ot) - indexOf(gt))$ to the given test. The intuition is that tests closer to the OD test is more likely to be an OD-relevant test.

Combined (+1, D): This heuristic uses Plus One to give scores to the tests and uses Distance to break ties.

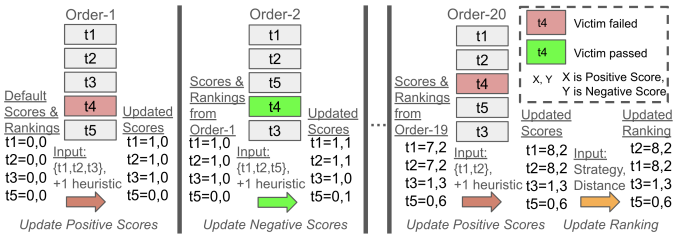

Fig. 4: Example of RankF$_O$ Combined (+1, D) heuristic.

Combined (#M, D): This heuristic uses #Methods to give scores to the tests and uses Distance to break ties.

After the positive/negative class scores are calculated, RankF$_L$ and RankF$_O$ use the scores to rank tests. Section III-C describes in detail how these scores are used. Regardless of how they are used, it is possible for the scores to tie (i.e., be the same for multiple tests). The intuition for the two Combined heuristics is to combat the problem where the use of scores from the #Methods and Plus One heuristics results in tests with tied scores, especially when the number of test-orders given to RankF$_O$ is small. We only consider these two Combined heuristics because Combined heuristics where Distance is first will rarely result in the same scores, unlike #Methods and Plus One, which updates the score of every test the same amount in each test-order. We also do not consider combining the two #Methods and Plus One heuristics with each other because, intuitively, they are not as effective as Distance at breaking ties. For the Distance, #Methods, and Plus One heuristics, if there are ties in the ranks of the tests after the scores are used, the tied tests are alphabetically ordered in the ranked list. For the two Combined heuristics, if there are ties in the ranks of tests, the tied tests are ordered in descending order of their distance to the OD test in the last test-order.

1) Example for RankF$_O$: Figure 4 showcases an example of the Combined (+1, D) heuristic of RankF$_O$. In this example, t4 is a victim and its true polluter is t2, along with a cleaner t5. Order-1 is a failing test-order (the true polluter t2 runs before t4 with no cleaners in between), therefore the positive class scores of tests before t4 (t1, t2, t3) are increased by 1. Order-2 is a passing test-order (t5 is a cleaner and runs in-between t2 and t4), so the negative class scores of tests before t4 (t1, t2, t5) are increased by 1. After processing the final test-order (Order-20), tests t1 and t2 have the same highest positive class scores and lowest negative class scores. The Combined (+1, D) strategy then relies on Distance to break the tie by considering the distance between t1 and t2 to the victim in Order-20 – ultimately, finding that t2 is closer to t4 and ranking it correctly higher than t1 as an OD-relevant test for t4.

C. Ranking Tests

After obtaining positive class scores and negative class scores for tests from RankF$_L$ or RankF$_O$, we use those scores to rank the tests in order of likelihood to be OD-relevant tests. We use three strategies to rank tests: positive class strategy, negative class strategy, and combined class strategy.

Positive class strategy organizes all potential OD-relevant tests by their positive class scores in descending order, placing those with the highest likelihood at the forefront. Negative class strategy arranges the candidate tests by their negative class scores in ascending order, aiming to position tests with the highest likelihood of being *not* an OD-relevant test at the bottom. Combined class strategy involves subtracting the test's negative class score from its positive class score, subsequently organizing the tests by this score difference in descending order. The idea is that tests with higher difference scores are those where the positive class score substantially outweighs the negative class score, suggesting that such tests are more likely to be true OD-relevant tests. In Section V-C, we evaluate how our approaches perform using either and both scores.

IV. Experimental Setup

We study the following research questions:

- **RQ1**: What is the efficiency and effectiveness of RankF$_L$ and RankF$_O$ at ranking + confirming OD-relevant tests?
- **RQ2**: What effect do different test-orders and heuristics have on RankF$_O$'s effectiveness?
- **RQ3**: What effect do different ranking strategies have on the effectiveness of RankF$_L$ and RankF$_O$?

We evaluate RQ1 to see whether RankF can find the true OD-relevant tests faster than baselines from prior work that also find OD-relevant tests [7, 11]. We evaluate RQ2 to assess how RankF$_O$ performs using different test-orders and heuristics. Finally, we evaluate RQ3 to see which ranking strategies enable RankF$_L$ and RankF$_O$ to be the most effective.

A. Dataset

RankF is meant to help find OD-relevant tests after OD tests have been detected. As such, RankF's evaluation requires a dataset of known OD tests and OD-relevant tests. There are various datasets [5, 12, 18]–[20, 38] that contain OD tests, however to the best of our knowledge, the only dataset that contains both OD tests and OD-relevant tests is from Wei et al. [12]. This dataset consists of 249 OD tests detected from 44 modules[1] across 25 open-source projects. This dataset also provides for each OD test the corresponding OD-relevant tests (e.g., the polluters for a victim and the cleaners for a polluter-victim pair). We further confirm the correctness of this data by explicitly running all the OD tests with their corresponding OD-relevant tests, i.e., we run each polluter/state-setter before their respective victim/brittle to ensure the test fails/passes as expected, and for each polluter-victim pair, we run the cleaner in-between to check that the victim passes. We find that some OD tests in the dataset could not be reproduced from running tests with Maven Surefire, and after sharing our findings with Wei et al. [12] to confirm, we exclude them from our dataset. In the end, we are able to reproduce 155 OD tests and their respective OD-relevant tests from 34 modules [21].

We also collect for each module the list of all tests in the test suite. We first run the test suite in each module to obtain

[1] A Maven project may contain many modules, each with their own test suite.

TABLE I: Details of our evaluation subjects. "B." represents Brittle, "V." represents Victim, "P." represents Polluter. The module corresponding to each ID is in our artifact [21].

