Shake It! Detecting Flaky Tests Caused by Concurrency with Shaker

Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim
Federal University of Pernambuco, Brazil
{dgs,lmt,damorim}@cin.ufpe.br

***Abstract*—A test is said to be flaky when it non-deterministically passes or fails. Test flakiness negatively affects the effectiveness of regression testing and, consequently, impacts software evolution. Detecting test flakiness is an important and challenging problem. ReRun is the most popular approach in industry to detect test flakiness. It re-executes a test suite on a fixed code version multiple times, looking for inconsistent outputs across executions. Unfortunately, ReRun is costly and unreliable. This paper proposes SHAKER, a lightweight technique to improve the ability of ReRun to detect flaky tests. SHAKER adds noise in the execution environment (e.g., it adds stressor tasks to compete for the CPU or memory). It builds on the observations that concurrency is an important source of flakiness and that adding noise in the environment can interfere in the ordering of events and, consequently, influence on the test outputs. We conducted experiments on a data set with 11 Android apps. Results are very encouraging. SHAKER discovered many more flaky tests than ReRun (95% and 37.5% of the total, respectively) and discovered these flaky tests much faster. In addition, SHAKER was able to reveal 61 new flaky tests that went undetected in 50 re-executions with ReRun.**

***Keywords*-test flakiness, regression testing, noise**

I. INTRODUCTION

A test is said to be *flaky* when it non-deterministically passes or fails depending on the running environment [1]. For example, a test may fail or pass depending on the availability of a server that the test itself has spawned—the test passes if the server is up-and-running at the point the test makes a request to the server and it fails, otherwise.

Test flakiness hurts software testing practice in multiple ways. For example, Thorve et al. [2] found that, because of test flakiness, developers lost trust in test results, choosing to ignore test failures altogether sometimes. Even ignoring failures of known flaky tests or ignoring previously classified flaky tests (i.e., not executing those tests) can be dangerous as those tests could reveal bugs in code. Furthermore, ignoring flakiness can produce the effect of observing even more failures to be analyzed during software evolution. For example, Rahman and Rigby [3] found that when developers ignored flaky test failures during a build, the deployed build experienced more crashes than builds that did not contain any flaky test failures. Test flakiness is a huge problem in industry. Most test failures at Google are due to flaky tests [1], [4]. At Microsoft, the presence of flaky tests also imposes a significant burden on developers. Wing et al. [5] reported that 58 Microsoft developers involved in a survey considered flaky tests to be the second most important reason, out of 10 reasons, for slowing down software deployments. Finally, Facebook considers flakiness detection a priority [6].

ReRun is the most popular approach in industry to detect test flakiness [1], [7]. It reruns tests multiple times. A test that failed and then passed is considered flaky, and vice-versa. The status of a test that persistently failed is unknown, but developers typically treat this scenario as a problem in application code as opposed to a bug in test code. Unfortunately, ReRun is unreliable and expensive. It is unreliable because it is hard to determine the number of reruns to find discrepancy in outputs. It is expensive because rerunning tests consumes a lot of computing power. Google, for example, uses 2-16% of its testing budget just to rerun flaky test [1]. Researchers have proposed various techniques to identify flaky tests. Bell et al. [8] proposed DeFlaker, a technique that determines that a test is flaky if the output of a test has changed even though there was no change in the code reachable from the execution trace. Note that DeFlaker cannot capture flakiness in test cases not impacted by changes and that change-impacted flaky tests do not reveal flakiness necessarily. Pinto et al. [9] proposed the use of text processing and classification to statically identify likely flaky test cases from the keywords they use.

This paper proposes SHAKER, a lightweight technique to improve the ability of ReRun to detect flaky tests. SHAKER adds noise in the execution environment with the goal of provoking failures in time-constrained tests. For example, it adds stressor tasks to compete with the test execution for the CPU or memory. SHAKER builds on the observations that concurrency is an important source of flakiness [10], [11] and that adding noise in the environment can interfere in the ordering of events related to test execution and, consequently, influence on the test outputs. The process of detecting flaky tests consists of two steps. In the first *offline* step, SHAKER uses a sample of tests known to be flaky to search for configurations of a noise generator to be used for revealing flakiness in new tests. To that end, SHAKER first builds a probability matrix encoding the relation between flaky tests and noise configurations. The matrix shows the probability of a test failing when executed with a given noise configuration. Then, SHAKER uses that matrix to search for sets of configurations that reveals the highest number of flaky tests. In the second *online* step, SHAKER uses those configurations to find time-constrained flaky tests in the project of interest.

We conducted experiments on a data set with 75 flaky tests of 11 open source Android apps. Preliminary results provide

Listing 1: AntennaPod Test

```
1  @Test
2  public void testReplayEpisodeContinuousPlaybackOff() throws Exception {
3   setContinuousPlaybackPreference(false);
4   uiTestUtils.addLocalFeedData(true);
5   activityTestRule.launchActivity(new Intent());
6   //Navigate to the first episode in the list of episodes and click
7   openNavDrawer();
8   onDrawerItem(withText(R.string.episodes_label)).perform(click());
9   onView(isRoot()).perform(waitForView(withText(R.string.all_episodes_short_label), 1000));
10  onView(withText(R.string.all_episodes_short_label)).perform(click());
11  final List<FeedItem> episodes = DBReader.getRecentlyPublishedEpisodes(0, 10);
12  Matcher<View> allEpisodesMatcher = allOf(withId(android.R.id.list), isDisplayed(), hasMinimumChildCount(2));
13  onView(isRoot()).perform(waitForView(allEpisodesMatcher, 1000));
14  onView(allEpisodesMatcher).perform(actionOnItemAtPosition(0, clickChildViewWithId(R.id.secondaryActionButton)));
15  FeedMedia media = episodes.get(0).getMedia();
16  Awaitility.await().atMost(1, TimeUnit.SECONDS).until( () -> media.getId() == PlaybackPreferences.
        getCurrentlyPlayingFeedMediaId()); ... }
```

early evidence that SHAKER is promising. SHAKER discovered many more flaky tests than ReRun (95% and 37.5% of the total from our data set, respectively) and discovered these flaky tests much faster than ReRun. SHAKER discovered 85% of the total number of possible flakies in 10% of the average time ReRun took to find its maximum number of flakies. In addition, SHAKER was able to reveal 61 new flaky tests that went undetected in 50 re-executions with ReRun. This paper makes the following contributions:

- ⋆ **Approach.** We proposed a simple lightweight approach to find time-constrained flaky tests by introducing noise in the test environment where tests will be executed.
- ⋆ **Implementation.** We developed a tool implementing SHAKER (available per request due to double blindness).
- ⋆ **Evaluation.** We evaluated SHAKER on 11 Android apps with encouraging results.
- ⋆ **Artifacts.** Our scripts, data sets, and list of issues submitted to GitHub projects, are publicly available at the following link **https://github.com/shaker-project/shaker**.

