# PEELER: Learning to Effectively Predict Flakiness without Running Tests

Yihao Qin*, Shangwen Wang*, Kui Liu†, Bo Lin*, Hongjun Wu*, Li Li‡,
Xiaoguang Mao*, Tegawendé F. Bissyandé§
*National University of Defense Technology, China,
{yihaoqin, wangshangwen13, linbo19, wuhongjun15, xgmao}@nudt.edu.cn
†Huawei Software Engineering Application Technology Lab, China, brucekuiliu@gmail.com
‡Monash University, Australia, li.li@monash.edu
§University of Luxembourg, Luxembourg, tegawende.bissyande@uni.lu

***Abstract*—Regression testing is a widely adopted approach to expose change-induced bugs as well as to verify the correctness/robustness of code in modern software development settings. Unfortunately, the occurrence of flaky tests leads to a significant increase in the cost of regression testing and eventually reduces the productivity of developers (i.e., their ability to find and fix *real* problems). State-of-the-art approaches leverage dynamic test information obtained through expensive re-execution of test cases to effectively identify flaky tests. Towards accounting for scalability constraints, some recent approaches have built on static test case features, but fall short on effectiveness. In this paper, we introduce PEELER, a new fully static approach for predicting flaky tests through exploring a representation of test cases based on the data dependency relations. The predictor is then trained as a neural network based model, which achieves at the same time scalability (because it does not require any test execution), effectiveness (because it exploits relevant test dependency features), and practicality (because it can be applied in the wild to find new flaky tests). Experimental validation on 17,532 test cases from 21 Java projects shows that PEELER outperforms the state-of-the-art FlakeFlagger by around 20 percentage points: we catch 22% more flaky tests while yielding 51% less false positives. Finally, in a live study with projects in-the-wild, we reported to developers 21 flakiness cases, among which 12 have already been confirmed by developers as being indeed flaky.**

***Index Terms*—Flaky tests, Deep learning, Program dependency**

## I. INTRODUCTION

Regression testing [1] has been widely adopted in software maintenance as part of the quality assurance due diligence when changes are applied to software. When a previously-passing test case fails, the developer is prompted to check the code change in order to address the introduced bug. Unfortunately, some tests can pass or fail in a non-deterministic way (i.e., unrelated to the code change) and are now colloquially referred to as *flaky tests*. Their numbers and occurrences in industrial settings have made them an important concern for the entire field of software testing.

Software testing professionals at Google have reported that almost 16% of their 4.2 million tests have some level of flakiness [2]. A follow-up report in 2019 indicated that flaky tests have given rise to 84% of conversions from passing to failing in Google continuous integration (CI) testing [3]. Microsoft deemed flaky tests as one of the most important reasons that impede software deployment period [4]. Beyond these industrial cases, flaky tests were also found to appear among automatically-generated tests by famous tools such as Randoop [5], Evosuite [6], etc.

Following up on the empirical analysis presented by Luo *et al.* [7], software testing researchers have invested significant effort into proposing approaches for mitigating the negative effects of flaky tests [8], [9], [10], [11]. In particular, detecting flaky tests is a research objective that is gaining momentum. Till now, several frameworks have been proposed to detect flaky tests by performing test reruns in various environments [12], [13], [14], [15], [16], [17]. Other dynamic approaches have been proposed where flaky tests are detected by monitoring test coverage [18], instrumenting programs [4] or varying execution orders [19], [20]. Because such dynamic approaches are expensive in terms of time and computational cost, static approaches are also being considered: Pinto *et al.* [21] extracted code vocabularies from test code and used machine learning algorithms to predict flaky tests based on the assumption that flaky test code follows certain grammatical patterns. Finally, a recent state of the art, FlakeFlagger [22], adopts a hybrid approach where a machine learning model is trained based on a combination of static test code features and features inferred from run-time testing data.

Despite growing research around flaky tests, flakiness detection is still at its infancy. On the one hand, dynamic approaches have shown their limitations since even multiple reruns may not reveal flakiness. For some programs, even 10K re-runs could not ensure the absence of flaky tests [22]. On the other hand, while static and hybrid approaches are relatively less costly, they achieve limited performance: FlakeFlagger and the vocabulary-based approach yielded 60% and 11% precision scores respectively [22]. Concretely, this means that dynamic

*Shangwen Wang and Kui Liu are the corresponding authors.
This work is partially supported by the National Natural Science Foundation of China (Grant No. 61872445, 61672529, 62172214), the National Key R&D Program of China (No. 2020AAA0107704), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06), as well as the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949014 for project NATURAL).

approaches may not uncover flaky tests while static/hybrid approaches will raise false alarms of flakiness in a large number of test cases.

**This paper** aims at improving the detection performance of flakiness while guaranteeing scalability. We thus focus on proposing a fully static approach (i.e., no test case is run), and develop a machine learning approach which explores features that are rich and relevant towards determining flakiness. Our intuition stems from the observation of a simple phenomenon in test case assertions: variables involved in such statements have a dependence relationship to code in the test case or to the code under test (CUT). We have observed from flaky tests examples that flakiness can be discussed in terms of these inner-test (i.e., within the test case) and test-CUT (i.e., between the test code and code under test) dependencies. So, our main intuition is that we should leverage such dependencies to characterize test cases and learn to predict flakiness.

We propose PEELER, a fully static, Program dEpendEncy based, fLakiness dEtectoR. For the design of PEELER, we define the concept of the test dependency graph (TDG), which, given a test case, is capable of capturing data dependency relations both inner-test (i.e., between statements within a test case) and test-CUT (i.e., across the test code and the CUT through function calls). Then, we extract a bag of *contextual paths* which can model both the inner-test and test-CUT data dependency relations for each variable that appears in the *assertion* statements of the test case. These paths are then embedded and fed into a deep learning architecture to build a flakiness binary predictor.

We assess the effectiveness of PEELER on a large-scale dataset that was previously used in the evaluation of FlakeFlagger [22]. On the 17,532 test cases (21 java projects) in the benchmark, PEELER significantly outperforms the state of the art: its F-score, which is established at over 80%, is higher by nearly 30 percentage points than the performance achieved by FlakeFlagger. While being fully static, PEELER can catch 22% more flaky tests (583 vs. 476) while yielding 50% less false positives (172 vs. 348) than FlakeFlagger, which is a hybrid approach requiring test execution. Finally, using PEELER predictions, we already helped developers successfully identify 12 flaky tests in 3 open-source projects in the wild.

The main contributions of this paper are:

- We propose to leverage contextual information in the form of a test dependency graph (TDG) to model the data dependencies within a test case and between the test case and code under test.
- We design PEELER, a neural network based approach that embeds contextual paths extracted from the TDG to build a representation of test cases and trains a classifier as flaky test detector. PEELER is open sourced at: **https://github.com/IntHelloWorld/Peeler**.
- We perform extensive experiments to assess the performance of PEELER on 17,532 test cases collected from 21 Java projects. Experimental results confirm that PEELER outperforms the state of the art both in terms of precision (less false positives) and recall (higher flakiness detection).

```
1  public void testRequestMetaForSuccessfulRequest()
2      throws Exception {
3    ......
4    String content =
5        fetch("http://example.com/request-meta");
6    RequestMeta requestMeta =
7        RequestMeta.fromJSON(content);
8    requestMeta.getHeaders().remove("Via");
9    requestMeta.getHeaders().remove("Cache-Control");
10   content = requestMeta.toJSON();
11 -  corporaAsserter.assertEquals(
12 -      content,
13 -      "testRequestMetaForSuccessfulRequest");
14 +  JSONAssert.assertEquals(
15 +      content, false,
16 +      corporaAsserter.getCorporaCache().read(
17 +          "testRequestMetaForSuccessfulRequest"))
18 }
```

**Listing 1:** Example of a fixed flaky test.

## II. MOTIVATION

The state-of-the-art in flakiness detection has so far focused on leveraging code metrics such as the number of lines of code in a test case (e.g., [22]), or on exploiting information in code identifiers (e.g., [21]). While these approaches have achieved some level of effectiveness, many flaky tests are still challenging to detect with high-level feature engineering. Our main intuition is that data dependency relations in test code can be leveraged to improve the detection of flakiness. To motivate our approach, we present two real-world examples of test flakiness where dependency with the test case (i.e., Inner-Test) and dependency between the test case and the code under test (i.e., Test-CUT) provide hints of flakiness.