ID	Project	Num. of Tests	OD	B.	V.	P.-V.	Suite Runtime
M1	Activiti/Activiti	42	21	4	17	115	5.5
M2	Apache/Struts	61	22	0	22	77	1.7
M3	alibaba/fastjson	4781	2	2	0	0	7.9
M4	apache/dubbo	106	3	0	3	8	1.8
M5	apache/dubbo	499	1	0	1	0	1.7
M6	apache/dubbo	5	1	0	1	0	1.8
M7	apache/dubbo	58	2	0	2	2	1.6
M8	apache/dubbo	65	2	1	1	4	43.9
M9	apache/dubbo	21	2	0	2	13	1.4
M10	apache/hadoop	388	1	0	1	1	16.7
M11	c2mon/c2mon	5	1	0	1	1	14.0
M12	ctco/cukes	14	1	0	1	1	2.0
M13	doanduyhai/Achilles	233	1	1	0	0	18.5
M14	dropwizard/dropwizard	74	1	0	1	2	2.2
M15	elasticjob/elastic-job-lite	479	2	0	2	4	63.5
M16	fhoeben/hsac...	251	1	1	0	0	3.8
M17	hexagonframework/spring...	48	4	4	0	0	3.2
M18	jhipster/jhipster-registry	53	2	0	2	2	8.7
M19	kevinsawicki/http-request	163	25	0	25	24	1.3
M20	ktuukkan/marine-api	925	12	0	12	12	1.4
M21	openpojo/openpojo	588	3	0	3	12	2.8
M22	spring-projects/spring-boot	1734	2	0	2	2	13.3
M23	spring-projects/spring-boot	735	6	0	6	0	13.6
M24	spring-projects/spring-boot	584	1	1	0	0	12.9
M25	spring-projects/spring-boot	219	1	1	0	0	59.4
M26	spring-projects/spring...	10	2	0	2	2	3.9
M27	spring-projects/spring-ws	61	2	0	2	2	1.9
M28	tbsalling/aismessages	44	2	0	2	0	0.9
M29	vmware/admiral	478	1	1	0	0	4.7
M30	vmware/admiral	302	1	0	1	0	9.5
M31	wikidata/wikidata-toolkit	49	3	3	0	0	1.4
M32	wikidata/wikidata-toolkit	23	2	0	2	1	1.1
M33	wildfly/wildfly	81	21	1	20	0	2.3
M34	zalando/riptide	40	1	0	1	0	6.9
Total/ Avg.		**13219**	**155**	**20**	**135**	**285**	**11.2**

16166 tests (excluding the tests skipped by Maven). We then extract the test-method body of each test using srcML [28]. We exclude 2947 tests that we cannot parse their test-method bodies. Upon inspection of a few such tests, we find them to inherit code from parent test classes that are not even in the same module as the tests. In total, we have 13219 tests in our 34 evaluation modules. Table I lists the subjects used in our evaluation. For each module, the number of tests, OD tests, and polluter-victim pairs that have cleaners are shown, along with the time to run the test suite of each module.

B. OBO Baseline

Our first baseline is to perform the one-by-one (OBO) approach [11] (Section II), a straightforward way to find OD-relevant tests. We simulate OBO by looking for the first polluter/state-setter in a failing/passing test-order, respectively, going through each test as a candidate OD-relevant test within the given test-order. As OBO's effectiveness is dependent on the given test-order, we randomly generate 10 different failing/passing test-orders for each victim/brittle, respectively, ensuring they are valid test-orders (i.e., no interleaving of tests across test classes [18]). By definition, an OD test must have both a failing and a passing test-order for it to be considered an OD test. We can determine whether an OD test passes or fails in a generated test-order based on the position of the OD-relevant tests relative to the OD test in a given test-order. (The possible failing/passing test-orders depends on the number of tests and the number of OD-relevant tests for each OD test, e.g., four tests containing one victim and one polluter has 12 failing test-orders, while five tests containing one victim, one polluter, and one cleaner has 40 failing test-orders. All of our subjects have more than 10 failing/passing test-orders. We also verify that each of our generated test-order is unique). We refer to the average results of 10 test-orders for OBO as OBO_{avg}.

We also simulate a theoretical worst case scenario for OBO to find OD-relevant tests by creating the test-order in which the first OD-relevant test is in the worst possible position. To create the test-order, we move all OD-relevant tests to the very end of the test-order; the other tests can be permuted in any ordering. If there are multiple OD-relevant tests, we sort them by runtime, keeping the OD-relevant test with the highest runtime first among the OD-relevant tests. Doing so ensures that OBO will find this highest runtime OD-relevant test before the others. There can be more than one worst test-order if there are multiple OD-relevant tests with the highest runtime, but choosing any such order will result in the same worst case scenario. We refer to this baseline as OBO_{max}.

C. Delta-Debugging Baseline

Shi et al. previously used delta-debugging [10] in iFixFlakies [7] to find OD-relevant tests (Section II), which we use as a baseline. Given a failing/passing test-order for a victim/brittle, respectively, delta-debugging divides the sequence of tests before the OD test in that test-order in half, checking which half makes the victim/brittle fail/pass, respectively, and recursively searches down that half. The search process stops once it reaches a single test that makes the victim/brittle fail/pass, respectively; that test is an OD-relevant test.

While delta-debugging can be directly used to find the polluter/state-setter, it may not always find a cleaner for a given polluter-victim pair. The starting condition for delta-debugging is a passing test-order where the polluter runs before the victim, so the tests in-between contain a cleaner, but such a test-order is not guaranteed, i.e., the passing test-orders for a victim has the victim run before the polluter. If the passing test-order does not have the polluter before the victim, then iFixFlakies resorts to using OBO on the entire passing test-order to find cleaners. Our use of delta-debugging is the same as iFixFlakies – we rely on delta-debugging if the polluter is before the victim, and rely on OBO otherwise.

D. RankF$_L$ Setup

For RankF$_L$, we prepare a labeled dataset for training using the OD tests we use in our evaluation. The training dataset needs both positive- and negative-labeled tuples of tests, where a positive label means the tuple represents an OD test and its true OD-relevant test (or a polluter-victim pair and its true cleaner), while a negative label means the other test in the tuple is not a true OD-relevant test for the OD test. We ensure that the tuples always include at least one positive label tuple. We train the models in a per-module manner, similar to prior work on predicting flaky tests [16]. When we evaluate on tests from one project, we train using tuples from all other projects. The

goal is to simulate a scenario where a developer takes a model previously trained on other projects and uses it for their own project. For such a simulation, our model is trained without observing any tests from the project that is being evaluated.

As we are guaranteed both failing and passing test-orders for every OD test, we can use those test-orders to help RankF$_L$ run fewer tests. Specifically, RankF$_L$ needs to score only the tests before the OD test in failing/passing test-orders for victim/brittles, respectively; the tests after the OD test need not be scored and ranked. For cleaners, RankF$_L$ scores and ranks all tests in the test suite as the first cleaner may be before or after a victim in any given test-order. RankF$_L$ uses the same failing/passing test-orders used by OBO$_{avg}$ (Section IV-B).

E. RankF$_O$ Setup

For RankF$_O$, we construct valid random test-orders of the tests. We use two types of test-orders (1) 20 random test-orders per module, (2) test-orders at which one can be at least 95% sure of a given OD test to pass/fail at least once and then evaluate how RankF$_O$ performs, to evaluate its effectiveness as it analyzes more test-orders. Developers may use rerun-based OD test detection techniques, such as iDFlakies [18], to obtain different test-orders for RankF$_O$. We choose 20 orders to match the number suggested by iDFlakies.

F. Runtime Environment

For our evaluation, we run on Ubuntu machines with i7 eight-core processors and 32 GB of RAM to run RankF$_L$ and RankF$_O$ and to measure test runtimes. For RankF$_L$, we fine-tune BigBird using a different Linux machine equipped with a single NVIDIA RTX A5000 GPU and 125 GB of RAM. This setup is based on CUDA version 12.0.