II. EXAMPLES

This section presents two examples of flaky tests in Android applications (apps) to motivate and illustrate SHAKER. Before introducing the examples, it is important to briefly explain concepts related to the Android OS. The execution of an Android app creates a separate process including an execution thread, typically called the *main* thread, or the UI thread. This thread is responsible for handling events such as callbacks from UI interactions, callbacks associated with the lifecycle of components, etc. Any costly operation, such as network operations, should be offloaded to separate threads to avoid blocking the main thread, and consequently freezing the UI. Blocking the main thread for more than 5 seconds results in an Application Not Responding (ANR) error, which crashes the app. Consequently, *asynchronous operations are common in Android apps*. Separate worker threads, responsible for these operations, are not allowed to manipulate the UI directly, which is responsibility of the main (UI) thread. Consequently, there must be coordination among worker threads and the main thread. Testing frameworks, such as Espresso [12], provide means to handle asynchronous operations. However, developers may fail to properly coordinate tasks from tests.

A. AntennaPod

AntennaPod [13] is an open source podcast manager for Android supporting episode download and streaming. The AntennaPod app is implemented in Java in ∼50KLOC. As of the date of submission,[1] the app had 250 GUI tests written in Espresso [12] and UIAutomator [14].

Listing 1 shows a simplified version of a test that checks whether a podcast episode can be played twice. It uses the Awaitility library [15] to handle asynchronous events, such as I/O events related to notifications of media playback. Executing the statement at line 3 turns off the continuous playback option. This stops the app from automatically playing the next episode in queue after it finishes the current one. The statement at line 4 then adds local data (e.g., podcast feeds, images, and episodes) to the app whereas the execution of the statements at lines 7–14 navigate through the GUI objects with the effect of playing the first episode in the queue. Line 16 shows an assertion based on the Awaitibility library. The assertion checks if the episode is being played and indicates that test execution should wait for at most one second to verify this (line 16). When the play button is pressed (line 14), the app runs a custom service in the background[2]—to load the media file from the file system and, subsequently, play the media to the user. These are typically expensive I/O operations. If the machine running the test is heavy-loaded, the 1 second limit may not be reached. Consequently, the execution of the test will fail with a `ConditionTimeoutException` exception raised by the Awaitibility library.

We ran this test case 50 times and found failures in 11 runs, i.e., 22% of the cases. Considering this example, one would need to execute the test for more than four times to detect the failure with high probability. When configured to execute the test suite using only the most effective noisy SHAKER configuration, we were able to detect failure in the first execution of the test suite. Furthermore, we re-executed the test case for 50 times with this configuration and were able

[1] Revision SHA dd5234c as per the time of submission.

[2] Look for "Thread" in the `PlaybackService.java` file [16].

Listing 2: Paintroid Test

```
1 @Test
2 public void testFullscreenPortraitOrientationChangeWithShape() {
3  onToolBarView().performSelectTool(ToolType.SHAPE);
4  setOrientation(SCREEN_ORIENTATION_PORTRAIT);
5  onTopBarView().performOpenMoreOptions();
6  onView(withText(R.string.menu_hide_menu)).perform(click());
7  setOrientation(SCREEN_ORIENTATION_LANDSCAPE); pressBack();
8  onToolBarView().performOpenToolOptionsView().performCloseToolOptionsView(); }
```

to detect flakiness in all of them. Although each individual execution of the test suite is more expensive with SHAKER, it requires fewer re-executions of the test suite to detect flakiness. For comparison, one non-noisy execution of this test case takes 11.08s whereas one execution of the test case under SHAKER's noisy environment takes 20.2s, i.e., the execution without noise is ∼2x faster in this particular case. However, comparing the total execution time, we observed that SHAKER enabled the discovery of flakiness 2.49x (=50.3/20.2) faster and, more importantly, it required a single execution for that.

B. Paintroid

Paintdroid is a graphical paint editor application for Android implemented in Java in ∼25KLOC. As of its latest version,[3] it had 250 Espresso tests. One of such tests checks whether some buttons can be clicked after changing the screen orientation. Listing 2 shows the test. It first selects the Shape drawing option (line 3), and then sets the screen orientation to portrait (line 4). Then, it opens a menu that shows a list of options (line 5) (e.g., option to save an image, option to export image to a file, etc). The test clicks on the full screen option (line 6), then it changes orientation to landscape (line 7), exits full screen mode (line 7), and clicks on the tool options to again open a menu, and then close it (line 8). Note that there are no assertions in the test. The intention is to validate that the options remain clickable as the orientation changes from portrait to landscape.

Like the previous example, this test can produce different results depending on the efficiency of the machine. More precisely, the click on the menu item (line 6) can be performed before or after the menu is rendered on the screen (line 5). As expected, the test fails, throwing the `PerformException`, if the click is performed before the menu is shown. Changing screen orientation corresponds to a configuration change in Android.[4] When a configuration change happens, Android destroys and recreates the current screen (represented by an `Activity` object). This happens because changing orientation might result in a different screen layout.

We ran this test case for 50 times with ReRun and found failures in 4 runs (8% of the cases). This example shows that, albeit practical and widely adopted in industry [1], [6], [10], ReRun can be (1) ineffective or (2) costly to proactively detect flaky tests. Given that it takes 12.5 (=50/4) executions on average to find flakiness and each regular execution takes 5.54s, the aggregate cost to detect this flaky test with ReRun is 69.25s (=12.5*5.54). We also ran the test case for 50 times with SHAKER and found failures in 18 runs (26% of the cases). As the test fails in every 2.78 (=50/18) executions and each execution with SHAKER takes 7.08s, the aggregate cost of SHAKER to find this flaky test is 19.68s (=2.78x7.08). Overall, SHAKER reveals the flaky test 3.52x faster than ReRun (=69.25/19.68) despite having the execution of the test case itself 1.28x (=7.08/5.54) slower.

[3]Revision SHA 1f302a2 as per the time of submission

[4]https://developer.android.com/guide/topics/resources/runtime-changes

III. SHAKER

The goal of SHAKER is to detect flaky tests. The observation that motivates SHAKER is that many tests are flaky because of timing constraints in test executions [2], [10], [11], such as those from the examples. Our hypotheses are that 1) such tests can be detected by adding noise in the environment where test cases will run and 2) rerunning tests in a noisy environment will more promptly reveal flaky tests compared with rerunning tests without noise.

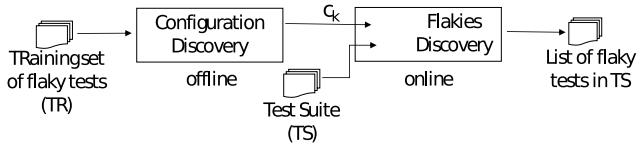

Fig. 1: SHAKER's workflow.