**Example#1 - Inner-Test Dependency.** Listing 1 illustrates the case of a flakiness fix for a test case in the *spinn3r* project.[1] In lines 11-13, we note that the flaky test uses a `corporaAsserter` object to assert whether the fetched `content` value is equal to the string `"testRequestMetaForSuccessfulRequest"`. The `content` object is declared as `String` (line 4-5). However, its value, when the assertion instruction is executed, is returned from `requestMeta.toJSON()` (line 10) that returns a string by converting a character stream, collecting inputs from the `JSON` object (cf., line 6-7). The assert at line 11 may fail since the returned `JSON` string of `requestMeta.toJSON()` may have a character order that is inconsistent, i.e., different from the expected string `"testRequestMetaForSuccessfulRequest"` in some cases. In the fixed code (lines 14-17), the expected string is first parsed into a `JSON` string. The assert is then performed on the adapted `JSONAssert` which checks the logical structure and data and can be parameterized (with the third boolean argument set to `false`) to forgive reordering data and extending results, "making tests less brittle".[2]

In this example, given only the `assert` statement (i.e., line 11), it is difficult to understand the flakiness of this test, which should consider the context of the variable `content`

[1] https://github.com/spinn3r/noxy/pull/21.
[2] https://github.com/skyscreamer/JSONassert.

```
1  public void giteeSample() throws Exception {
2    // See https://git.mydoc.io/?t=154711
3    Map<String,Object> value =
4        new ObjectMapper().readValue(
5            new ClassPathResource(
6                "pathsamples/gitee.json")
7                    .getInputStream(),
8            new TypeReference<Map<String,Object>>(){}
9        );
10   this.headers.set(
11       "x-git-oschina-event", "Push Hook");
12   PropertyPathNotification extracted =
13       this.extractor.extract(this.headers, value);
14   assertThat(extracted).isNotNull();
15   assertThat(extracted.getPaths()[0])
16       .isEqualTo("d.txt");
17 }
```

**Listing 2:** Example of a flaky test.

(i.e., that it is a `JSON` string). Fortunately, this context can be readily inferred from the code from line 4 to line 10, and be represented as program dependency relations in the test code (`content` → `fromJSON()` → `toJSON()`). Upon extracting such dependency relations, it is trivial to determine the flakiness of this test since the flaky test does not use an appropriate API to test the `JSON` string. Nevertheless, existing approaches do not consider data dependencies within a test case, and thus cannot be expected to detect such flaky test cases.

**Example#2 - Test-CUT Dependency.** Listing 2 illustrates the case of a flaky test encountered in the project *spring-cloud*.[3] The flakiness is related to the implementation of the code under test (CUT). On line 13 of the test case (`giteeSample`), a call is made to function `extract()`, which is defined in the CUT within the class `BasePropertyPathNotificationExtractor` (cf. Listing 3) . In the original `extract()` (i.e., before flakiness is fixed), the `paths` variable, which is a Set, is initialized on Line 7 as a `HashSet` object, which is later converted to an array (line 15). Unfortunately, the `HashSet` data type cannot guarantee the order of entries in the set. Therefore, the assertion in the test case (line 15-16 in Listing 2) will fail from time to time since it asserts that the first element in the set must always be `"d.txt"`. To fix this flakiness issue, developers did not change the test case but rather the CUT: they replaced the usage of `HashSet` with `LinkedHashSet`. This ensures that the function `extract()` called in the test case will return an array where items are deterministically ordered.

With this example, we observe that the flakiness of a test could depend on the source code files that have dependency relations with the test. Therefore, we propose to taking the test-CUT dependency into consideration for assessing flakiness.

## III. Approach

Figure 1 overviews the basic steps in our approach. We take as input a test case method and the source code of the associated code under test (CUT). A first module, PathExtractor, is

[3]https://github.com/spring-cloud/spring-cloud-config/pull/1546.

```
1  public PropertyPathNotification extract(
2      MultiValueMap<String,String> headers,
3      Map<String,Object> request) {
4    if (requestBelongsToGitRepoManager(headers)) {
5      if (request.get("commits")
6          instanceof Collection) {
7  -     Set<String> paths = new HashSet<>();
8  +     Set<String> paths = new LinkedHashSet<>();
9        Collection<Map<String, Object>> commits =
10           (Collection<Map<String,Object>>) request
11               .get("commits");
12       addPaths(paths, commits);
13       if (!paths.isEmpty()) {
14         return new PropertyPathNotification(
15             paths.toArray(new String[0]));
16       }
17     }
18   }
19   return null;
20 }
```

**Listing 3:** A patch of the source code for the flaky test in Listing 2.

designed to construct the test dependency graph (TDG) from which contextual paths will be extracted (cf. Section III-A). Then, each extracted path will be embedded with the help of a pre-trained *code2vec* model and an attention mechanism is used to combine these paths into the representation of the test. Finally, a one-layer neural network will be fed with the representation of the input test method to identify whether this test is a flaky one (cf. Section III-B).

### A. PathExtractor

Given a test case and its corresponding code under test, PEELER first generates a test dependency graph (TDG), which aims to reveal the data dependency relations in a test. As argued by recent studies [23], [24], such graphs are overloaded with information that may not all be critical. Informed by our empirical observations, we propose to focus on the variables which possess data dependencies with the assertion statements in a test. Specifically, we aim to extract the *contextual paths* for each assertion statement, which connect the variables that have dependency relation with it in the graph. In this way, the information from the graph with limited usefulness for determining the flakiness of a test can be discarded (e.g., lines 4-6 in Listing 1). We next give definitions for TDG and the contextual path, after which we introduce how we extract these paths.

*Definition 1.* **Test Dependency Graph (TDG):** A TDG is defined as $TDG = (N, E, \mu, \sigma, \gamma)$ to represent a test method, where $N$ is a set of nodes in the graph and $E$ is a set of edges, $N = \{n_0, n_1, ..., n_k\}$, $E = \{e_0, e_1, ..., e_l\}$, $e = (n_i, n_j)$. Function $\mu$ assigns nodes with line number positions in their code text, function $\sigma$ denotes the relationships between nodes and edges, and function $\gamma$ points out whether an edge is related to the code under test.