V. Evaluation

A. RQ1: RankF Efficiency and Effectiveness

1) Efficiency Methodology: We evaluate efficiency in terms of time per approach to find the first OD-relevant test. For RankF$_L$, this time includes the time to run the model on the tests in a given test-order. For RankF$_O$, this time includes the time to analyze the provided set of test-orders (we assume the sets of test-orders were run previously as part of some detection process [5, 18]). After obtaining rankings, we measure the confirmation time as the time needed to run each test in the ranked order together with the OD test until we confirm the first correct OD-relevant test. Each time we run a group of tests, we include the overhead from invoking Maven Surefire. We obtain the overhead time to invoke Surefire from running the entire test suite of a module and subtracting the total time taken by all tests (as specified in Surefire reports) from the total time taken by Surefire; we run this process 30 times to obtain an average overhead time. We report the total time for using an approach as the sum of the analysis time and the time to run the tests in the ranked order. As both RankF$_L$ and RankF$_O$ produce 10 ranked lists of OD-relevant tests, we report the average time for the 10 lists per OD test for each approach. For OBO, we compute the average time in the same way as RankF$_L$, except we count only the time to run the tests with the same 10 test-orders used by RankF$_L$. For delta-debugging, we compute the cumulative time to run all tests together plus the Surefire overhead time from each iteration of the search, where each iteration may run multiple tests along with the OD test. The overall time for an OD test and a given test-order is the sum of the time for all iterations until an OD-relevant test is found.

2) Efficiency Results: Table II, Table III, and Table IV show the time for our two proposed approaches and the baselines under the column "Time to Rank + Confirm in seconds". We further highlight the cell corresponding to the best (lowest) time across all approaches and baselines for each module.

We see that RankF always outperforms OBO$_{avg}$, and it outperforms delta-debugging for all modules except for two modules for polluters and one module for state-setters. For example, Table III shows RankF$_L$ and RankF$_O$ achieve a median time of 59.5 and 13.3 seconds, respectively, to find the first state-setter, compared to 489.5 and 118.5 seconds for OBO$_{avg}$ and delta-debugging, respectively. When looking at medians, RankF$_O$ always finds the first OD-relevant test faster than the baselines and RankF$_L$. Although RankF$_O$'s average performance is affected by extreme outliers (e.g., M22 in Table II), it is still the fastest for a majority of modules (fastest for 36/56 modules in tables II to IV).

If we compare just RankF$_L$ with the baselines, RankF$_L$ is always faster than OBO$_{avg}$, and is faster than delta-debugging, on median, except for finding polluters, in which they have the same median time. From Table II, we see that delta-debugging is the faster at finding a polluter for most modules (16/26). However, RankF$_L$ outperforms delta-debugging for finding the first state-setter (7/11 modules) or cleaner (18/19 modules). Unlike the baselines, RankF$_L$ provides rankings for all tests – a developer looking to find more than one OD-relevant test can use those rankings to speed up their search.

We also show under "Time to Rank in seconds" the time it takes for RankF$_L$ and RankF$_O$ to just rank the tests, along with the percentage of that ranking time relative to the overall time to both rank and confirm tests. RankF$_L$ takes longer to rank tests than RankF$_O$ (which has a negligible cost of <100 ms). RankF$_L$ needs to process tuples of test code (larger sized tuples take longer processing time) and uses a complex LLM, which substantially increases its computational time.

3) Effectiveness Methodology: To measure the effectiveness of RankF at ranking OD-relevant tests, we report (1) the number of OD tests for which the approach finds a correct OD-relevant test as the top-ranked test, essentially Rank-1 (the more the better), and (2) the rank of the first OD-relevant test found by the approach in the ranked list (the lower the better).

Number of OD test with OD-relevant test at Rank-1. For RankF$_L$, we consider it to have found the OD-relevant test as top-ranked if such a test is Rank-1 for at least 50% of the 10 test-orders we use per OD test (Section IV-D). For RankF$_O$, we randomly sample 10 sets of test-orders, each with 20 test-orders, for each module (Section IV-E). We consider RankF$_O$ to have found the OD-relevant test as top-ranked if such a test

TABLE II: Results of finding Polluter (P.) for a given Victim. "# Victim w/ P. @ Rank-1" is the number of victims where P. is found at rank one. "1st P. Ranking" is the rank of the first P. "Time to Rank + Confirm in seconds" is the sum of the time to rank and confirm the first P. Highlighted cells represent the best approach in ranking + confirming.