Figure 1 shows SHAKER's workflow. SHAKER has an offline step and an online step. In the first *offline* step, SHAKER uses a sample of tests known to be flaky to search for configurations of a noise generator. In the second *online* step, SHAKER uses those configurations to find flaky tests in the test suite of a project provided as input.

The following sections detail SHAKER. Section III-A describes how an off-the-shelf tool can be used to generate noise in the execution environment. Section III-B describes the offline step of generating configurations for a noise generator. Finally, Section III-C describes how SHAKER optimizes ReRun to find flaky tests efficiently.

A. Noise Generation

A noise generator is a tool to create load. For example, a noise generator can spawn "stressor" tasks that can influence the execution environment where tests will be executed. Existing tools provide different choices of target for noise generation. We focused on CPU and memory options as we empirically found that they influence detection of test flakiness

(see Section V-A). We used the stress-ng [17] tool to create CPU and memory load. We used the following options:

- **–cpu** n. Starts n stressors to exercise the CPU by working sequentially through different CPU stress methods like Ackermann function or Fibonacci sequence.
- **–cpu-load** p. Sets the load percentage p for the *–cpu* command.
- **–vm** n. Starts n stressors to allocate and deallocate continuously in memory.
- **–vm-bytes** p. Sets the percentage p of the total memory available to use by the tasks created with option *–vm*.

For example, the command `stress-ng --cpu 2 --cpu-load 50% --vm 1 --vm-bytes 30%` configures stress-ng to run two CPU stressors with 50% load each and one virtual memory stressor using 30% of the available memory. The documentation of stress-ng can be found elsewhere.[5]

In addition to the options listed above, SHAKER uses an option that we found important for finding flaky Android tests—the number of cores available for use by an Android emulator. This option can be used to restrict an Android emulator to run on a specified number of cores.

B. Step 1: Discovering Configurations

The goal of this step is to identify configurations of the noise generator that are more likely to reveal flakiness in a test suite. We search for configurations as a heavy-load could crash the emulator. It takes as input a set of tests TR, *known to be flaky*, and reports on output a list of configurations $c_k = [v_1, ..., v_n]$. A noise generator is configured from a list of options $[o_1, ..., o_n]$, with each option o_i ranging over the interval lo_i-hi_i. Section III-A describes which options are currently used. The flaky tests in TR can be obtained with ReRun, i.e., by rerunning test sets of several applications for multiple times. Section IV explains how we obtained the training set to evaluate SHAKER. To identify "good" configurations, it is necessary to define a metric for configuration quality. We use the symbol $fit(c_k, TR)$ to denote the fitness value of configuration c_k for the test set TR. Fitness value is obtained by computing the average probability of detecting flakiness on TR when the configuration c_k is used for noise generation.

Example. Consider that the set TR contains three tests. A given configuration c that detects flakiness in TR with probabilities $\{0, 2, 0.5, 0.0\}$ has $fit(c, TR)$=(0.2+0.5)/3=0.23. These probabilities are obtained by running the test suite multiple times and computing the average number of failures. In this case, one test failed in 20% of the reruns, another test failed in 50% of the reruns, and another did not fail at all.

The search for noise configuration is realized in two stages. First, SHAKER samples configurations and generates a probability matrix characterizing the probability of each configuration to find flakiness in TR. Second, SHAKER uses that matrix to search for sets of configurations. The following sections elaborate each of these steps in detail.

[5]https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html

1) Generation of probability matrix: This step takes as input a set of flaky tests TR and reports on output a probability matrix M, relating the tests in TR and *randomly-sampled* configurations in K by their corresponding probabilities. The symbol $M[t][c]$ denotes the probability of configuration $c \in K$ detecting flakiness in $t \in TR$. To obtain approximate probability measurements, SHAKER runs each test several times on each sampled configuration. The probability measurement $M[t, c]$ is obtained by dividing the number of failures (of t on c) found by the total number of reruns (of t on c).

2) Search for sets of configurations: SHAKER offers two strategies to search for configurations.

The **Greedy** strategy sorts configurations according to their fitness value and reports the top-n configurations in the sorted list. The value n is provided by the user. Note that this strategy does not offer guarantees that every flaky test will be detected.

The Minimum Hitting Set (**MHS**) strategy takes that problem into account. MHS [18] is a well-known intractable problem with efficient polynomial-time approximations [19]. To sum up, it enables SHAKER to obtain minimum sets of configurations (columns of the matrix) that detect the maximum number of flaky tests (rows in the matrix). Variations of the MHS problem exist considering weights and returning complete or partial (sub-optimal) solutions [20]. SHAKER builds on the unweighted and complete MHS version, which takes a *boolean* matrix as an input and produces a minimum hitting set encoding configurations as an output. We abstracted the probability matrix M to only encode low or high likelihood of configurations detecting flaky tests. Intuitively, we are only interested in a configuration c to detect flakiness of a certain test t if the observed probability $M[t][c]$ is above a certain threshold. More precisely, SHAKER computes an abstract matrix A defined as $A[t][c]$=1 if $M[t][c] >= threshold$, otherwise 0. SHAKER runs MHS on the boolean matrix A. The goal is to find a set of configurations (columns in the matrix) that detects flakiness in tests (rows in the matrix). Note that, although MHS assures that all flaky tests are covered (i.e., a test would be detected with some configuration in the MHS), there is no guarantee that these tests are uniformly covered.

Example. Figure 2 shows an illustrative example of the MHS procedure to discover configurations for detecting flakiness. The left-hand side of the figure shows the probability matrix M. The abstract matrix A appears on the right-hand side. For space, we used a 3x4 matrix, i.e., the test suite contains three test cases and SHAKER sampled four configurations. In practice, these matrices are much bigger. The matrix on the left shows the probabilities of each configuration detecting flakiness on TR. The matrix on the right-hand side is obtained using the abstraction function described above with a threshold value of 0.5. There are five hitting sets associated with the abstract matrix, namely $\{c_1, c_2, c_4\}$, $\{c_1, c_3, c_4\}$, $\{c_2, c_3, c_4\}$, $\{c_2, c_4\}$, $\{c_1, c_2, c_3, c_4\}$. The MHS algorithm is able to identify that the set $\{c_2, c_4\}$ is a minimal set that hits (i.e., covers) the tests in TR. In this case, it is also the minimum.

	c_1	c_2	c_3	c_4
t_1	0.1	0.6	0.5	0.2
t_2	0.6	0.6	0.1	0.2
t_3	0.1	0.1	0.1	0.5

	c_1	c_2	c_3	c_4
t_1	0	1	1	0
t_2	1	1	0	0
t_3	0	0	0	1

Fig. 2: Original probability matrix (M) and its abstracted version (A) using a threshold of 0.5. MHS(A)=$\{c_2, c_4\}$.