**Nodes** The nodes $N$ are the abstract syntax tree (AST) entities of code presented in $TDG$, which are limited on three kinds of the AST entities (i.e., variables, literals, and function calls that will return specific values). For a line of code `Tool tool = Tool.load( "string", get(item), array[10])`, the nodes are the AST en-

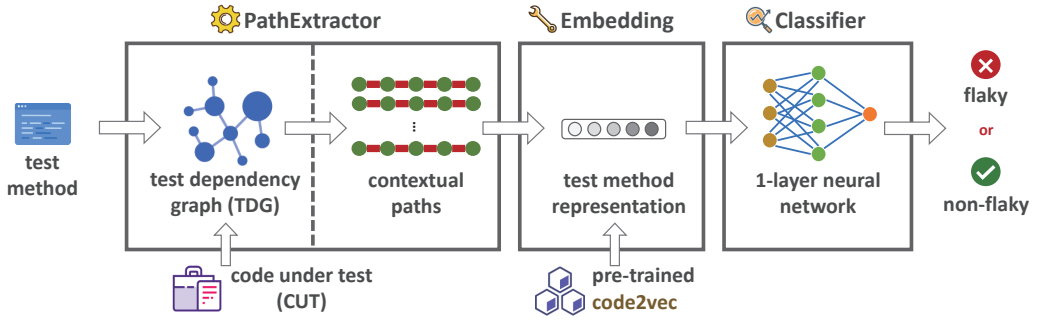
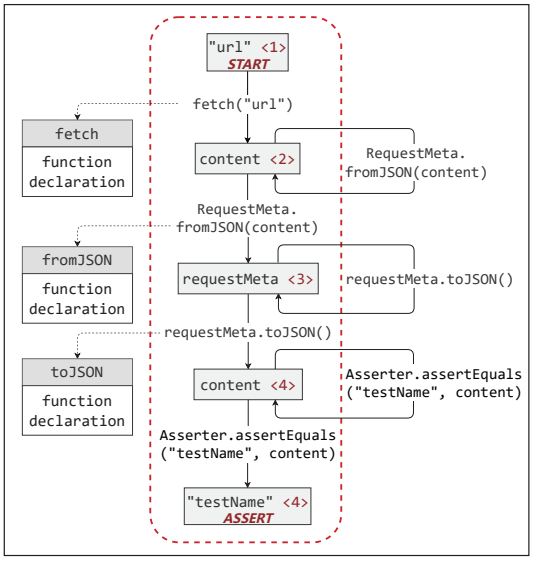



**Fig. 1:** Overview of PEELER.

```
1: String content = fetch("url");
2: RequestMeta requestMeta =
            RequestMeta.fromJSON(content);
3: content = requestMeta.toJSON();
4: Asserter.assertEquals("testName", content);
```

(a) Test code




(b) TDG

**Fig. 2:** Example of a test code and the generated TDG.

tities with the code content "`tool`, `"string"`, `item`, `get(item)`, and `array`". Function $\mu : N \leftarrow L$ assigns each node with the corresponding line number ($l \in L$) in code text to represent its position. If the literal nodes and variable declaration nodes are presented for the first time, they will be labeled with *START*. In addition, if an AST entity is extracted from an assert statement, its node will be labeled with *ASSERT*.

**Edges** Edges denote all of the data transfer relations from node $n_i$ to node $n_j$. For the code listed before, the object `tool` is initialized through the function `Tool.load(...)`, edges `("string", tool)`, `(item, get(item)`, `(get(item), tool)`, `(item, tool)`, and `(array, tool)` can be extracted. Function $\sigma : E \leftarrow O$ denotes the direct data transfer operation $o \in O$ (e.g., the function call and arithmetic operators) working on an edge to describe the semantic behavior of the edge. In the previous code, the function call `Tool.load(...)` is the data transfer

**Algorithm 1:** Extracting contextual paths.

**Input:** TC: the test code.
**Input:** CUT: the code under test.
**Output:** $CtxtP$: A set of contextual paths for the test code.

$CtxtP = \emptyset$;
$CtxtP' = \emptyset$ ; /* *Temporary set of contextual paths.* */
$AST$ = parse(TC) ; /* *Parse test code into AST.* */
$TDG$ =TDGGenerator($AST$) ; /* *Generate TDG from AST.* */
Sort($E$) ; /* *Sort in descending order by line numbers.* */
**for** $e_i$ *in* $E$ **do**
  **if** $e_i.n_j$ *is a ASSERT node* **then**
    $ctxtP = (n_i \rightarrow e_i \rightarrow n_j)$;
    **if** $e_i.o$ *is from CUT AND* $\gamma(e_i)$ *is null* **then**
      /* *Labeling* $e_i$ *is related to CUT.* */
      $\gamma$: $e_i \leftarrow p$;
    $CtxtP'$.add($ctxtP$);
    **while** $CtxtP'$ *is not null* **do**
      **for** $ctxtP'$ *in* $CtxtP'$ **do**
        $CtxtP'$.remove($ctxtP'$);
        /* *Find edges connected to the path* $ctxtP'$. */
        $E'$ = findConnectedEdges($ctxtP'$, E, $e_i$);
        **if** $E'$ *is null* **then**
          $CtxtP$.add($ctxtP'$);
          continue;
        **for** $e' = (n'_l, n'_r)$ *in* $E'$ **do**
          $\_ctxtP'$ = $ctxtP'$;
          $\_ctxtP'$.concatenate($n'_l$, $e'$);
          **if** $e'.o$ *is from CUT AND* $\gamma(e')$ *is null* **then**
            $\gamma$: $e' \leftarrow p$;
          $CtxtP'$.add($\_ctxtP'$);
Return $CtxtP$

operation of those edges. Note that, the functions called in test code could be defined in the code under test. To distinguish such function calls from others, the function $\gamma : E \leftarrow P$ is to point out that the function call is defined in the code under test for the related edge. Note that we only consider the calls made directly from the CUT.

*Definition 2.* **Contextual Path:** A contextual path $ctxtP$ is a sub-path in $TDG$ that denotes the complete path between two connectable nodes, $ctxtP = (n_1 \rightarrow e_1 \rightarrow n_2 \rightarrow e_2 \rightarrow \cdots \rightarrow n_{l-1} \rightarrow e_{l-1} \rightarrow n_l)$, where $\{n_1, \cdots, n_l\} \subseteq N$, $n_1$ is a *START* node, $n_l$ is an *ASSERT* node, $\{e_1, \cdots, e_{l-1}\} \subseteq E, e_i = (n_i, n_{i+1}), i \in [1, l-1]$.

**Workflow of the PathExtractor.** PathExtractor first constructs the test dependency graph (TDG) for the test code, which is used to construct contextual paths by concatenating node sequences in the graph. Algorithm 1 illustrates the details.

Taking as input the test code (TC), PathExtractor applies *JavaParser* [25] to parse the code into abstract syntax tree (AST) (line 3), and visits AST nodes in terms of the depth-first search to generate the $TDG$ (line 4), where the nodes and edges are identified according to the previous definitions. Once $TDG$ is generated, the edge set $E$ is then sorted in descending order by line position of the second node $n_y$ of each edge $e = (n_x, n_y) \in E$ (line 5), to optimize the process of constructing contextual paths. For each edge $e_i = (n_i, n_j)$ in $E$, if the node $n_j$ was signed with the label *ASSERT*, a single contextual path $ctxtP$ linking $n_i$ and $n_j$ through $e_i$ will be constructed (lines 7-8), and the edge $e_i$ will be assigned with the function declaration code $p$ if the function call of $e_i$ is defined in the code under test (lines 9-11). To deal with the polymorphism, the parameter information is taken into account in this step. After such a path is constructed, we search for all edges $E'$ that are connected with the path through the node $n_i$ (line 17). To optimize the search space, the searching starts from the next edge of $e_i$ since each $e_i$ will not be connected by the path before $ctxtP'$. If $E'$ is empty, the construction of the contextual path $ctxtP'$ is finished (lines 18-20). Otherwise, each edge in $E'$ will be used to construct a new contextual path by concatenating the node $n'_l$ and the edge $e'$ with the path $ctxtP'$ (lines 21-26). All of the newly constructed contextual paths $CtxtP'$ will be further used to continuously concatenate the corresponding nodes and edges. It should be noted that each contextual path extracted from such a procedure starts with a *START* node.