ID	# Victim w/ P. @ Rank-1		1st P. Ranking				Time to Rank + Confirm in seconds					Time to Rank in seconds		Min
	$RankF_L$	$RankF_O$	$RankF_L$	$RankF_O$	OBO_{avg}	OBO_{max}	$RankF_L$	$RankF_O$	OBO_{avg}	OBO_{max}	DD	$RankF_L$	$RankF_O$	Orders
M1	4	6	7.4	3.4	5.7	23.6	41.8	18.8	31.3	130.6	35.8	0.7 (1.6%)	0.3 (1.6%)	468.9
M2	0	6	14.9	8.3	18.2	49.9	26.6	14.3	31.1	85.4	16.1	1.0 (3.8%)	<0.1 (0.2%)	250.6
M4	0	1	13.0	7.6	42.4	99.3	26.2	13.9	77.8	182.8	19.6	1.9 (7.2%)	<0.1 (0.0%)	89.8
M5	0	1	193.6	2.9	228.9	491.0	368.2	5.5	430.4	914.3	70.5	7.4 (2.0%)	<0.1 (0.0%)	34.4
M6	1	1	1.5	1.0	1.7	5.0	4.2	2.7	4.6	14.3	5.4	0.1 (3.2%)	<0.1 (3.4%)	1.4
M7	0	0	10.1	3.7	22.2	57.0	20.5	6.8	42.2	108.6	25.7	1.3 (6.2%)	<0.1 (0.0%)	208.3
M8	0	0	9.9	35.1	41.8	55.0	24.4	176.2	209.3	241.9	309.2	1.5 (6.1%)	<0.1 (0.0%)	944.8
M9	2	2	1.8	2.2	3.1	7.5	3.8	4.1	5.7	13.8	7.6	0.5 (13.0%)	<0.1 (0.4%)	12.7
M10	0	0	2.3	9.4	149.0	365.0	19.5	49.2	758.7	1825.6	545.6	8.1 (41.6%)	<0.1 (0.0%)	-
M11	0	1	3.0	1.0	2.3	5.0	47.2	15.7	36.1	78.9	32.6	0.1 (0.2%)	<0.1 (0.0%)	2.9
M12	0	1	6.8	1.0	8.1	14.0	10.4	2.5	12.2	21.0	6.6	0.2 (2.4%)	<0.1 (0.0%)	23.5
M14	0	0	10.3	9.3	43.5	68.0	24.5	21.2	95.0	148.1	28.5	1.2 (4.9%)	<0.1 (0.0%)	209.5
M15	0	1	63.5	88.4	231.0	468.5	2679.2	5614.9	14670.9	29754.4	5012.5	7.8 (0.3%)	<0.1 (0.0%)	5.9
M18	0	1	30.9	4.0	14.1	48.0	274.2	35.9	124.5	423.2	78.9	1.0 (0.3%)	<0.1 (0.1%)	27.8
M19	0	18	84.7	1.7	66.8	163.0	116.8	2.2	89.7	218.7	17.8	3.2 (2.8%)	<0.1 (0.0%)	20.9
M20	0	0	319.3	94.5	278.8	917.2	497.1	136.9	402.8	1325.1	92.3	16.4 (3.3%)	<0.1 (0.0%)	905.8
M21	0	0	29.3	35.3	173.2	577.3	87.5	100.8	489.9	1631.7	90.5	8.7 (10.0%)	<0.1 (0.1%)	110.1
M22	0	0	136.6	1330.5	553.4	1683.5	1849.2	17697.5	7360.5	22390.3	396.3	31.7 (1.7%)	<0.1 (0.0%)	-
M23	0	6	146.7	1.0	410.7	390.5	2006.9	13.6	5529.7	9892.8	1116.7	14.7 (0.7%)	<0.1 (0.0%)	3.9
M26	2	0	1.2	3.8	6.3	8.0	5.1	15.0	24.9	31.8	19.8	0.2 (3.7%)	<0.1 (0.1%)	118.2
M27	0	2	9.1	2.6	25.6	56.0	18.4	5.1	48.8	107.0	36.9	1.0 (5.6%)	<0.1 (0.0%)	-
M28	0	2	8.3	1.0	23.3	43.0	8.2	0.9	20.9	38.6	5.1	0.7 (8.9%)	<0.1 (2.7%)	3.3
M30	0	1	125.5	1.0	146.6	302.0	1199.8	6.1	914.8	1887.3	211.1	6.1 (0.5%)	<0.1 (0.0%)	2.2
M32	0	2	4.8	1.0	15.7	23.0	6.9	1.3	21.0	31.0	6.0	0.5 (6.6%)	<0.1 (0.1%)	4.8
M33	0	2	20.6	6.3	32.5	75.8	49.3	14.8	75.3	176.0	25.1	1.5 (3.1%)	<0.1 (0.0%)	16.6
M34	0	1	13.0	1.0	19.0	39.0	90.6	6.9	131.2	269.2	36.4	0.8 (0.9%)	<0.1 (0.0%)	2.7
Total/	**9**	**55**	-	-	-	-	-	-	-	-	-	-	-	-
Avg./	-	-	**64.6**	**34.2**	**86.7**	**211.9**	**259.8**	**371.1**	**680.8**	**1518.4**	**166.5**	**4.3 (1.6%)**	**<0.1 (<0.1%)**	**206.0**
Median	-	-	**11.7**	**3.5**	**29.0**	**56.5**	**34.2**	**14.1**	**83.8**	**179.4**	**34.2**	**1.2 (5.3%)**	**<0.1 (<0.1%)**	**23.0**

TABLE III: Results of finding State-Setter (SS.) for a given Brittle. "# Brittle w/ SS. @ Rank-1" is the number of brittles for which SS. is found at top one. "1st SS. Ranking" is the rank of first SS. "Time to Rank + Confirm in seconds" is the sum of the time to rank and confirm the first SS. Highlighted cells represent the best approach in ranking + confirming SS.

ID	# Brittle w/ SS. @ Rank-1		1st SS. Ranking				Time to Rank + Confirm in seconds					Time to Rank in seconds		Min
	$RankF_L$	$RankF_O$	$RankF_L$	$RankF_O$	OBO_{avg}	OBO_{max}	$RankF_L$	$RankF_O$	OBO_{avg}	OBO_{max}	DD	$RankF_L$	$RankF_O$	Orders
M1	1	4	11.5	1.0	18.6	39.8	64.2	5.5	103.1	219.7	29.4	0.7 (1.1%)	<0.1 (0.7%)	4.9
M3	0	0	19.6	144.6	93.6	187.0	218.2	1140.5	738.1	1485.1	301.7	64.3 (29.5%)	<0.1 (0.0%)	-
M8	1	1	6.5	1.0	15.5	58.0	23.5	1.9	60.9	266.4	419.0	1.2 (5.3%)	<0.1 (0.3%)	15.6
M13	1	0	2.1	3.7	57.4	199.0	43.6	68.4	1062.4	3684.0	245.1	4.7 (10.9%)	<0.1 (0.0%)	999.0
M16	0	1	160.9	1.0	130.9	251.0	607.3	3.7	489.5	939.0	33.6	5.3 (0.9%)	<0.1 (0.0%)	3.8
M17	1	0	9.8	22.0	19.8	41.0	32.0	70.0	62.9	130.5	35.2	0.9 (2.7%)	<0.1 (0.0%)	-
M24	0	1	36.3	1.0	408.5	584.0	495.7	13.3	5423.5	7753.8	119.7	10.4 (2.1%)	<0.1 (0.2%)	2.1
M25	0	1	5.4	1.0	86.2	219.0	76.9	13.3	1146.4	2912.4	388.2	5.0 (6.5%)	<0.1 (0.0%)	3.4
M29	0	0	10.9	87.9	310.2	464.0	59.5	446.9	1538.3	2306.4	118.5	6.6 (11.1%)	<0.1 (0.0%)	153.0
M31	3	1	1.1	5.4	2.3	5.7	2.2	7.4	3.4	9.0	11.1	0.6 (28.7%)	2.5 (33.8%)	155.6
M33	0	0	14.2	11.8	20.7	41.0	33.6	26.8	47.2	93.2	22.2	1.3 (3.8%)	<0.1 (0.0%)	60.2
Total/	**7**	**9**	-	-	-	-	-	-	-	-	-	-	-	-
Avg./	-	-	**18.2**	**25.2**	**68.9**	**126.5**	**108.4**	**159.0**	**595.9**	**1117.7**	**112.1**	**8.6 (7.9%)**	**0.4 (0.2%)**	**86.2**
Median	-	-	**10.9**	**3.7**	**57.4**	**187.0**	**59.5**	**13.3**	**489.5**	**939.0**	**118.5**	**4.7 (5.3%)**	**<0.1 (<0.1%)**	**15.0**

TABLE IV: Results of finding Cleaner (C.) for a given Victim-Polluter. "# Victim w/ C. @ Rank-1" is the number of victims for which C. is found at top one. "1st C. Ranking" is the rank of first C. "Time to Rank + Confirm in seconds" is the sum of the time to rank and confirm the first C. Highlighted cells represent the best approach in ranking + confirming C.

