# Shaker: a Tool for Detecting More Flaky Tests Faster

Marcello Cordeiro
*Universidade Federal de Pernambuco*
Recife, Brazil
msc3@cin.ufpe.br

Denini Silva
*Universidade Federal de Pernambuco*
Recife, Brazil
dgs@cin.ufpe.br

Leopoldo Teixeira
*Universidade Federal de Pernambuco*
Recife, Brazil
lmt@cin.ufpe.br

Breno Miranda
*Universidade Federal de Pernambuco*
Recife, Brazil
bafm@cin.ufpe.br

Marcelo d'Amorim
*Universidade Federal de Pernambuco*
Recife, Brazil
damorim@cin.ufpe.br

***Abstract*—A test case that intermittently passes or fails when performed under the same version of source code and test code is said to be flaky. The presence of flaky tests wastes testing time and effort. The most popular approach in industry to detect flakiness is ReRun. The idea behind ReRun is very simple: failing test cases are re-executed many times looking for inconsistencies in the output. Despite its simplicity, the ReRun strategy is very expensive both in terms of time and in terms of computational resources. This is particularly true for contexts where thousands of test cases are performed on a daily basis. Reducing the rerunning overhead is, thus, of utmost importance. This paper presents SHAKER, an open-source tool for detecting flakiness in time-constrained tests by adding noise in the execution environment. The main idea behind SHAKER is to add stressing tasks that compete with the test execution for the use of resources (CPU or memory). SHAKER is available as a GitHub Actions workflow that can be seamlessly integrated with any GitHub project. Alternatively, SHAKER can also be used via its provided Command Line Interface. In our evaluation, SHAKER was able to discover more flaky tests than ReRun and in a faster way (less re-executions); besides, our approach revealed tens of new flaky tests that went undetected by ReRun even after 50 re-executions. Thanks to its flexibility and ease of use, we believe that SHAKER can be useful for both practitioners and researchers.**
**Demo video: https://youtu.be/7-aiQwOb4rA**
**Shaker website: https://star-rg.github.io/shaker**

***Index Terms*—software testing, regression testing, continuous integration**

## I. INTRODUCTION

A test case that intermittently passes or fails when performed under the same version of source code and test code is said to be flaky. The presence of such tests negatively affect regression testing for many reasons: Flaky tests waste developer's time in case they need to debug a failing test that is related to flakiness rather than an actual regression in the system under test. In addition to that, if the presence of flaky tests is recurrent, developers' confidence in test results is affected and they can even choose to ignore test failures sometimes [1]. This problem is worsened in a Continuous Integration environment, where, ideally, all tests must pass before a change can be integrated.

In recent years, many approaches and tools have been proposed to address the test flakiness problem [2]–[7]. Bell et al. [2] proposed DeFlaker, a technique that identifies flaky tests by monitoring the coverage of latest code changes. The technique proposed in [2] is dynamic because it relies on the test execution results. In addition to the dynamic techniques, researchers have also proposed approaches to statically predict the likelihood of a test being flaky during execution. For example, Pinto et al. [3] proposed a classification-based approach to statically analyze the test code and identify tests that are likely flaky. ReRun, which is the most popular approach in industry to detect test flakiness [8], [9], identifies flaky tests by repeating them a number of times: a failed test that passes in a subsequent run is considered flaky, and vice-versa.

Despite its simplicity, the ReRun strategy is very expensive: rerunning tests multiple times can be very costly both in terms of time and in terms of computational resources. This is particularly true for contexts where thousands of test cases are performed on a daily basis. At Microsoft, for example, 4.6% of the tests were identified as flaky after monitoring five projects over a one-month period [10]. Google reports that between 2-16% of its testing budget is consumed just to rerun flaky tests [8]. Reducing the rerunning overhead is, thus, of utmost importance.

In our prior work [11] we used a prototype tool for evaluating how the addition of noise in the execution environment would influence the detection of flakiness in time-constrained tests. The results were encouraging: our approach was able to discover more flaky tests than ReRun and in a faster way (less re-executions); besides, our approach revealed tens of new flaky tests that went undetected by ReRun even after 50 re-executions. Motivated by these results, we then made several changes to our initial prototype [11] to make it robust and usable on real, large software projects, as well as extending it for using it in different contexts.