Figure 2 illustrates the test dependency graph with an example of a code fragment (Figure 2(a)) excerpted from Listing 1. PathExtractor traces variable `content` to generate the TDG shown in Figure 2(b). In the TDG generated for the code fragment, the nodes are illustrated with grey rectangles, that are assigned with line numbers and *START/ASSERT*. The edges are presented through solid arrows, which are labeled with function calls. We use dotted arrows to assign function declarations from the code under test to the corresponding edges. Once a TDG is constructed, PathExtractor will then follow its workflow to extracted *contextual paths* from the TDG. The longest path (marked with red dotted-line framework in Figure 2(b)) in TDG will be generated as the contextual path for the assert statement in line 4 of the test code snippet.

### B. Test Embedding Model

Considering the diversity of the contextual paths, it is non-trivial to manually summarize heuristics for identifying test flakiness based on the contextual paths. Instead, we decide to use a neural network (shown in Figure 3) to represent the semantics of the tests and thus predict the flakiness.

***Path embedding.*** Suppose the extracted contextual paths set for a specific test is $CtxtP = \{ctxtP_1, \ldots, ctxtP_y\}$. In order to embed a contextual path $ctxtP_j$ with a continuous vector, we first need to embed its elements (i.e., nodes and edges). To this end, we employ a pre-trained code2vec [26] model that was proposed to extract semantic features from the AST paths of code functions and was trained on a large corpus containing +12M methods.

For each contextual path $ctxtP_j = (n_1 \rightarrow e_1 \rightarrow n_2 \rightarrow e_2 \rightarrow \cdots \rightarrow n_{l-1} \rightarrow e_{l-1} \rightarrow n_l)$, we first convert it into a sequence of tuples $\{(n_1, e_1), (n_2, e_2), \cdots, (n_l, PAD)\}$, where $PAD$ is the padding token. Each node $n_i$ is also split into several tokens by the characters in the string of the node (e.g., "." and "("). And each token is divided into sub-tokens based on camel case and under score naming conventions. All sub-tokens are embedded into representing vectors by utilizing the pre-trained embedding matrix from code2vec. Finally, the vectors of all sub-tokens from the node $n_i$ are averaged to represent the embedded node $v_i^n \in \mathbb{R}^d$.

As for the edge $e_i$, if $e_i$ cannot map with any code under test, the edge is embedded using the same way as embedding nodes with the assigned function call or arithmetic operator. Otherwise, the code under test will be fed into the pre-trained code2vec model and the output vector of this method is considered as the representation of the edge $v_i^e \in \mathbb{R}^h$. Eventually, the two embeddings of node $n_i$ and edge $e_i$ are concatenated to a single vector: $v_i = [v_i^n; v_i^e] \in \mathbb{R}^{d+h}$ that represents the $i_{th}$ tuple in $ctxtP_j$.

To represent the contextual path $ctxtP_j$, the tuples $t = \{v_1, v_2, \ldots, v_l\}$ are first conveyed through a fully connected layer for extracting more features:

$$v'_i = tanh(LN(W \times v_i))$$

where $v'_i$ is the resulting vector computed for each tuple vector $v_i$, $W$ is a learned matrix, $LN$ denotes the layer normalization for stabilizing the layer input, and $tanh$ is the hyperbolic tangent function, which is commonly used as the activation function for increasing the ability of model expression [27].

We recall that each contextual path has its beginning node (which is labelled as $START$) and its end node (which is labelled as $ASSERT$) and the data is transferred in order in the path. Therefore, to capture such sequential features, the obtained vectors ($v'_1$, …, $v'_l$) are sent into a 2-layer unidirectional LSTM model:

$$c_j = LSTM(v'_1, \ldots, v'_l)$$

The final hidden state of the second layer of the LSTM model is used to represent the path $ctxtP_j$, which is denoted as $c_j$. We apply an LSTM of 2 layers by following the previous code embedding study [28]. We decide to adopt the unidirectional setting as the data dependency is transferred unidirectionally in the contextual path.

***Path attention mechanism.*** After obtaining the representation of each path $ctxtP_i$, we need to combine them to represent the whole test method. Inspired by previous studies [28], [29], we choose to adopt a path attention mechanism since different paths may contribute unequally to the semantic of the test method. This process can be described as:

$$\beta_i = \frac{\exp(c_i^T \otimes \mathbf{a})}{\sum_{j=1}^{n} \exp(c_j^T \otimes \mathbf{a})} \quad z = \sum_{i=1}^{n} \beta_i \cdot c_i$$

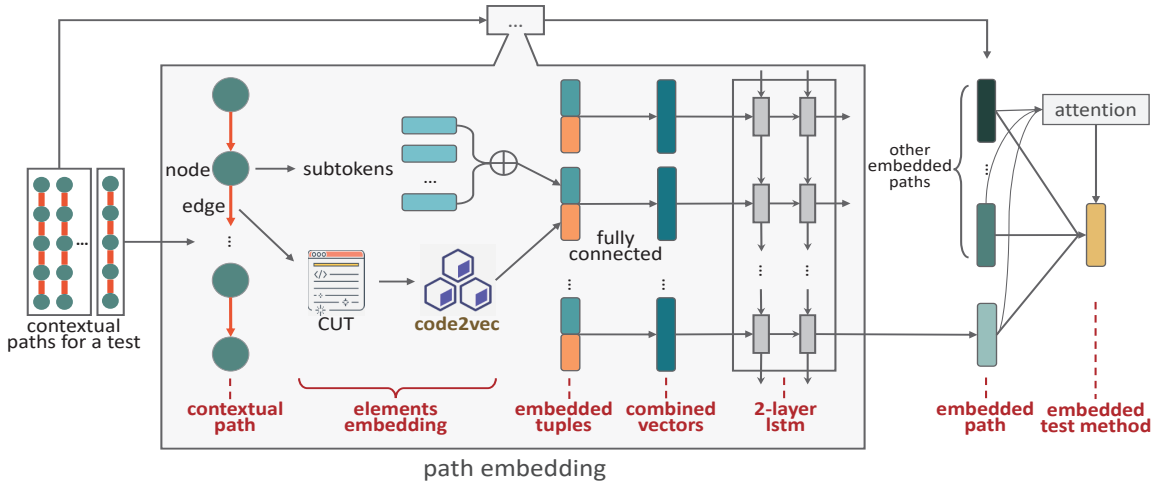



**Fig. 3:** Neural network architecture for embedding a test case.

where $\mathbf{a}$ is a randomly initialized attention parameter vector which is learned with the network, $\otimes$ is the normalized inner product, and $z$ is the output vector of the attention layer which is a numerical representation of a single test method.