ID	# Victim w/ C. @ Rank-1		1st C. Ranking				Time to Rank + Confirm in seconds					Time to Rank in seconds		Min
	$RankF_L$	$RankF_O$	$RankF_L$	$RankF_O$	OBO_{avg}	OBO_{max}	$RankF_L$	$RankF_O$	OBO_{avg}	OBO_{max}	DD	$RankF_L$	$RankF_O$	Orders
M1	1	12	15.3	5.9	11.8	26.6	86.0	32.8	65.3	147.7	100.0	1.1 (1.3%)	<0.1 (0.0%)	273.6
M2	3	0	26.1	20.3	35.9	52.7	47.5	36.0	63.3	92.6	57.2	1.6 (3.5%)	<0.1 (0.0%)	61.1
M4	2	4	21.2	1.1	27.1	103.0	43.2	1.9	50.4	190.2	44.8	3.2 (7.5%)	0.2 (8.9%)	20.2
M7	0	0	4.0	5.5	15.9	51.0	9.3	9.4	30.4	97.5	37.4	1.9 (20.7%)	<0.1 (0.0%)	49.9
M8	4	4	1.0	1.0	6.8	33.0	4.1	2.4	17.4	88.9	1486.4	2.1 (51.0%)	<0.1 (0.0%)	103.1
M9	2	1	4.2	2.5	8.2	17.2	8.5	4.5	15.0	31.3	11.9	0.8 (9.9%)	<0.1 (0.1%)	157.0
M10	0	0	323.0	27.0	248.3	366.0	2097.4	204.6	1608.1	2357.6	1615.8	11.0 (0.5%)	<0.1 (0.0%)	-
M11	1	1	1.0	1.0	2.2	4.0	16.3	16.4	36.5	66.0	33.4	<0.1 (0.6%)	<0.1 (0.1%)	5.8
M12	1	1	1.0	1.0	7.2	13.0	1.3	1.4	10.8	19.5	9.6	<0.1 (6.0%)	<0.1 (0.0%)	9.3
M14	2	0	1.0	2.0	9.2	62.0	5.5	6.5	30.9	203.7	84.7	2.0 (36.8%)	<0.1 (0.0%)	115.6
M15	0	0	9.8	1.9	32.8	378.0	585.8	123.9	2083.2	24008.4	4860.0	13.6 (2.3%)	<0.1 (0.0%)	942.5
M18	0	1	10.0	12.2	33.7	48.5	90.5	108.1	300.5	432.6	286.9	1.4 (1.6%)	<0.1 (0.0%)	11.6
M19	0	14	44.5	2.1	97.8	163.0	64.6	2.8	131.4	218.8	111.5	4.8 (7.4%)	<0.1 (0.8%)	20.9
M20	0	0	30.8	14.8	571.5	903.6	70.5	21.4	825.9	1305.6	367.7	26.0 (36.8%)	<0.1 (0.0%)	139.3
M21	0	0	70.6	119.1	170.0	575.8	216.5	336.4	480.1	1628.0	376.6	16.6 (7.7%)	<0.1 (0.0%)	237.2
M22	0	0	443.0	35.8	973.4	1675.5	5948.8	476.7	12953.3	22292.5	7862.8	48.5 (0.8%)	<0.1 (0.0%)	298.6
M26	1	1	1.5	1.5	3.5	4.5	6.4	4.5	14.0	18.3	25.6	0.2 (3.6%)	<0.1 (0.0%)	43.6
M27	2	1	1.0	1.1	11.0	45.0	3.5	2.0	21.0	86.0	32.8	1.6 (46.5%)	<0.1 (2.7%)	123.5
M32	0	1	6.0	1.0	9.7	23.0	8.6	1.3	13.0	31.0	13.0	0.6 (6.8%)	<0.1 (0.0%)	7.1
Total/	**19**	**41**	-	-	-	-	-	-	-	-	-	-	-	-
Avg./	-	-	**26.6**	**14.4**	**63.9**	**125.1**	**125.0**	**45.4**	**240.6**	**742.5**	**251.7**	**3.9 (3.1%)**	**<0.1 (<0.1%)**	**171.5**
Median	-	-	**9.8**	**2.1**	**15.9**	**51.0**	**43.2**	**9.4**	**50.4**	**147.7**	**84.7**	**1.9 (6.8%)**	**<0.1 (<0.1%)**	**82.0**

is Rank-1 for at least 50% of the sets of test-orders. We do not report Rank-1 for OBO_{avg}, OBO_{max}, and delta-debugging, because they have no ranking scheme, so any OD-relevant test ranked first is by pure chance, or such tests will always be ranked after non-OD-relevant tests for OBO_{max}.

Rank of the first OD-relevant test. For $RankF_L$, we report the average rank for the first true OD-relevant test found per OD test across the 10 test-orders. Similarly, we average the rank across 10 sets of test-orders to compute the rank of the first OD-relevant test for $RankF_O$. For the baseline OBO_{avg}, we use the same 10 test-orders used by $RankF_L$ for each OD test. We find the rank of the first OD-relevant test for each OD test in each test-order and report that average rank. For OBO_{max}, we compute the rank of the first OD-relevant test found in the generated worst test-order (Section IV-B). We do not report for delta-debugging as it does not rank tests.

4) Effectiveness Results: From Tables II, III, and IV, we find that $RankF_L$ and $RankF_O$ have a non-trivial number of OD tests with OD-relevant tests at Rank-1. For example, we find that a substantial number of brittles, up to 45% (9/20), can already have a state-setter at Rank-1 for $RankF_O$. That being said, these results suggest that there can still be meaningful future work that improves the number of OD tests with OD-relevant tests at Rank-1.

We also see that the time each approach takes is highly correlated to the rankings, where the better the rankings of real OD-relevant tests are, the faster the tests are found. This relationship between the rank and time directly influences how $RankF_O$ compares with all baselines in our results (i.e., if an approach has a better rank than another approach, then the first approach must also have a better runtime than the second approach). Any technique where the time to rank is high can result in the relationship being not true, which we find is not true for $RankF_L$ (Section V-A2). For instance, in Table IV module M27, there are two OD tests and $RankF_L$ identify the corresponding cleaner at Rank-1 for both OD tests, while $RankF_O$ identify only one OD test's cleaner at Rank-1. Despite having more cleaners at Rank-1, $RankF_L$'s total time is still worse than $RankF_O$, because $RankF_L$ takes more time to rank.

B. RQ2: Effect of Test-Orders and Heuristics on $RankF_O$

1) RQ2 Methodology: We evaluate the effect of using a different number of test-orders on $RankF_O$'s performance. To do so, we use 10 sets of 20 test-orders, where each set is called STO_{20}, compared against $STO_{95\%}$, which represents a dynamic number of test-orders that is calculated so that there is a ≈95% chance the OD test will pass and fail at least once when all test-orders are run. Both STO_{20} and $STO_{95\%}$ always draw from the same superset of test-orders (e.g., if $STO_{95\%}$ is 20, then $STO_{95\%}$ and STO_{20} will be the exact same set of test-orders). The way we calculate the number of orders is based on the probability of OD test failure obtained from prior work [12]. E.g., for brittles, we use $NumOrders = \frac{\log(0.05)}{\log(1-\frac{1}{X+1})}$, where X represents the number of state-setters.