C. Step 2: Discovering Flakies

Finally, SHAKER uses the configurations obtained on Step 1 to determine which of the tests the user provides are flaky. Figure 1 illustrates the inputs and output of this step in the box named "Flakies Discovery". SHAKER reruns the test suite on each configuration for a specified number of times and reports divergences on the test outputs.

Note the tension between cost and effectiveness of SHAKER. The execution of a test suite under a loaded environment should be slower compared to a regular (noiseless) execution as different tasks are competing for the machine resources. Compared to ReRun, however, SHAKER may require less executions to detect flakiness. As result, the aggregate cost of detecting flakiness would be lower.

IV. Objects of Analyses and Setup

This section describes the methodology we used to build a data set of flaky tests (Section IV-A) and the setup of SHAKER that we used to run the experiments (Section IV-B).

A. Objects

We mined flaky tests from various GitHub projects to build a dataset to evaluate SHAKER. We used the following search criteria to select projects:

1) the project must be written in Java or Kotlin;
2) the project must have at least 100 stars;
3) the project must include tests in Espresso or UIAutomator;
4) the project needs to be built without errors.

We sampled a total of 11 projects that satisfied this criteria and used ReRun to find test flakiness. More precisely, we executed the test suite of these projects for 50 times using a generic Android Emulator (AVD) with Android API version 28. As usual, we consider a test to be flaky if there was a disagreement in the outcomes (i.e., pass, fail, or error) across those runs. For example, we considered as flaky a test that passes in all but one run. To confirm that we could reproduce flakiness, we repeated the execution of each flaky test for 100 times. Section V-E shows results of running SHAKER to find other flaky tests on this same set of projects, i.e., flakies that went undetected with the aforementioned procedure and only SHAKER could detect. Table I shows the 11 projects we used. The column "App" shows the name of the Android app, the column "#Tests" shows the number of Espresso or UIAutomator tests in that application, the column "#Flakies" shows the number of flaky tests detected, column "`@FlakyTest(+/-)` shows numbers x/y with x indicating the number of flaky tests we found that do *not* contain the `@FlakyTest` annotation,[6] whereas y indicates the number of tests from the test suite containing the annotation `@FlakyTest` that we missed. Column "#Stars" shows the number of stars that the project received on GitHub. Finally, columns "GitHub..." and "SHA" show the GitHub address and corresponding SHA prefix of the revision.

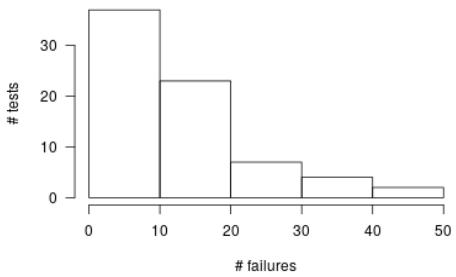

Fig. 3: Histogram of test failures.

We found flakiness in 75 of the 1,298 tests we analyzed (=5.78% of the total). We found flaky tests in 7 of the 11 apps. In two of these apps, namely `Espresso` and `Flexbox-layout`, we found only 1 flaky test. We highlighted the apps without flaky tests in gray color. Results also show that the `AntennaPod`, `FirefoxLite`, and `Orgzly` were the apps with the highest number of flaky tests, with 12, 15, and 38 flaky tests, respectively. These projects are among those with the highest number of test cases too, with 250, 70, and 266 tests, respectively. Curiously, we found that `Flexbox-layout` is one of the apps with the highest number of tests (232) and lowest number of flakies detected (1). Despite this evidence of determinism on the test suite, developers of this app chose to use the annotation `@FlakyTest` in *all* tests. We inspected the code and it appears that developers used the annotation only for documentation—all tests in that app have that annotation and filtering test cases in that state would result in no test executed. `Flexbox-layout` consists in a library for implementing widgets that adhere to the CSS Flexible Box Layout module using the `RecyclerView` widget, which allows displaying lists and grids in Android applications. Tests rely heavily on checking list/grid elements, which are typically computed in worker threads, that require synchronization to be posted to the Main Thread.

Figure 3 shows a histogram of number of failures for the 75 flaky tests we discovered using ReRun. Nearly 52.1% of the flaky tests revealed failure in 10 or less executions. This number is reflected in the leftmost bar from the histogram. Intuitively, to capture those cases, one would need to rerun the test suite for five times given that the failure probability for that group of tests is 20% (=10/50).

B. Setup

To evaluate SHAKER, we need to define (1) the training set of test cases TR that will be used to discover configurations (step 1) and (2) the testing set of test cases TS that will be used to discover the flaky tests (step 2). For that, we divided

[6]The `@FlakyTest` annotation is a JUnit test filter. It is used on test declarations to indicate JUnit to exclude those tests from execution (if a corresponding command is provided on input).

TABLE I: Apps and tests.

#	App	#Tests	#Flakies	`@FlakyTest`(+/-)	#Stars	GitHub URL (https://github.com/URL)	SHA
1	AntennaPod	250	12	12/0	2.8k	/AntennaPod/AntennaPod	dd5234c
2	AnyMemo	150	0	0/0	117	/helloworld1/AnyMemo	7e674fb
3	Espresso	14	1	1/0	1.1k	/TonnyL/Espresso	043d028
4	FirefoxLite	70	15	15/3	220	/mozilla-tw/FirefoxLite	048d605
5	Flexbox-layout	232	1	0/231	15.5k	/google/flexbox-layout	611c755
6	Kiss	16	3	3/0	1.5k	/Neamar/KISS	00011ce
7	Omni-Notes	10	0	0/0	2.1k	/federicoiosue/Omni-Notes	b7f9396
8	Orgzly	266	38	38/0	1.5k	/orgzly/orgzly-android	d74235e
9	Paintroid	270	5	5/0	101	/Catrobat/Paintroid	1f302a2
10	Susi	17	0	0/0	2k	/fossasia/susi_android	17a7031
11	WiFiAnalyzer	3	0	0/0	1k	/VREMSoftwareDevelopment/WiFiAnalyzer	80e0b5d
Total	-	1,298	75 (5.78%)	74/234	-	-	-

our dataset containing 75 flaky tests as follows. We randomly sampled 35 tests and added them to the training set. The remaining 40 tests were added to the testing set.

The configuration discovery step of SHAKER also requires a sample of random configurations that will constitute the columns of the concrete and abstract matrices. For that, we sampled 50 configurations uniformly distributed across the domains of the five parameters we analyzed (see Section III-A): four parameters from stress-ng and one parameter from the AVD. To obtain probabilities for the concrete matrix M, we ran each test on each of these configurations for 3 times. The result of this execution is a probability matrix with failure probability values 0, 0.33, 0.66, 1.0. To construct the abstract matrix A, we used a probability threshold of 0.66, i.e., values equal or above that level are set to 1 (true) and values below that level are set to 0 (false).