### C. Flakiness Classifier

The resulting vector $z$ from the previous step is further sent into a fully connected layer: $z' = W_1 \times z$ where $W_1$ is a learned matrix for feature reshape. After that, $z'$ is sent into another fully connected layer for dimension reduction: $o = W_2 \times z'$. Finally, we use the *softmax* function for predicting the final results: $v_{out} = softmax(o)$. Since our task is a binary classification problem, the dimension of the final output of the model is 2. We adopt the cross-entropy criterion [30] as our loss function for training the networks. The loss can be calculated as:

$$\text{loss} = -\frac{1}{N}\sum_{i=1}^{N}\log\left(\frac{\exp(x_i[label_i])}{\exp(x_i[0]) + \exp(x_i[1])}\right)$$

where $x = [x_1, x_2, \ldots, x_N] \in \mathbb{R}^{2\times N}$ is the output of the model correspond to a batch of test methods, $N$ is the batch size, $label_i$ denotes the oracle classification of the test $x_i$, which equals to 1 if the test method is a flaky test and 0 otherwise; $x_i[0]$ and $x_i[1]$ denote the predicted probability of the test $x_i$ to be non-flaky and flaky, respectively.

## IV. Evaluation Design

### A. Research Questions

- **RQ1:** ***Is*** PEELER ***effective in detecting flaky tests in real-world Java projects?*** This RQ aims to build a new effectiveness baseline for researchers working on detecting flaky tests statically.
- **RQ2:** ***To what extent do the inner-test and test-CUT contextual dependency relations affect the performance of*** PEELER***?*** This RQ could help understand the individual contribution from the two types of dependency information, and thus facilitate future studies.
- **RQ3:** ***Can*** PEELER ***detect flaky tests in the wild?*** This RQ helps potential users of PEELER understand how to apply it in practice and further how useful it could be in practice.

### B. Dataset

In order to train our model, we require a dataset including both flaky tests and non-flaky tests. Such a dataset has fortunately been proposed recently by Alshammari *et al.* [22]. The authors considered 21,734 tests from 23 open-source Java projects and extensively re-ran each test 10K times to decide on flakiness (each execution is independently performed). Overall, they identified 808 test cases as flaky, among which 96 test cases have been already addressed by the developers. We chose to use this dataset because (1) it is an authentic dataset containing a large number of flaky tests in the community, and (2) it has been leveraged by two recent state-of-the-art approaches (i.e., FlakeFlagger [22] and the vocabulary-based approach [21], which we will compare against), to support their experimental evaluations. Note that, due to the limitation of PathExtractor, we failed to extract any contextual paths from some tests, hence, we had to discard them from our experiments (we will discuss this situation later). The final dataset contains 17,532 tests from 21 projects, of which 689 are flaky.

### C. Training and Testing

We randomly split the benchmark into 10 same-size folds (i.e., with identical numbers of flaky and non-flaky tests in each fold). We thus have 620/15 159 (69/1 684) flaky/non-flaky tests in the training (test) set of each fold. We then applied 10-fold cross validation to evaluate our model, following the experimental setup by Alshammari *et al.* [22]. The final performance of PEELER is summed up over the 10 rounds of each training and testing process. Note that our benchmark is imbalanced (the number of non-flaky tests is much larger than that of flaky ones), we thus utilized the re-sampling method [31] by sampling the identical number of flaky and non-flaky tests, and combined them into each batch during training. The model was trained on Ubuntu18.04 system with 256G RAM, AMD-3970x CPU, and GeForce RTX3090 graphics card.

**TABLE I:** The hyper-parameters we use for training our network.

| parameter | epoch size | batch size | max path length | max path count | learning rate |
|---|---|---|---|---|---|
| value | 20 | 64 | 20 | 100 | 0.01 |

As for the hyper-parameters of our model, two critical ones are the maximum number of paths from each test and the maximum length of each path. To avoid the input information being too large, we set the max path count to 100 and the max path length to 20 for each test method. This decision is based on our statistics that on average, the test in our dataset contains 36 contextual paths and the length of each path is 16. Other hyper-parameters about the network were set based on values in previous works [26], [28]. The parameters we ultimately used in our model are shown in Table I.

### D. Metrics

Following prior studies [22], [21], we assess PEELER's effectiveness based on precision, recall, and F-score metrics. Given the following:

**True Positive (TP):** # of flaky tests identified as flaky;
**False Positive (FP):** # of non-flaky tests identified as flaky;
**False Negative (FN):** # of flaky tests identified as non-flaky;
**True Negative (TN):** # of non-flaky test identified as non-flaky.

The proposed metrics are computed as:

$$Precision = TP/(TP + FP) \quad Recall = TP/(TP + FN)$$
$$F - score = 2 \times Precision \times Recall/(Precision + Recall)$$

## V. EXPERIMENTAL RESULTS

### A. RQ1: Detection Performance

Performance results of PEELER on the flaky tests dataset are shown in Table II. Overall, PEELER achieves reasonable performance: it can detect 85% of all flaky tests in the dataset with a relatively high precision (i.e., 77%). The trade-off between precision and recall leads to an overall F-score value that exceeds 80%.

To investigate how it performs against state of the art, we select two flaky test classifiers *FlakeFlagger* [22] and Pinto *et al.*'s approach [21] based on code vocabulary as our baselines since: (0) they are recent in the literature (ICSE 2021 and MSR 2020); (1) we use the same dataset as them for the evaluation; and (2) both approaches target scalability (i.e., not requiring to re-run tests many times to decide on flakiness). Recall that we have introduced how these baselines work in Section I, and that our scalability constraints are higher: PEELER **does not even need to run any tests**.

The comparison results between PEELER and the baselines are presented in Table II. Note that the performances of baselines are provided by Alshammari *et al.* [22]. Overall, PEELER presents a better performance of detecting flaky tests than the two state-of-the-art baseline approaches (i.e., FlakeFlagger/the vocabulary-based one) with higher scores of precision, recall and F-score, respectively. For instance, PEELER correctly detects 583 flaky tests out of 689 ones while the baselines detect 476 and 416 respectively, leading to an increase with 22%/40% of them. As for the false positives, PEELER false positively identifies 172 non-flaky tests as flaky, which is a decrease with 51%/95% false positives of the two baselines.

According to the hypothesis proposed by Alshammari *et al.* [22], supposing that a developer is using the detection results from PEELER, she only needs to re-run 755 tests to expose the flakiness and she can find 583 flaky ones. If she uses FlakeFlagger or the vocabulary-based one, she can only identify 476 or 416 by re-running 824 or 3,569 tests which will consume much more time for her than PEELER. From this perspective, PEELER is more user-friendly with respect to time consumption.

We also investigated the distributions of the flaky tests detected by different approaches and the results are shown in Figure 4. The figure illustrates the complementarity of PEELER with existing approaches on detecting different flaky tests. Specifically, 49 flaky tests can be uniquely detected by PEELER while the numbers for FlakeFlagger and the vocabulary-based approach are 13 and 24 respectively.

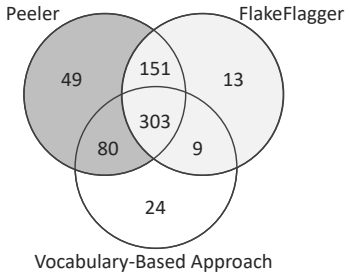


**Fig. 4:** Overlaps of detected flaky tests by different approaches.

When it comes to the individual project, we find the performance of PEELER varies among different projects. On some projects such as *alluxio*, all three approaches achieve good performances (e.g., the F-scores of them all exceed 90%). However, on some projects with limited known flaky tests (e.g., *assertj-core* and *handlebars.java*), all of the three approaches fail to detect any of the flaky tests, which leads to the F1-score being 0. As revealed by Alshammari *et al.*, the lack of the flaky test data is the main reason for this phenomenon [22]. Nonetheless, this problem is alleviated for PEELER since it achieves satisfactory results on other projects where the number of flaky tests is also limited (e.g., *elastic-job-lite*). In general, PEELER does not require extensive training data to yield reasonable performance. We refer the reader to more detailed discussions in Section VI.