2) RQ2 Results: For $STO_{95\%}$, on average, we used 24.6 and 22.7 test-orders for victims and brittles, respectively, slightly more than the default 20 test-orders used by iDFlakies [18] and STO_{20}. To use $STO_{95\%}$, one needs to know the number of OD-relevant tests, which a developer may not know a priori.

TABLE V: Comparison of $RankF_{O20}$ ranking functions, with average rank of Polluter (P.), State-Setter (SS.), Cleaner (C.).

	Plus One			#Methods			Distance		
ID	P.	SS.	C.	P.	SS.	C.	P.	SS.	C.
M1	3.4	1.0	5.9	4.1	1.0	6.0	3.5	1.1	5.8
M2	8.3	-	20.3	5.2	-	21.6	7.6	-	24.3
M3	-	144.6	-	-	82.6	-	-	103.3	-
M4	7.6	-	1.1	19.2	-	2.5	10.2	-	1.6
M5	2.9	-	-	1.9	-	-	1.4	-	-
M6	1.0	-	-	1.0	-	-	1.0	-	-
M7	3.7	-	5.5	13.7	-	3.5	5.5	-	7.7
M8	35.1	1.0	1.0	33.3	1.1	1.4	26.9	1.3	1.0
M9	2.2	-	2.5	2.1	-	3.9	2.0	-	4.6
M10	9.4	-	27.0	8.2	-	17.7	33.0	-	65.4
M11	1.0	-	1.0	1.0	-	1.0	1.0	-	1.0
M12	1.0	-	1.0	2.4	-	1.0	1.0	-	1.0
M13	-	3.7	-	-	6.8	-	-	28.1	-
M14	9.3	-	2.0	26.5	-	3.1	25.7	-	1.6
M15	88.4	-	1.9	73.5	-	7.5	80.0	-	2.6
M16	-	1.0	-	-	1.0	-	-	1.0	-
M17	-	22.0	-	-	23.6	-	-	13.2	-
M18	4.0	-	12.2	8.4	-	7.3	7.5	-	8.8
M19	1.7	-	2.1	3.1	-	2.5	15.1	-	18.2
M20	94.5	-	14.8	375.8	-	4.6	420.1	-	193.7
M21	35.3	-	119.1	52.3	-	32.5	62.2	-	81.3
M22	1330.5	-	35.8	543.5	-	276.8	650.8	-	472.2
M23	1.0	-	-	1.0	-	-	1.0	-	-
M24	-	1.0	-	-	1.0	-	-	1.0	-
M25	-	1.0	-	-	1.0	-	-	1.0	-
M26	3.8	-	1.5	4.0	-	1.2	1.2	-	2.3
M27	2.6	-	1.1	15.8	-	2.6	11.1	-	1.1
M28	1.0	-	-	1.0	-	-	1.1	-	-
M29	-	87.9	-	-	1.0	-	-	1.1	-
M30	1.0	-	-	1.0	-	-	1.0	-	-
M31	-	5.4	-	-	5.7	-	-	2.7	-
M32	1.0	-	1.0	1.9	-	1.5	1.0	-	1.5
M33	6.3	11.8	-	5.9	7.2	-	11.2	5.2	-
M34	1.0	-	-	1.0	-	-	1.0	-	-
Avg.	**34.2**	**25.2**	**14.4**	**46.4**	**12.0**	**20.9**	**53.2**	**14.5**	**47.1**

We refer to $RankF_O$ that uses STO_{20} as $RankF_{O20}$, and $RankF_O$ that uses $STO_{95\%}$ as $RankF_{O95\%}$. We randomly sample 10 sets of test-orders with the corresponding number of test-orders in each set depending on the variant. We average the ranking of the first OD-relevant test found between the 10 sets of test-orders. The average ranking of the first OD-relevant test for $RankF_{O20}$ is actually lower than $RankF_{O95\%}$, and the differences in the average ranking of the first polluter, state-setter, and cleaner found are 4.2, 2.0, and 0.3, respectively, higher for $RankF_{O95\%}$ than $RankF_{O20}$. Our results suggest that a small change in test-orders do not appear to increase the effectiveness of $RankF_O$ (and may even appear detrimental), although this change has no impact on the relative ranking of $RankF_O$ compared to $RankF_L$ and the baselines (i.e., $RankF_O$ still appears best).

We also evaluate the effects of the five different scoring heuristics (Section III-B) for $RankF_O$. Table V shows the average ranking of the first OD-relevant test found by $RankF_O$

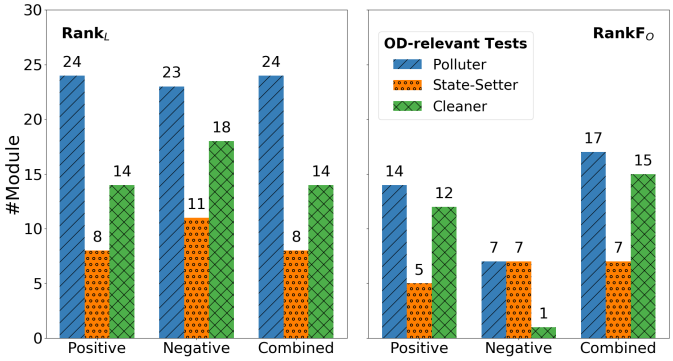

Fig. 5: Number of modules where a strategy is the best at ranking OD-relevant tests.

using three scoring heuristics. The cells with "-" indicate that there is no corresponding OD test of a particular category in that module. This table shows only the results using RankF$_{O20}$, given that RankF$_{O20}$ and RankF$_{O95\%}$ have similar results. We notice that RankF$_{O20}$ never needed to use any tie-breaking. Hence, Combined (+1, D) is the same as Plus One, and Combined (#M, D) is the same as #Methods, so we do not show their results. On average, RankF$_{O20}$ needs 19.0, 17.5, 18.2, test-orders for Distance, #Methods, Plus One heuristics, respectively, to not have any ties in the test scores. We find that the Plus One heuristic generally performs the best, finding the first OD-relevant test at the lowest rankings between all scoring heuristics, on average, except for finding the first state-setter per brittle. As such, we present results using Plus One in all other tables involving RankF$_O$.