V. EVALUATION

The goal of this section is to evaluate the effectiveness of SHAKER. We pose the following research questions.

RQ1. Do tests fail more often in noisy environments than in regular (non noisy) environments?

The purpose of this question is to evaluate if executing tests in noisy environments has the effect of making tests fail more often. This is an important question because SHAKER builds on that assumption to detect flaky tests, so if results are positive, than SHAKER is likely to be useful.

RQ2. How repeatable is the discovery of flaky tests with a given noise configuration?

This question analyzes the variance of results obtained with a given configuration of the noise generator. If results are very non-deterministic then selecting configurations, as described on Section III-B2, is helpless as results would be unpredictable.

RQ3. How effective is the search for configurations of the noise generator (e.g., Greedy and MHS) that SHAKER uses?

This question evaluates the effect of the (configuration) search strategies proposed by SHAKER in their ability to detect flaky tests. More precisely, this question compares Greedy MHS, and Random search in their effect to detect flakiness.

RQ4. How effective is SHAKER to find flaky tests?

This question evaluates how SHAKER compares with ReRun to find the flaky tests in our data set. To answer this question, we measured how long SHAKER and ReRun took to discover all flaky tests and how quickly each technique finds most tests.

RQ5. How effective is SHAKER to find *new* flaky tests?

For fairness with ReRun, the previous question restricted the evaluation of SHAKER to the set of flaky tests that ReRun itself was able to discover within 50 re-executions (see Section IV). This question evaluates the ability of SHAKER to discover flaky tests not detected by ReRun in the same set of projects.

The following sections elaborate the answers to each of these questions.

A. Answering RQ1: Do tests fail more often in noisy environments than in regular (non noisy) environments?

The purpose of this question is to evaluate the effect of introducing noise in the environment where tests are executed. This question is important as SHAKER assumes there is such effect. To answer the question, we ran statistical tests to evaluate if there are differences in the rate of failures observed in noisy executions versus standard executions.

As we evaluated the *effect* of noise on the same data set, we used a statistical test that takes two paired distributions on input–one distribution associated with regular executions and one distribution associated with noisy executions. Each number in the distribution corresponds to the rate of failures in one execution of the test suite, i.e., the fraction of the 75 tests from our data set that fails. We ran each test suite on each treatment for 30 times. Consequently, each distribution contains 30 samples. First, we ran a Shapiro-Wilk test to check if the data is normally distributed. The p-values of this test are higher than the traditional threshold of $\alpha = 0.05$. As such, we cannot reject the null hypothesis that the data is normally distributed with 95% confidence. Given that both data sets are normally distributed, we used the paired t-test parametric statistical hypothesis test to check if there is difference in the measurements. The null hypothesis (H_0) is that the measurements are the same, that is, introducing noise does *not* impact rate of failure. The test indicated that we could

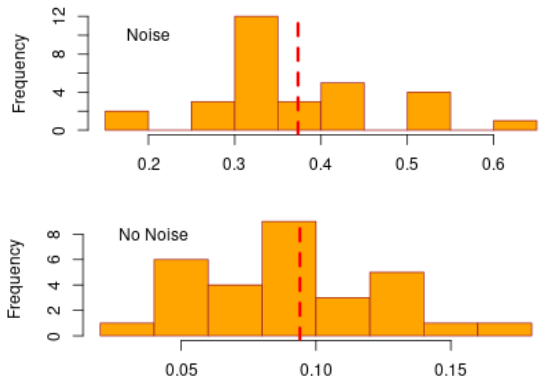

Fig. 4: Histograms of ratio of failures for execution with and without noise. Vertical lines show average value.

reject H_0 with 99% confidence as the p-value is less than 0.01. Since the distributions were different, we proceeded to evaluate the effect size. For that, we used Cohen's d to measure the difference in group means in terms of standard deviation units. This is given by $d = (\mu_1\text{-}\mu_2)/\sigma$, where $\mu_1\text{-}\mu_2$ is the difference over the two means from the sample data and σ is the standard deviation of the data. The magnitude of the effect varies with the value of d, ranging from very small (<0.01) to huge (>2.0) [21]. We obtained a value of $d = 2.52$, which indicates that the effect of introducing noise was huge.

Figure 4 shows histograms for the distributions of values associated with noisy and noiseless runs. Note that, although some configurations manifest a relative small number of failures (0.15-0.2 range), most executions with noisy configurations raise more failures in tests. The average rate of failures in a noisy execution is 0.37. In contrast, regular executions have average failure rate of 0.09, which is substantially smaller compared with that of noisy runs. To sum up:

> *Summary:* Results indicate that introducing noise in the environment increases the ratio of failures in test suites with time-constrained test cases.

B. Answering RQ2: How repeatable is the discovery of flaky tests with a given noise configuration?

The goal of this question is to evaluate how repeatable are the results obtained with a given configuration. If results obtained with two runs of the test suite with the same configuration are very different, then choosing configurations randomly would be no worse than systematically searching for configurations as described on Section III-B.

To answer this question, we randomly selected 15 noise configurations and run the test suite on each one of them for 10 times, measuring the percentage of failures detected on each execution. For each one of the 15 configurations, we generated a distribution with 10 samples, where each sample indicates the percentages of failures detected on a given configuration.

Figure 5 shows a boxplot associated with the standard deviations of these 15 distributions. Results indicate that the

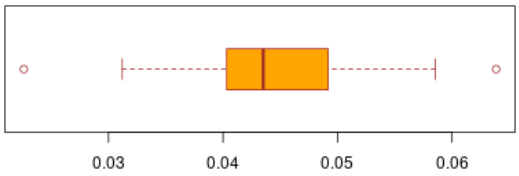

Fig. 5: Standard deviations of distributions of 10 reruns on each of the 15 randomly-selected configurations.

average and median standard deviation is ∼.04, indicating that the average difference in measurements is not superior to 8% (avg±σ for a 95% confidence interval) of the total number of flaky tests found. We conclude that:

> *Summary:* Despite the non-deterministic nature of the noise generation process, we observed relatively similar rates of failures across multiple executions of the same noise configuration.

C. Answering RQ3: How effective is the search for configurations of the noise generator (e.g., Greedy and MHS) that SHAKER *uses?*

Recall that SHAKER selects configurations in two steps (see Section III-B). First, it generates a probability matrix and then it selects configurations from that matrix. This research question evaluates the effectiveness of the configuration selection strategies SHAKER uses. More precisely, it evaluates the ability of different techniques to select configurations.