> ➽ PEELER *can effectively detect the flaky tests in the dataset, of which results at precision, recall and F-score metrics outperform the state-of-the-art approaches with higher values by detecting more flaky tests with fewer false positives and false negatives.*

### B. RQ2: Ablation Study

For this research question, we investigate the contributions from inner-test and test-CUT contextual dependency relations to PEELER. Therefore, we separately removed the inner-test and test-CUT contextual dependency information from the contextual paths for PEELER. For the former case, we only kept the edges that are assigned with code under test and the

**TABLE II:** Results of detecting flaky tests with PEELER, FlakeFlagger, and the vocabulary-based approach.

| Project | # Tests | # Flaky Tests | PEELER | | | | **FlakeFlagger** [22] | | | | **Vocabulary-based** [21] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | **TP** | **FN** | **FP** | **TN** | **TP** | **FN** | **FP** | **TN** | **TP** | **FN** | **FP** | **TN** |
| alluxio | 187 | 116 | 116 | 0 | 0 | 71 | 116 | 0 | 0 | 71 | 108 | 8 | 6 | 65 |
| hbase | 351 | 114 | 106 | 8 | 4 | 233 | 100 | 14 | 21 | 216 | 53 | 61 | 118 | 119 |
| okhttp | 766 | 100 | 94 | 6 | 47 | 619 | 43 | 57 | 145 | 521 | 73 | 27 | 360 | 306 |
| spring-boot | 1582 | 82 | 69 | 13 | 23 | 1477 | 61 | 21 | 11 | 1489 | 48 | 34 | 484 | 1016 |
| ambari | 316 | 46 | 43 | 3 | 1 | 269 | 40 | 6 | 3 | 267 | 25 | 21 | 72 | 198 |
| hector | 130 | 33 | 28 | 5 | 3 | 94 | 30 | 3 | 11 | 86 | 7 | 26 | 14 | 83 |
| java-websocket | 145 | 23 | 20 | 3 | 5 | 117 | 19 | 4 | 1 | 121 | 22 | 1 | 67 | 55 |
| httpcore | 712 | 22 | 19 | 3 | 17 | 673 | 13 | 9 | 24 | 666 | 12 | 10 | 292 | 398 |
| wildfly | 848 | 23 | 19 | 4 | 11 | 814 | 12 | 11 | 22 | 803 | 20 | 3 | 362 | 463 |
| http-request | 161 | 18 | 17 | 1 | 0 | 143 | 12 | 6 | 5 | 138 | 15 | 3 | 62 | 81 |
| activiti | 2010 | 32 | 15 | 17 | 13 | 1965 | 10 | 22 | 45 | 1933 | 10 | 22 | 344 | 1634 |
| wro4j | 1089 | 16 | 9 | 7 | 16 | 1057 | 4 | 12 | 2 | 1071 | 2 | 14 | 96 | 977 |
| incubator-dubbo | 1443 | 19 | 10 | 9 | 6 | 1418 | 8 | 11 | 24 | 1400 | 8 | 11 | 415 | 1009 |
| logback | 787 | 22 | 8 | 14 | 5 | 760 | 2 | 20 | 11 | 754 | 6 | 16 | 213 | 552 |
| orbit | 82 | 7 | 3 | 4 | 4 | 71 | 1 | 6 | 11 | 64 | 5 | 2 | 22 | 53 |
| undertow | 151 | 5 | 3 | 2 | 11 | 135 | 3 | 2 | 6 | 140 | 1 | 4 | 39 | 107 |
| elastic-job-lite | 534 | 3 | 2 | 1 | 0 | 531 | 0 | 3 | 0 | 531 | 0 | 3 | 29 | 502 |
| achilles | 732 | 4 | 2 | 2 | 1 | 727 | 2 | 2 | 0 | 728 | 0 | 4 | 0 | 728 |
| zxing | 260 | 2 | 2 | 2 | 2 | 256 | 0 | 2 | 2 | 256 | 1 | 1 | 84 | 174 |
| assertj-core | 4939 | 1 | 0 | 1 | 3 | 4935 | 0 | 1 | 2 | 4936 | 0 | 1 | 6 | 4932 |
| handlebars.java | 307 | 1 | 0 | 1 | 0 | 306 | 0 | 1 | 2 | 304 | 0 | 1 | 68 | 238 |
| **Total** | 17532 | 689 | 583 | 106 | 172 | 16671 | 476 | 213 | 348 | 16495 | 416 | 273 | 3153 | 13690 |
| **Precision** | | | **77%** | | | | 58% | | | | 12% | | | |
| **Recall** | | | **85%** | | | | 69% | | | | 60% | | | |
| **F-score** | | | **81%** | | | | 63% | | | | 20% | | | |

*__TP__: True Positives, __FN__: False Negatives, __FP__: False Positives, and __TN__: True Negatives.

**TABLE III:** Ablation study results on the importance of contextual dependency information in PEELER.

| Models | Precision | Recall | F-score |
|---|---|---|---|
| PEELER | **77%** | **85%** | **81%** |
| PEELER w/o Inner-Test context | 41% | 71% | 52% |
| PEELER w/o Test-CUT context | 73% | 84% | 78% |

nodes that are connected by such edges. Other nodes and edges in the original contextual paths were padded with zero. While for the latter case, we ignored if the edges could be linked with any code under test and treated them sorely as tokens.

The performances of the variants of PEELER are shown in Table III. We note that when the inner-test contextual dependency information is discarded, the efficacy of PEELER decreases sharply; while when the test-CUT dependency information is excluded, the performances of PEELER only drop slightly. Specifically, the F-score is decreased from 81% to 52% when the inner-test dependency is neglected; while the F-score only drops from 81% to 78%, when the test-CUT dependency is overlooked.

To deeply understand the reasons for such phenomenon, we further defined two concepts which are *elements per path (EPP)* denoting the average number of elements (including the nodes and edges) in a contextual path, and *test-CUT elements per path (IFEPP)* denoting the average number of elements related to the code under test (including edges assigned with code under test and the nodes connected by them) in a contextual path. After calculation on the whole dataset, the values of EPP and IFEPP are 16.22 and 8.55 respectively. This indicates that if PEELER only focuses on the test-CUT dependency information, around half of the elements in a contextual path will be discarded, leading to a significant performance decrease. On the contrary, if PEELER discards the test-CUT dependency relation, the amount of the elements in a contextual path is not affected. Moreover, PEELER still can capture the semantic information of the code under test with the method names that are involved in the tokens of the edges and can reflect the aggregated behavior of the method body [32], [33], [34].

We further investigate the proportion of tests whose flakiness is related to test-CUT dependencies. For the flaky tests in our dataset, 96 of them have their patches confirmed by the developers, but only 6 of them are fixed by making code changes in its code under test (i.e., test-CUT contexts). Such a low proportion indicates that the flaky tests caused by their test-CUT contexts are not as pervasive as those whose flakiness can be addressed in the test code themselves. Therefore, we obtain a slight decrease when excluding test-CUT dependency relation for PEELER.