We also compute the minimum number of test-orders that RankF$_O$ needs for it to always rank a correct OD-relevant test at Rank-1 and for its ranking to not change even if more test-orders are provided. To determine the minimum number of test-orders, we generate 10 sets of 1000 test-orders (which the first 20 in each set was used for STO$_{20}$) and find the test-order where a true OD-relevant test is ranked first and remains first even when subsequent test-orders are processed by RankF$_O$. We repeat this process for each of the 10 sets to get an average minimum number of test-orders. Tables II, III, and IV show this minimum number of test-orders to be 206.0, 86.2, and 171.5, on average, needed for polluters, state-setters, and cleaners, respectively. These numbers are substantially more than the 20 test-orders used by STO$_{20}$, indicating that a developer may need to run many more test-orders if they want to obtain stable Rank-1 results from RankF$_O$. However, based on RQ1, we see that just 20 test-orders can still substantially reduce the time needed to rank and confirm OD-relevant tests.

C. RQ3: Effect of Ranking Strategies

We compare the effectiveness of the three ranking strategies (Section III-C) against each other by counting the number of modules in which a particular strategy obtains the lowest OD-relevant test ranks among all strategies. That is, we obtain the average rank of OD-relevant tests for each approach and strategy, then count a module for a strategy when it produces the lowest average rank among all strategies. Figure 5 shows the effectiveness of the different ranking strategies used for RankF$_L$ and RankF$_O$, with the left subfigure for RankF$_L$ and the right for RankF$_O$. For each strategy, we show three bars, each corresponding to finding a different OD-relevant test: polluter (– blue bars with diagonal lines), state-setter (– orange bars with star patterns), and cleaner (– green bars with a crosshatched pattern).

RankF$_L$ benefits the most from using the negative class strategy. For polluters, the negative class strategy achieves the best ranking for 23 out of 26 modules (only one less than the positive class strategy and combined class strategy). For state-setters and cleaners, the negative class strategy is better than the other strategies (outperforming them by 3 modules for state-setters and 4 modules for cleaners). RankF$_O$ benefits the most from using the combined class strategy. For polluters, the combined class strategy achieves the best ranking for 17 out of 26 modules. For state-setters, the negative class strategy and combined class strategy achieve the best ranking for 7 out of the 11 modules. For cleaners, the combined class strategy achieves the best ranking for 15 out of 19 modules.

The results shown in all other tables use the best ranking strategy for each approach, namely negative class strategy for RankF$_L$ and combined class strategy for RankF$_O$.

VI. Discussion

Reason for RankF$_L$ effectiveness. RankF$_L$ is effective at predicting OD-relevant tests despite analyzing only test code. To better understand its effectiveness, we use SHAP [39] to analyze how the model attributes importance scores to tokens. We find that the model assigns higher importance scores to tokens that appear in-common between tests that are truly related, but these tokens receive lower scores for unrelated tests. As it seems the model relies on similarities between tests, we further investigate using TF-IDF [40] to compute similarity scores between the code of the test pairs, with the assumption that tests with higher similarity are more likely related. We find that using TF-IDF this way results in a worse ranking of tests, suggesting that the RankF$_L$ model analyzes more complex interactions beyond the presence of similar tokens.

Practicality of RankF$_O$. RankF$_O$ needs to analyze many test-orders to effectively rank OD-relevant tests, though it performs well even with just 20 test-orders (Section V-B). A developer using tools (e.g., iDFlakies [18]) to detect OD tests would already have many test-orders, which RankF$_O$ can use to find OD-relevant tests more efficiently. Developers may also be running tests in random test-orders as part of their development process during regression testing [17], which RankF$_O$ may also use. If a developer has very few or no test-orders to consult, we recommend the use of RankF$_L$.

Learning-to-rank evaluation. Our evaluation metrics focus on the practical application of running tests in a ranked order to more quickly find the first OD-relevant test. There are other learning-to-rank metrics that could be used to evaluate RankF's effectiveness. If we evaluate using the standard MAP score [41] with relevance score as 10, we find that RankF$_L$, RankF$_O$, and OBO$_{avg}$ achieve average MAP scores of 0.007,

0.251, and 0.000, respectively, for ranking polluters (higher is better). For state-setters, the scores are 0.067, 0.144, and 0.000, and for cleaners the scores are 0.050, 0.450, and 0.000, for RankF$_L$, RankF$_O$, and OBO$_{avg}$, respectively. We see that RankF$_L$ and RankF$_O$ achieve higher MAP scores than OBO$_{avg}$ for all types of OD-relevant tests. We do not have delta-debugging MAP scores as delta-debugging does not rank tests. More details on these results are in our artifact [21].

Use of GPT. We evaluate whether we can prompt a pre-trained LLM, namely GPT, without fine-tuning it to find OD-relevant tests. We design a prompt for gpt-3.5-turbo-0125 [42] that uses both code from an OD test paired with code from another candidate test, asking whether the other test is an OD-relevant test for the OD test[2].

We encounter two main problems with using GPT: (1) GPT is a third-party service with request limitations, and (2) there is a monetary cost. To reduce the cost of our experiments, we conducted a preliminary experiment with 32 modules and found that GPT performs worse than the BigBird model used by RankF$_L$. We excluded two modules (M3 and M22), because they have many tests (over 1500).

VII. Threats to Validity

Our work does not handle cases where one OD test is dependent on two or more tests (e.g., for victim v to fail, it must run after two tests in order <p1, p2, v>, while v passes in both <p1, v> and <p2, v>). However, prior work found dependence on multiple tests at once to be very rare [7].

We use srcML [28] to extract the test-method body. While srcML is a well-developed tool, we still encountered challenges such as needing to parse special characters within the code. In our manual analysis of a few dozen examples, we noticed only one inconsistency in extracting the method body.

For RankF$_L$, we use BigBird, which can be problematic when dealing with smaller test methods, leading to excessive padding. To mitigate this problem, we used a smaller limit (2048) than BigBird's absolute maximum (4096).

Our results may not generalize to projects not in our evaluation. To mitigate this threat, we used the data from Wei et al. [12]. This data consists of OD and OD-relevant tests in popular, Java projects that use Maven.

VIII. Related Work

Luo et al. [2] conducted the first empirical study on flaky tests in open-source projects, finding that test-order dependence was among the top-three reasons for test flakiness. Prior work proposed to detect OD tests, by generating random test-orders [5, 18]–[20], using code changes [43], systematically generating test-orders [12,13], and analyzing the code [44,45]. These techniques are only able to detect OD tests and categorize OD tests (victim or brittle), but they cannot find the corresponding OD-relevant tests. A developer can use RankF to efficiently find OD-relevant tests to help debug and fix OD tests. RankF$_O$ does leverage random test-orders, such as those taken from iDFlakies [18]; future work can evaluate RankF$_O$'s effectiveness when using systematically generated test-orders [12, 13]. RankF$_O$ and spectrum-based fault localization [46]–[50] share a common foundation of analyzing contrasting execution data to localize relevant entities. RankF$_O$ ranks tests based on their likelihood of being OD-relevant tests, while spectrum-based fault localization ranks program statements by their suspiciousness scores.