We evaluated three strategies for selecting configurations, namely, (i) MHS, (ii) Greedy, and (iii) Random. Recall that MHS is the technique that finds the smallest set of configurations whose execution of the test suite is capable of detecting all flaky tests from the training set (as per their associated probabilities in the abstract matrix). Greedy is the technique that selects configurations with maximum individual fitness scores (see Section III-B). Random serves as our control in this experiment. It is the technique that randomly selects configurations regardless of their scores. Both Greedy and Random select the same number of configurations as MHS.

The metric we used to compare techniques is the ratio of flaky tests detected when re-running the test suite against all configurations in the set associated with a technique. For example, let us consider that MHS produced four configurations. To obtain the score for that set, we execute the test suite four times, once for each configuration. Let us assume, we observe discrepancy in the outputs of 30 of the 40 flaky tests from the testing set. In that case, the score associated with the configuration set will be 0.75 (=30/40).

We ran a statistical test to evaluate if there are differences in the measurements obtained by MHS, Greedy, and Random. The metric used was the fitness score, as described above, i.e., we measured the ratio of flaky tests detected when using each selection strategy. We ran each technique for 10 times, so each distribution of measurements contain 10 samples.

As for RQ1, we ran a Shapiro-Wilk test to check if the data is normally distributed. We found that the p-values are above $\alpha = 0.05$, therefore we concluded that the data is normally distributed. We then used Bartlett's test to check the homogeneity of variances, which reveals that the samples come from populations with the same variance. As such, we chose to use the one-way ANOVA (ANalysis Of VAriance) parametric test to check statistical significance of the sample means by examining the variance of the samples. The null hypothesis (H_0) is that there is no variation in the means of sample measurements, which would indicate that there is no impact on changing selection strategies. The test reveals that there are statistically significant differences among treatments at the $p < 0.01$ level (ANOVA F(2,27)=66.83, $p < 0.01$).

TABLE II: Post-hoc analysis for RQ3 — Multiple Comparison of Means using Tukey HSD

Group 1	Group 2	Mean Diff.	p adj.	Lower	Upper	Reject
Greedy	MHS	0.008	0.8352	-0.0279	0.0439	**False**
Greedy	Random	-0.141	0.001	-0.1769	-0.1051	**True**
MHS	Random	-0.149	0.001	-0.1849	-0.1131	**True**

We performed a post-hoc paired comparison to evaluate which of the techniques differ. For that, we used the Tukey HSD test to execute multiple pairwise comparisons. Table II shows the difference in means, the adjusted p-values, and confidence levels for all possible pairs. Columns p-values and confidence levels show that between-group differences are significant only when comparing Random to MHS and Greedy. However, statistically, results do reject the hypothesis that Greedy and MHS are significantly different.

To sum up:

> *Summary:* Results indicate that there is advantage in selecting noise configurations based on their fitness scores as opposed to randomly picking them. However, there is no statistical support to claim significant differences between MHS and Greedy.

D. Answering RQ4: How effective is SHAKER *to find flaky tests?*

The goal of this research question is to evaluate SHAKER's performance. To that end, we analyzed two dimensions: (i) efficiency, i.e., how fast it finds flaky tests, and (ii) completeness, i.e., what is the fraction of the set of known flaky tests the technique detects. The set of flaky tests to be detected corresponds to the test set, as defined on Section IV-B.

Figure 6 shows the progress of SHAKER and ReRun in detecting flaky tests over time. The x-axis denotes time in minutes whereas the y-axis denotes the number of flaky tests detected. The steep increase in the number of flaky tests detected by SHAKER indicates that it quickly discovers many flaky tests. For example, 26 of the 40 flaky tests failed (i.e., 65% of the total) in the first execution of the test suite with SHAKER. Recall that the test set involves test cases of multiple projects. ReRun detected flakiness at a much slower pace, as reflected by the growth of its plot. ReRun needed 43 re-executions of the test suite and 316m (=5h16m) to reach saturation with 15 flaky tests detected (i.e., 37.5% of the total) whereas SHAKER needed 14 re-executions and 106m (=1h46m) to reach saturation with 38 flaky tests detected (i.e., 95% of the total). The vertical dotted line on Figure 6 marks the 32m point in time corresponding to 10% of the time required by ReRun to saturate, which is the limit of the x-axis. In contrast, at that point, SHAKER had already found 34 of the flaky tests (i.e., 85% of the total).

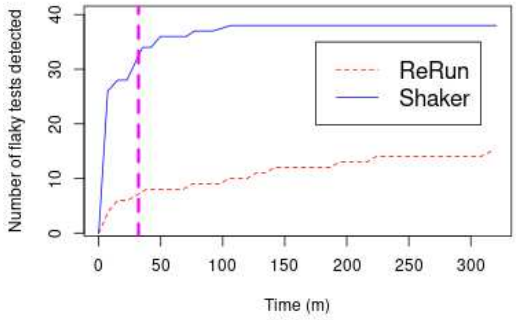

Fig. 6: Progress over time of SHAKER and ReRun in detecting flaky tests.

We used the Area Under the Curve (AUC) as a proxy of effectiveness. Rothermel et al. [22] pioneered the use of this metric to assess performance of test case prioritization techniques. The larger the area under the curve the better. For these progress plots, a higher area indicates higher ability to detect flaky tests and to detect them quickly. Intuitively, an optimal technique would detect all flaky tests in one execution and would have the AUC of a big trapezoid. Considering the plot from Figure 6, the ReRun curve has an AUC of 3,491 where as the SHAKER curve has an AUC of 11,628, i.e., the area of SHAKER is 3.33x (=11,628/3,491) higher than that of ReRun. We used the `auc` function of the MESS library in R to compute the AUCs of these plots. That implementation uses the traditional trapezoidal integration method[7] to obtain AUCs.

It is worth noting that, because of the increased load in the environment, one execution of the test suite with SHAKER is slower than one execution of the test suite with ReRun. The average cost of one execution of the test suite with SHAKER is 8m55s whereas the average cost of one execution of the test suite with ReRun is 7m38s. Also note that SHAKER has the additional cost of finding configurations. However, this phase does not need to be executed as software evolves. Although the configurations that SHAKER finds can be calibrated with the inclusion of new projects and test suites for training, the proposal is to use the set of configurations found in test runs of any Android test suite. To sum up:

[7]https://www.lexjansen.com/nesug/nesug02/ps/ps017.pdf

Summary: Results show that (1) SHAKER discovered many more flaky tests than ReRun (95% versus 37.5%) and (2) it discovered these flaky tests much faster. SHAKER discovered 85% of the total number of possible flakies in 10% of the time ReRun took to find its maximum number of flakies.

E. Answering RQ5: How effective is SHAKER to find new flaky tests?

We ran SHAKER on each project for 5h with the goal of finding additional flaky tests. Table III shows the new flaky tests we found. We discarded tests with the `@FlakyTest` annotation from our search to avoid the risk of inflating our results given that developers already signaled a potential issue on those tests. As such, we needed to discard the project Flexbox-layout as all tests from that project contain that annotation.