> ➽ *Both the inner-test and test-CUT dependencies are useful in ensuring the effectiveness of* PEELER. *Inner-Test dependencies however appear to be significantly more rewarding.*

### C. RQ3: Live Study

We investigate whether PEELER can actually help developers identify previously-unknown flaky tests in real-world projects. To that end, we conduct a live study on open-source projects. We use the model trained in RQ1 (i.e., the one with the best performance in the 10-fold training and testing) and apply it to 3 open-source projects which are not included in the dataset of RQ1. We then report the predicted flaky tests to the development teams for confirmation.

**TABLE IV:** Results of the live study.

| Project | Tests | Predicted | Reported | Confirmed | Refuted | P |
|---|---|---|---|---|---|---|
| Google/Guava[4] | 947 | 32 | 11 | 11 | 0 | 0 |
| DataflowTemplates[5] | 624 | 29 | 9 | 1 | 0 | 8 |
| Alibaba/fastjson[6] | 264 | 4 | 1 | 0 | 0 | 1 |
| **Total** | 1835 | 65 | 21 | 12 | 0 | 9 |

**Predicted**: the number of flaky tests predicted by PEELER.
**Reported**: the number of predicted flaky tests reported to their developers.
**Confirmed**: the number of reported flaky tests confirmed by their developers.
**Refuted**: the number of reported flaky tests refuted by their developers.
**P**: the number of reported flaky tests are still pending.

As presented in Table IV, PEELER predicts 65 out of 1,835 test cases as flaky (i.e., probability threshold at 0.5). Given that we cannot leave a heavy burden for developers with tens of issues that may or may not be accurate, we chose to first report tests with high prediction probabilities of being flaky, according to the returned results of PEELER. Specifically, we selected the 21 test cases with flakiness probability scores $\geq 0.7$. We took further steps to assess through dynamic tests that these tests are indeed probably flaky: we executed each predicted flaky test 100 times and reported test results to their developers. At the time of submission of this paper, 12 out of 21 test cases had already been confirmed as flaky by the projects' development teams. 9 cases of flakiness reports were still pending: developers did not yet reply to us. None of the submitted flakiness report were refuted so far. In addition, for the Guava project, developers discuss with us concurrency properties of the program which might explain some of the flakiness. They admitted that they may not have time to dig into all these test cases, but they welcome our support in fixing them as well. This suggests that automatic fixing of flaky tests would be a relevant research avenue towards addressing flakiness in a systematic manner. Our live study also helps us understand the application scenario of PEELER: once trained, it could be applicable to other software systems.

> PEELER *can help detect silent flaky tests in the wild which have not been uncovered by the developers.*

## VI. DISCUSSION

### A. PEELER *as a Code Representation Technique*

PEELER can generally be considered as a code representation technique that relies on the dependency relation. To better illustrate the rationale of the approach design, we also compared PEELER with another state-of-the-art code representation technique, code2seq [28]. Specifically, in the encoding part of code2seq, we generated a vector representation for a single test method with the attention mechanism, after which we used a fully-connected layer to predict the flakiness. We also performed 10-fold cross validation on the evaluation dataset for this model, and results show that the values of the precision, recall, and F-score are 69%, 43%, and 53%, respectively, which are lower than those of PEELER systematically. Such results show that our customized TDG can better capture the features for flakiness prediction than existing code representation techniques.

[4]https://github.com/google/guava
[5]https://github.com/GoogleCloudPlatform/DataflowTemplates
[6]https://github.com/alibaba/fastjson

```
public void
    oldSpringModulesAreNotTransitiveDependencies()
        throws IOException {
  runBuildForTask("checkSpring");
}

private void runBuildForTask(String task) {
  try {
    project.newBuild().forTasks(task)
        .withArguments(this.buildArguments)
            .run();
  } catch (BuildException ex) {
    Throwable root = ex;
    while (root.getCause() != null) {
      root = root.getCause();
    }
      fail(root.getMessage());
  }
}
```

**Listing 4:** Example of a flaky test on which PEELER didn't work.

**TABLE V:** The results of FlakeFlagger and vocabulary-based approach on tests where PEELER does not work.

| | All | TP | FN | FP | TN | Pr | R | F |
|---|---|---|---|---|---|---|---|---|
| **FlakeFlagger** | 4202 | 116 | 3 | 14 | 4069 | 89% | 98% | 94% |
| **Vocabulary-Based** | 4202 | 101 | 18 | 693 | 3390 | 13% | 86% | 22% |

### B. *Efficiency*

PEELER is fully static: it does not collect any runtime execution information. Instead, its overhead is in the extraction of test dependency graphs, the training of the model, and the prediction. For all 17,532 tests that are assessed in the 10-fold cross-validation process, PEELER took 80 minutes for path extraction, 163 minutes for training, and 430 seconds for prediction. Therefore, PEELER takes 0.86 second on average for a single test case. We foresee this as being an affordable time cost in practice. Unfortunately, the compared baselines approaches [22], [21] did not report their overhead. Nevertheless, given that they do perform some runtime executions, we can conclude that they would require more overhead, and thus are certainly less scalable.

### C. *Limitations*

As introduced in Section IV-B, we discarded 4,202 tests where we failed to extract any contextual paths. The reason is related to the working mechanism of the PathExtractor. As presented in Section III-A, PathExtractor can only extract contextual paths when there are *START* nodes and *ASSERT* nodes in the corresponding TDG. These nodes are used to denote the beginning and ending of a contextual path. If such conditions are not satisfied, the contextual paths cannot be extracted from the TDG. In Listing 4, we give a concrete example where the PathExtractor failed. In this test method, there is only one statement calling another method and all the tested behaviors are defined in the callee. There is no assert statement in the test and thus PathExtractor is not able to extract any contextual paths.

For the tests on which PEELER does not work, we respectively investigate the performance of FlakeFlagger and of the

vocabulary-based approach. The results are shown in Table V. We note that the performance of FlakeFlagger is substantially higher than that on the overall dataset (cf. Table II). By investigating the features of these 4,202 tests, we summarized the flakiness reasons as follows. First, a large proportion of the flaky tests (92/119) present test smells[7] "Indirect Testing: the test interacts with the object under test via an intermediary" and "Fire and Forget: the test launches background threads or tasks". Listing 4 shows such an example. On the one hand, it calls another method and defines all the test behavior in the callee; on the other hand, its callee launches a task in the background during execution. Therefore, this test presents both test smells. Second, all non-flaky tests do not suffer from the aforementioned two test smells, which may be a feature of the dataset. Since the uncovered test smells are captured and used by FlakeFlagger as features, it can perform well on these test cases. In contrast, the vocabulary-based approach, which only focuses on the code tokens, achieves similar performance as for the rest of the dataset (cf. Table II). Overall, although PEELER outperforms the state of the art in general, the state of the art can complement PEELER, and vice versa.

### D. Threats to Validity

An external threat in our experiment lies in the reliability of the ground truth data that we used. It was collected from a recent study, where they run the test cases to decide on flakiness. Unfortunately, flakiness is inherently difficult to reproduce: a test which has been considered as non-flaky by developers but may actually be a flaky one. Nevertheless, this threat is mitigated by the fact that the dataset was created after each test 10K times [22].

A second threat to validity is related to our implementation of PEELER, which was targeted at the Java language. While our design is generic and does not present specific programming-language constraints, our implementation was guided by the availability of artefacts (e.g., program parser) and benchmarks for evaluation.