Researchers also proposed techniques to automatically repair OD tests [7]–[9]. These approaches all rely on OD-relevant tests to generate a patch. The prominent baselines for finding OD-relevant tests are either running tests one-by-one (OBO) with the OD test [11] or to apply delta-debugging to search for them within a test-order [7]. In our work, we propose the idea of first ranking tests based on likelihood of being OD-relevant tests and running them in that ranked order to confirm. We compare against OBO and delta-debugging as baselines, finding that we can find OD-relevant tests faster.

There has been recent work in leveraging LLMs for flaky-test related tasks. Fatima et al. [16] trained a LLM to predict which tests are flaky without running them. Others [26, 27] have recently proposed training a LLM to further predict the category of a flaky test. Fatima et al. [51] proposed using a LLM to predict the type of fix needed for flaky tests. Chen and Jabbarvand leveraged GPT to repair flaky tests, including OD tests [9]. Unlike prior work, we leverage LLMs to compute the likelihood of OD-relevant tests for a given OD test. Our approach requires the model to process features from multiple tests, as OD-relevant tests only matter w.r.t. an OD test. We evaluate effectiveness and efficiency in this problem domain, i.e., in how quickly one finds a correct OD-relevant test, instead of machine-learning metrics like F1-score.

IX. Conclusions

We propose RankF$_L$ and RankF$_O$ to rank tests by their likelihood of being OD-relevant tests for an OD test. Prior approaches for finding OD-relevant tests are often time-consuming, so we propose first ranking tests based on their likelihood of being OD-relevant tests, before running them to confirm, thereby reducing the overall search time. RankF$_L$ employs an LLM that considers only the test-method body – the code explicitly within a test method – to rank tests, whereas RankF$_O$ analyzes historical test-order execution data for the same purpose. Our results show that both RankF$_L$ and RankF$_O$ are more efficient than baselines at finding OD-relevant tests.

In the future, we plan to improve RankF by giving it other sources of information, such as dynamic execution traces of tests, and explore the combination of different techniques (e.g., using the results of RankF$_L$ or RankF$_O$ to guide the other).

Acknowledgements

We would like to acknowledge NSF grants CCF-2145774, CCF-2217696, and CCF-2338287, as well as Dragon Testing and the Jarmon Innovation Fund for their support.

[2]A sample of our prompt can be found in our artifact [21].

REFERENCES

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification & Reliability*, 2012.

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

[3] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *ESEC/FSE 2019: Proceedings of the 13th Joint Meeting on Foundations of Software Engineering*, 2019.

[4] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Transactions on Software Engineering and Methodology*, 2021.

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

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

[7] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "iFixFlakies: A framework for automatically fixing order-dependent flaky tests," in *ESEC/FSE 2019: Proceedings of the 13th Joint Meeting on Foundations of Software Engineering*, 2019.

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

[9] Y. Chen and R. Jabbarvand, "Neurosymbolic repair of test flakiness," in *ISSTA 2024: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

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

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

[12] 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 2021: Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2021.

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

[14] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "GraphCodeBERT: Pre-training code representations with data flow," in *ICLR 2021: Proceedings of the 9th International Conference on Learning Representations*, 2021.

[15] "Probabilistic and systematic coverage of consecutive test-method pairs for detecting order-dependent flaky tests - Tools and dataset," 2024, https://sites.google.com/view/tuscan-squares-probabilities.

[16] S. Fatima, T. A. Ghaleb, and L. Briand, "Flakify: A black-box, language model-based predictor for flaky tests," *IEEE Transactions on Software Engineering*, 2023.

[17] "Make −−random the default in rspec 3," 2024, https://github.com/rspec/rspec-core/issues/635.

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

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

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

[21] "Ranking relevant rests for order-dependent flaky tests," 2024, https://sites.google.com/view/ranking-od-relevant-tests.

[22] "Apache Dubbo," 2024, https://github.com/apache/dubbo.

[23] 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.

[24] Y. Guo, Y. Zheng, M. Tan, Q. Chen, J. Chen, P. Zhao, and J. Huang, "Nat: Neural architecture transformer for accurate and compact architectures," *Advances in Neural Information Processing Systems*, 2019.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

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

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

[28] "srcML," 2024, https://www.srcml.org/.

[29] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," *CoRR*, 2018.

[30] "PreTrainedTokenizerBase batch_encode_plus," 2024, https://huggingface.co/docs/transformers/v4.33.3/en/internal/tokenization_utils#transformers.PreTrainedTokenizerBase.batch_encode_plus.

[31] D. Ingle, R. Tripathi, A. Kumar, K. Patel, and J. Vepa, "Investigating the characteristics of a transformer in a few-shot setup: Does freezing layers in RoBERTa help?" in *BlackBoxNLP 2022: 5th BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, 2022.

[32] A. F. Agarap, "Deep learning using rectified linear units (ReLu)," *arXiv preprint arXiv:1803.08375*, 2018.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, 2014.

[34] X. Liang, X. Wang, Z. Lei, S. Liao, and S. Z. Li, "Soft-margin softmax for deep classification," in *ICONIP 2017: Proceedings of the the 24th International Conference on Neural Information Processing*, 2017.

[35] Z. Zhuang, M. Liu, A. Cutkosky, and F. Orabona, "Understanding AdamW through proximal methods and scale-freeness," *Transactions on Machine Learning Research*, 2022.

[36] H. Yao, D.-l. Zhu, B. Jiang, and P. Yu, "Negative log likelihood ratio loss for deep neural network classification," in *in FTC 2019: Proceedings of the 4th Future Technologies Conference*, 2019.

[37] R. Caruana, S. Lawrence, and C. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," *Advances in neural information processing systems*, 2000.

[38] S. Habchi, M. Cordy, M. Papadakis, and Y. L. Traon, "On the use of mutation in injecting test order-dependency," in *MSR 2021: 18th Working Conference on Mining Software Repositories*, 2021.

[39] H. Wang, Q. Liang, J. T. Hancock, and T. M. Khoshgoftaar, "Feature selection strategies: a comparative analysis of SHAP-value and importance-based methods," *Journal of Big Data*, 2024.

[40] J. Ramos, "Using TF-IDF to determine word relevance in document queries," 2003, https://api.semanticscholar.org/CorpusID:14638345.

[41] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *ASE 2013: Proceedings of the 28th Annual International Conference on Automated Software Engineering*, 2013.

[42] "GPT-3.5 Turbo," 2024, https://platform.openai.com/docs/models/gpt-3-5-turbo.

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

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

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

[46] A. Bandyopadhyay, "Improving spectrum-based fault localization using proximity-based weighting of test cases," in *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering*, 2011.

[47] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE 2002: Proceedings of the 24th International Conference on Software Engineering*, 2002.

[48] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, 2005.

[49] S. Ali, J. H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," in *ASE 2009: Proceedings of the 24th Annual International Conference on Automated Software Engineering*, 2009.

[50] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *JSS*, vol. 82, no. 11, 2009.

[51] S. Fatima, H. Hemmati, and L. Briand, "FlakyFix: Using large language models for predicting flaky test fix categories and test code repair," *arXiv preprint arXiv:2307.00012*, 2023.