TABLE III: Number of new flaky tests found with SHAKER.

App	# flaky tests	
	All	First 3
AntennaPod	27	20
FirefoxLite	5	3
Omni-Notes	4	3
Orgzly	14	10
Paintroid	10	8
WiFiAnalyzer	1	1
Total:	61	45

Overall, we found new flaky tests in 6 of the 11 projects (10 if we exclude Flexbox-layout) and a total of 61 new flaky tests across these projects. SHAKER detected a total of 45 of these flaky tests in the first three runs of the test suite, confirming the behavior observed in the previous experiment when we showed that SHAKER could reveal most flaky tests very early during the search. We did not find any flaky tests on projects AnyMemo, Espresso, Kiss, and Susi. As per Table I, note that the previous experiment we ran could not reveal any flaky tests on AnyMemo and Susi either. In contrast, we found one flaky test in Omni-Notes and WifiAnalyzer, projects that previously did not manifest test flakiness. It is important to recall that 95% of the 75 flaky tests found by ReRun are also found by SHAKER (see Section V-D). To sum up:

Summary: SHAKER revealed 61 new flaky tests that went undetected in 50 re-executions with ReRun.

For each project where we found flaky tests, we opened issues on Github informing the detected flaky tests, explaining how we detected flakiness, and pointing to a shell script with instructions for reproducing our steps.[8] AntennaPod developers confirmed they are aware of flakiness for some of their tests, due to emulator performance when testing the Playback functionality. This is consistent with our discussion in Section II. Some of those tests are ignored on their Continuous Integration systems. Nevertheless, we still detected flakiness in other tests, that developers considered as important to fix.[9] Omni-Notes developers also modified their test suite to include the `@FlakyTest` annotation in the tests we reported as flaky.[10]

F. Threats to validity

Threats to the *construct validity* are related to the appropriateness of the evaluation metrics we used. We used popular metrics previously used. For example, we used the ratio of detected flakiness, the number of flaky tests detected, and the Area Under the Curve (AUC) to evaluate the techniques.

Threats to the *internal validity* compromise our confidence in establishing a relationship between the independent and dependent variables. To mitigate this threat, we carefully inspected the results of our evaluation. In addition, we ran our experiments in different machines to confirm the impact of noise in detecting flakiness.

Threats to the *external validity* relate to the ability to generalize our results. We cannot claim generalization of our results beyond the particular set of projects studied. In particular, our findings are intrinsically limited by projects studied, as well as their domains. The problems we found are related to task coordination. Nevertheless, future work will have to investigate to what extent our findings generalize to software written in other programming languages and frameworks (beyond Android and UI tests).

VI. RELATED WORK

We describe in the following recently related papers to ours.

A. Empirical studies about bugs in test code

Different empirical studies [10], [23]–[25] have attempted to characterize the causes and symptoms of buggy tests, i.e., problematic tests that can fail raising a false alarm when there is no indication of a bug in the application code. This paper focuses on test flakiness, which is one of several possible types of test code issues. For example, Vahabzadeh et al. [23] mined the JIRA bug repository and the version control systems of Apache Software Foundation and found that 5,556 unique bug fixes exclusively affected test code. They manually examined a sample of 499 buggy tests and found that 21% of these false alarms were related to flaky tests, which they further classified into Asynchronous Wait, Race Condition, and Concurrency Bugs. In principle all such problems can result in timing constraints that SHAKER could capture. Note, however, that we focused on Android, the diagnosis of defect is out of scope, and that SHAKER is a technique to find these issues whereas the aforementioned works manually analyze test artifacts.

Luo et al. [10] analyzed the commit history of the Apache Software Foundation central repository looking specifically for flakiness. They analyzed 1,129 commits including the keyword “flak” or “intermit”, and then manually inspected each commit. They proposed 10 categories of flakiness root causes and summarized the most common strategies to repair them. Many of the problems reported are related to timing constraints that could, in principle, be captured by SHAKER. We remain to investigate how SHAKER performs for software

[8] See https://bit.ly/3dbqCZe for a script for AntennaPod

[9] https://github.com/AntennaPod/AntennaPod/issues/4194

[10] https://github.com/federicoiosue/Omni-Notes/issues/761

of different domains. Thorve et al. [2] conducted a study in Android apps and observed that the causes of test flakiness in Android apps are similar to those identified by Luo et al. [10]. They also found two new causes as Program Logic and UI. Altogether, these studies show that test flakiness is prevalent and a potential deterrent to software productivity.

B. Detection of test smells

Code smells are syntactical symptoms of poor design that could result in a variety of problems. Test smells manifest in test code as opposed to application code. Van Deursen et al. [26] described 11 sources of test smells and suggested corresponding refactorings to circumvent them. More recent studies have been conducted on the same topic. Bavota et al. [27] and Tufano et al. [28] separately studied the sources of test smells as defined by Van Deursen et al. [26]. They used simple syntactical patterns to detect these smells in code and then manually inspected them for validation. Bavota et al. found that up to 82% of the 637 test classes they analyzed contains at least one test smell. In related work, Tufano et al. studied the life cycle of test smells and concluded that they are introduced since test creation—instead of during evolution—and they survive through thousands of commits. Waterloo et al. [24] developed a set of (anti-)patterns to pinpoint problematic test code. They performed a study using 12 open source projects to assess the validity of those patterns. Garousi et al. [29] prepared a comprehensive catalogue of test smells and a summary of guidelines and tools to deal with them. Test flakiness *may* relate to test smells. For example, the use of sleeps are good predictors of flakiness [9], [30]; they induce time constraints that could be violated. We remain to investigate whether the extent to which static methods of flakiness prediction can improve the detection ability of SHAKER. For example, in principle, it is possible to instrument particular tests to initiate and terminate noise generation.

C. Detection of flaky tests

In principle, a test case should produce the same results regardless of the order it is executed in a test suite [31]. Unfortunately, this is not always the case as the application code that is reached by the test cases can inadvertently modify static area and resetting the static area after the execution of a given test is impractical. Test dependency is one particular source of flakiness [10]. Gambi et al. [32] proposed a practical approach, based on flow analysis and iterative testing, to detect flakiness due to broken test dependencies. SHAKER is complementary to techniques for capturing broken test dependencies. It remains to investigate how a technique that forcefully modifies the test orderings (e.g., discarding tests from test runs and modifying orderings of test execution) compares with the approach proposed by Gambi et al..