In this work we present SHAKER, a tool for detecting

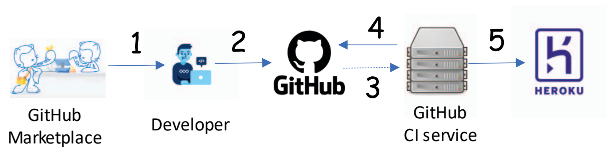

Fig. 1. SHAKER's GitHub Actions workflow. At step 1, the developer copies the template of the Shaker GitHub action, available on the GitHub Marketplace or from our web site, to update her .github/workflow/main.yml file. At step 2, the developer makes a push or pull request (configurable) to her GitHub repository. At step 3, GitHub notifies its CI service about that event. At step 4, the CI service pulls the changes from the corresponding commit from GitHub and runs the SHAKER action within a Linux container that is prepared with a tool for stressing the resources of the machine (`stress-ng`). Finally, at step 5, SHAKER notifies a web service, hosted in Heroku (could be any PaaS host), to store telemetry data about the execution.

flakiness in time-constrained tests by adding noise in the execution environment. The main idea behind SHAKER is to add stressor tasks that compete with the test execution for the use of resources — CPU or memory. Such idea derives from the observations that *i*) concurrency is an important source of flakiness [12], [13]; and *ii*) adding noise in the execution environment can change the ordering of events and, consequently, influence the test outputs.

SHAKER is available as a GitHub Actions workflow that can be seamlessly integrated with any GitHub project. Alternatively, SHAKER can also be used via its provided command line interface, similar to our original prototype tool. We envision two main use cases for SHAKER:

- developers concerned with flaky tests can integrate the GitHub Action into their project repositories and be notified when new flaky tests are identified by our tool.
- researchers can use SHAKER for building data sets of flaky tests for conducting research.

SHAKER source code is publicly available on GitHub at https://star-rg.github.io/shaker.

## II. USAGE

SHAKER can be used in two ways, via the GitHub Action that can be integrated directly into a GitHub repository or via its provided command line interface.

*a) GitHub Action:* Figure 1 illustrates the workflow associated with the execution of SHAKER's GitHub Action. To use SHAKER's GitHub Action in a repository, the developer needs to include the code from Listing 1 in a repository's workflow file as a new job. The effect of that inclusion is to create a new job, `shaker`, that is executed when a specified workflow is triggered. The three arguments declared on the listing —`tool`, `no_stress_runs`, and `runs`– are the configurable inputs from the user, and are further described in subsection III-B.

After finishing the workflow run, the results are displayed in the Actions tab on the GitHub repository. If no test failures have been detected, the job is marked as successful. If test

```
shaker:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - name: Shaker
      uses: STAR-RG/shaker@main
      with:
        tool: maven
        runs: 3
        no_stress_runs: 1
```

Listing 1: SHAKER GitHub Actions configuration

failures have been detected, the job is marked as unsuccessful, and SHAKER reports the failing tests.

*b) CLI:* For using SHAKER via its provided CLI the user needs to clone the source code via `git clone https://github.com/STAR-RG/shaker`. After that, SHAKER can be invoked with the following command `shaker/shaker.py --no-stress-runs 1 --stress-runs 4 {pytest,maven} directory`, where `<directory>` refers to the path of the project to be tested. The number of stressless and stressful runs can also be configured by the user. After the execution is complete, SHAKER reports whether flaky tests were found.

In what follows, we provide further details over the technique and implementation.

## III. TECHNIQUE AND IMPLEMENTATION

We describe the technique implemented in SHAKER, and the corresponding GitHub Action.

### A. Technique

The goal of SHAKER is to detect flaky tests. It receives, as input, the path to a target Java or Python project and outputs a report listing the tests identified as flaky.

SHAKER builds on the observation that flakiness very often occurs because of concurrency issues, including asynchronous wait [1], [12], [13]. One example scenario of flakiness due to asynchronous wait occurs when a test attempts to access a server —that it itself spawned— before that server is available to receive requests. The test fails in a non-deterministic way, depending on speed of the threads that execute the test code and the server. The test fails if the thread that executes the test runs faster compared to the thread that runs the server. Our insight for SHAKER is that adding noise in the execution environment could exercise different thread interleaving associated with test runs and, consequently, manifest failures. Our hypothesis, validated in a previous study [11], was that SHAKER reveals flaky tests more promptly, with less (re)executions, as compared with rerunning tests without noise. Detecting flakiness not caused by concurrent behavior is not the intent of SHAKER.

SHAKER runs a project test suite under a noisy environment, i.e., it imposes a load on the CPU or memory (Section III-B). In our previous study [11], we have evaluated different noise configurations and found four such configurations that showed better performance on revealing flaky tests when compared to ReRun. SHAKER runs the entire test suite four times, each time with a different noise configuration. At the end of those executions, SHAKER reports to the developer the tests that manifested flaky behavior (i.e., discrepant pass-fail outcomes), the amount of times that the test failed, and the stack trace. Moreover, SHAKER can also be optionally used to perform ReRun (in a noiseless environment). SHAKER provides different configuration knobs. For instance, the users can indicate how many times the four selected configurations will be executed.