## VII. RELATED WORK

### A. Flaky Test Studies

Flaky tests were firstly comprehensively investigated by Luo *et al.* [7]. They dissected the most common reasons for the presence of flaky tests. Eck *et al.* [11] deepened this knowledge by surveying the developers about the root cause of flaky tests. Strandberg *et al.* [35] analyzed the typical cause of flaky tests in the embedded systems. Considering that test flakiness brings negative impacts on software testing practice such as mutation testing [36] and program repair [37], a number of approaches have been proposed to automatically detect flaky tests. An intuitive way to detect flaky test is to rerun the test for many times: if the test results are not consistent at all the times (i.e., both passing and failing are witnessed), the flakiness is exposed [18], [19], [4]. However, it is widely known that executing tests is rather time-consuming [38] and thus this way is impractical. Based on the assumption that some specific identifiers may have more occurrences in the flaky tests, Pinto *et al.* [21] introduced an approach to predict flaky tests statically. They transferred the test code into token sequence and then each split token is considered as a feature and the overall sequence is sent to machine learning classifiers (e.g., the Random Forest) to determine the flakiness of the test. However, as shown in our evaluation, sorely based on the test tokens is not effective enough. A potential reason proposed by Alshammari *et al.* [22] is that many tokens occur frequently in both flaky and non-flaky tests. As a static approach, our PEELER is far more effective with the help of dependency information. FlakeFlagger [22] is another state-of-the-art flaky test detector. However, our PEELER significantly outperforms it in our evaluation. Moreover, PEELER can be more efficient than it since some of the features utilized by this approach are dynamic ones (i.e., need to be extracted by running the tests).

Upon detecting flaky tests, researchers also propose to automatically fix them. iFixFlakies [39] aims at fixing flakiness caused by test-order dependencies. It searches for tests from the same test suite that can make the order-dependent tests pass and reuses their code to generate the patch. DexFix [40] targets at unordered collections (e.g., HashMap and HashSet) and adopts a number of simple heuristics to fix these cases. FLEX [41] resolves the problems caused by the algorithmic randomness in ML algorithms. It identifies the acceptable bound and updates the bound used in the test.

### B. Dependencies in Software Engineering

From results in other works reported by software engineering community, taking program dependency into consideration can always help make the approach more solid. Wang *et al.* [32] involve the information from callers/callees of a method for recommending its name. Geng *et al.* [42] utilize data/control flow from the intermediate representation of programs to perform cross-language code search. PDG [43] as well as its variant [24] have been utilized to represent code semantics and further detect semantic code clones. Moreover, PDG has also been applied to bug/vulnerability detection for programs [29], [23], [44]. These works motivate us to focus on the program dependency relations for flaky test detection.

## VIII. CONCLUSION

In this paper, we advance the state-of-the-art in flaky tests detection by introducing PEELER, a fully static approach that relies on data dependency in a Test Dependency Graph. PEELER captures test flakiness by modeling how the values of the variables involved in assertion statements are transferred as reflected in the embedded *contextual paths*. Our experiment results show that PEELER outperforms prior works by about 20 percentage points in terms of Precision and F-score. Nevertheless, while it achieves overall higher recall (by 15 percentage points), PEELER can be used complementarily to FlakeFlagger. During a live study, PEELER has already helped developers identify 12 flaky test cases in real-world project test suites.

[7] A "Test smell" denotes some bad practices during writing test code [22].

## REFERENCES


[1] H. Leung and L. White, "Insights into regression testing (software testing)," in *Proceedings. Conference on Software Maintenance - 1989*, 1989, pp. 60–69.

[2] J. Micco, "The state of continuous integration testing at google," 2017, https://bit.ly/2OohAip.

[3] "The state of continuous integration testing," 2020, https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45880.pdf.

[4] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 101–111.

[5] S. Paydar and A. Azamnouri, "An experimental study on flakiness and fragility of randoop regression test suites," in *Fundamentals of Software Engineering*, H. Hojjat and M. Massink, Eds. Cham: Springer International Publishing, 2019, pp. 111–126.

[6] Z. Fan, "A systematic evaluation of problematic tests generated by evosuite," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19. IEEE Press, 2019, p. 165–167. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00068

[7] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653.

[8] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 09 2018, pp. 534–538.

[9] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482. [Online]. Available: https://doi.org/10.1145/3377811.3381749

[10] 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 *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 857–862.

[11] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[12] "Circleci: continuous integration and delivery," https://circleci.com/, 2021.

[13] "Flakytest," https://developer.android.com/reference/androidx/test/filters/FlakyTest, 2021.

[14] "Flaky test handler plugin - jenkins - jenkins wiki," https://wiki.jenkins.io/display/JENKINS/Flaky+Test+Handler+Plugin, 2021.

[15] "Maven surefire plugin – rerun failing tests," https://maven.apache.org/surefire/maven-surefireplugin/examples/rerun-failing-tests.html, 2021.

[16] "pytest: helps you write better programs," https://docs.pytest.org/en/latest/, 2021.

[17] "Selenium and testng," https://testng.org/doc/selenium.html, 2021.

[18] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 433–444.

[19] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 312–322.

[20] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Flaky test detection in android via event order exploration," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 367–378. [Online]. Available: https://doi.org/10.1145/3468264.3468584

[21] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–502.

[22] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1572–1584.

[23] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021.

[24] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Shi, "Functional code clone detection with syntax and semantics fusion learning," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 516–527.

[25] "Javaparser : The most popular parser for the java language," https://javaparser.org/, 2021.

[26] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[27] B. Karlik and A. Vehbi, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems (IJAE)*, vol. 1, no. 4, pp. 111–122, 2011.

[28] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, 2019.

[29] Y. Li, S. Wang, T. N. Nguyen, and S. Van Nguyen, "Improving bug detection via context-based code representation learning and attention-based neural networks," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.

[30] R. Rubinstein, "The cross-entropy method for combinatorial and continuous optimization." *Methodology & Computing in Applied Probability*, vol. 1, no. 2, pp. 127 – 190, 1999.

[31] N. Japkowicz and S. Stephen, "The class imbalance problem: A systematic study," *Intell. Data Anal.*, vol. 6, no. 5, p. 429–449, Oct. 2002.

[32] S. Wang, M. Wen, B. Lin, and X. Mao, "Lightweight global and local contexts guided method name recommendation with prior knowledge," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[33] E. W. Høst and B. M. Østvold, "Debugging method names," in *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009, p. 294–317.

[34] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 602–614.

[35] P. E. Strandberg, T. J. Ostrand, E. J. Weyuker, W. Afzal, and D. Sundmark, "Intermittently failing tests in the embedded systems domain," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 337–348.

[36] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 112–122.

[37] Y. Qin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "On the impact of flaky tests in automated program repair," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 295–306.

[38] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[39] 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 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, Inc, Aug. 2019, pp. 545–555.

[40] P. Zhang, Y. Jiang, A. Wei, V. Stodden, D. Marinov, and A. Shi, "Domain-specific fixes for flaky tests with wrong assumptions on underdetermined specifications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 50–61.

[41] S. Dutta, A. Shi, and S. Misailovic, "Flex: Fixing flaky tests in machine learning projects by updating assertion bounds," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[42] M. Geng, S. Wang, D. Dong, S. Gu, W. Ruan, X. Mao, and X. Liao, "Intermediate representation-based semantic graph for cross-language code search," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

[43] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "Ccgraph: a pdg-based code clone detector with approximate graph matching," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 931–942.

[44] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.