Bell et al. proposed DeFlaker [8], a dynamic technique that uses test coverage to detect flakiness during software evolution. DeFlaker observes the latest code changes and marks any new failing test that did *not* execute changed code as flaky tests. The expectation is that a test that passed in the previous execution and did not execute changed code should still pass. When that does not happen, DeFlaker assumes that the changes in the coverage profile must have been caused by non-determinism. Note that DeFlaker is unable to determine flakiness if the coverage profile was impacted by change and the ability of DeFlaker to detect flakiness is bound by the ability of ReRun itself. Shi et al. [33] proposed iFixFlakies to find and fix flaky tests caused by broken test dependencies. SHAKER focuses on a different source of flakiness, which often relates to time-constraints, such as those brought by concurrency. Recently, Dong et al. [11] proposed FlakeShovel, a tool to detect flakiness in Android apps by monitoring and manipulating thread executions to change event orderings. It directly interacts with the Android runtime, instead of generating stress loads, as we did. We remain to evaluate how SHAKER compares with FlakeShovel.

Purely static approaches have also been proposed to identify flaky tests [9], [34]–[36]. An important benefit of these approaches is scalability. For example, it is possible to build services to proactively search for suspicious tests in open source repositories. On the downside, they only offer estimates of flakiness; re-execution is still necessary to confirm the issue. Herzig and Nagappan [34] developed a machine learning approach that mines association rules among individual test steps in tens of millions of false test alarms. Lam et al. [35] used Bayesian networks for flakiness classification. Pinto et al. [9] used binary text classification (e.g., Random Forests) to predict test flakiness. They used typical NLP techniques to classify flaky test cases—they tokenized the body of test cases, discarded stop words, put words in camel case, and built language models from the words associated with flaky and non-flaky tests. SHAKER is complementary to static techniques. We remain to evaluate how static classification techniques could be used to selectively run tests in a noisy environment.

VII. CONCLUSIONS

Flaky tests are a huge problem in industry. Their presence makes it difficult for developers to unambiguously interpret the results of a regression testing cycle. This paper proposes SHAKER, a lightweight approach to detect flakiness in time-constrained tests by adding noise in the execution environment. For example, SHAKER adds stressor tasks to create load in the CPU and memory. We evaluated SHAKER on a sample of 11 Android apps. Results are very encouraging. SHAKER discovered many more flaky tests than ReRun (95% and 37.5% of the total, respectively) and discovered these flaky tests much faster. In addition, SHAKER was able to reveal 61 new flaky tests that went undetected in 50 re-executions with ReRun. Our data sets and results are publicly available at https://github.com/shaker-project/shaker.

Acknowledgements. This research was partially funded by INES 2.0, FACEPE grants PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, CAPES grant 88887.136410/2017-00, FACEPE grant APQ-0570-1.03/14 and CNPq (grants 465614/2014-0, 309032/2019-9, 406308/2016-0, 409335/2016-9).

REFERENCES

[1] J. Micco, "Flaky tests at google and how we mitigate them," 2016, https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html.

[2] 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)*. IEEE, 2018, pp. 534–538.

[3] 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. [Online]. Available: https://doi.org/10.1145/3236024.3275529

[4] J. Listfield, "Where do our flaky tests come from?" 2017, https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html.

[5] 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. [Online]. Available: https://doi.org/10.1145/3293882.3330570

[6] M. Harman and P. W. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proc. SCAM'18*, 2018.

[7] J. Palmer, "Test flakiness – methods for identifying and dealing with flaky tests," 2019, https://labs.spotify.com/2019/11/18/test-flakiness-methods-for-identifying-and-dealing-with-flaky-tests/.

[8] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 433–444.

[9] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *International Conference on Mining Software Repositories (MSR)*, 2020, to Appear.

[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proc. FSE'14*, 2014.

[11] Z. Dong, A. Tiwari, X. L. Yu, and A. Roychoudhury, "Concurrency-related flaky test detection in android apps," *ArXiv*, vol. abs/2005.10762, 2020.

[12] Google, "Espresso api webpage," 2020, https://developer.android.com/training/testing/espresso.

[13] "Antennapod app website," 2020, https://antennapod.org.

[14] Google, "Uiautomator api webpage," 2020, https://developer.android.com/training/testing/ui-automator.

[15] "Awaitibility library," 2020, http://www.awaitility.org.

[16] "Antennapod playbackservice.java," 2020, https://tinyurl.com/y9vltj5e.

[17] C. King and A. Waterland, "stress-ng," https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html#description, 2020, [Online; accessed April-2020].

[18] N. Alon, D. Moshkovitz, and S. Safra, "Algorithmic construction of sets for k-restrictions," *ACM Trans. Algorithms*, vol. 2, no. 2, p. 153–177, Apr. 2006. [Online]. Available: https://doi.org/10.1145/1150334.1150336

[19] A. Gainer-Dewar and P. Vera-Licona, "The minimal hitting set generation problem: algorithms and computation," *CoRR*, vol. abs/1601.02939, 2016. [Online]. Available: http://arxiv.org/abs/1601.02939

[20] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *USENIX*, 2014, pp. 861–875.

[21] S. Sawilowsky, "New effect size rules of thumb," *Journal of Modern Applied Statistical Methods*, vol. 8(2), pp. 597–599, 2009.

[22] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *ICSM*, 1999.

[23] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. USA: IEEE Computer Society, 2015, p. 101–110. [Online]. Available: https://doi.org/10.1109/ICSM.2015.7332456

[24] M. Waterloo, S. Person, and S. Elbaum, "Test analysis: Searching for faults in tests," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 149–154. [Online]. Available: https://doi.org/10.1109/ASE.2015.37

[25] H. K. V. Tran, N. B. Ali, J. Börstler, and M. Unterkalmsteiner, "Test-case quality–understanding practitioners' perspectives," in *International Conference on Product-Focused Software Process Improvement*. Springer, 2019, pp. 37–52.

[26] A. Van Deursen, L. Moonen, A. Van Den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, 2001, pp. 92–95.

[27] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 56–65.

[28] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 4–15.

[29] V. Garousi, B. Kucuk, and M. Felderer, "What we know about smells in software test code," *IEEE Software*, vol. 36, no. 3, pp. 61–73, 2018.

[30] F. Palomba, A. Zaidman, and A. De Lucia, "Automatic test smell detection using information retrieval techniques," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 311–322.

[31] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, 2014, pp. 385–396.

[32] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 1–11.

[33] A. Shi, W. Lam, R. Oei, T. Xie, and D. Marinov, "ifixflakies: A framework for automatically fixing order-dependent flaky tests," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 545–555.

[34] K. Herzig and N. Nagappan, "Empirically detecting false test alarms using association rules," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 39–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819018

[35] T. M. King, D. Santiago, J. Phillips, and P. J. Clarke, "Towards a bayesian network model for predicting flaky automated tests," in *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018, pp. 100–107.

[36] A. Bertolino, E. Cruciani, B. Miranda, and R. Verdecchia, "Know Your Neighbor: Fast Static Prediction of Test Flakiness," Jan. 2020. [Online]. Available: https://doi.org/10.32079/ISTI-TR-2020/001