## B. Implementation

The core application of SHAKER is developed in Python. For the integration with GitHub projects, SHAKER relies on GitHub Actions, a GitHub service that allows developers to automate tasks within the software development life cycle [14]. It offers free CI/CD services and can be seamlessly integrated into any repository in GitHub.

Shaker is configurable in three ways:

- **Testing tool**: the tool used for running the test cases. Currently, SHAKER supports Java (`maven-based`) and Python projects (`pytest-based`);
- **Number of no-stress runs**: the number of runs without any stressing in the execution environment. This is an optional parameter and if the user does not provide a value, no stressless runs are performed;
- **Number of stressing runs**: the number of runs for each stress-ng configuration. For example, if the user inputs 3, since there are four different stress-ng configurations, the tests will run 12 times in total.

SHAKER relies on stress-ng [15] for creating the noisy environment for the stressing runs. In particular, SHAKER makes use of 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*.

The SHAKER GitHub Action is implemented as a Docker image containing the core application and it is configured to support repository integration in GitHub. The Docker image is based on Ubuntu 20.04 and it contains all of the tools and dependencies required by SHAKER. As required for all GitHub Actions, a YAML file specifies the Action inputs and how to execute it.

## IV. EVALUATION

The prototype tool from which we implemented SHAKER was evaluated on a sample of 11 Android apps [11]. SHAKER discovered more flaky tests than ReRun 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.

We have also performed a preliminary evaluation of the GitHub Actions integration. To conduct this evaluation, we have used 11 projects as objects of analysis based on the following criteria:

- The project is written in Java and uses Maven as its project management tool;
- The project has more than 1000 test cases or the repository has more than 1000 stars.

The list of repositories, followed by the ref used for analysis (release tag or commit SHA-1 hash), number of tests, and number of stars is presented in Table I.

TABLE I
OBJECTS OF ANALYSIS

| Repository | Ref | Tests | Stars |
|---|---|---|---|
| `Azure/azure-iot-sdk-java` | a9226a5 | 4563 | 153 |
| `CorfuDB/CorfuDB` | b99ecff | 954 | 541 |
| `OpenHFT/Chronicle-Queue` | bec195b | 408 | 2.3k |
| `soabase/exhibitor` | d345d2d | 52 | 1.7k |
| `vaadin/flow` | 6.0.6 | 4679 | 304 |
| `apache/hbase` | d50816f | 6024 | 4k |
| `intuit/karate` | 09bc49e | 529 | 4.7k |
| `killbill/killbill` | killbill-0.22.21 | 1828 | 2.3k |
| `mock-server/mockserver` | b1093ef | 3532 | 3.2k |
| `apache/ozone` | dfd2aaf | 1900 | 359 |
| `RipMeApp/ripme` | 19ea20d | 247 | 2.4k |

To perform the experiment with all projects in parallel, a workflow file containing 11 jobs, one for each repository, was created. We have configured a specific action, named `actions/checkout`, to clone each repository with a given commit hash or release tag, and allow the subsequent actions to use the given repository. Listing 2 shows a code snippet illustrating how a job can be created to perform such task.

```
azure-iot-sdk-java:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
      with:
        repository: Azure/azure-iot-sdk-java
        ref: a9226a5...
    ...
```

Listing 2: Using SHAKER in other repositories

After a workflow run was finished, results for each run were displayed in the workflow logs for each job. If all tests always fail, the module containing these tests was discarded because they show defective tests, broken modules or broken dependencies, and thus are not classified as flaky tests.

We configured SHAKER to perform 4 stressing runs for each noiseless run. The only exception was `apache/hbase`, because more than one no-stress run and one set of stressing runs caused the job to take longer than 6 hours, triggering a timeout from GitHub Actions, and cancelling the job.

Table II shows the test failures discovered. We can observe that stress runs result in more failures observed. On average, a workflow configured to run 1 no-stress run and 4 sets of stress runs took 4 hours. Table III shows the ratio of failures discovered per run, which highlights the effect of using SHAKER. The numbers show that stress runs are 1.13 to 6.25 times more likely to find flaky tests than standard ReRun.

TABLE II
FAILURES

| Repository | flakies | no-stress | stress |
|---|---|---|---|
| `OpenHFT/Chronicle-Queue` | 6 | 4 | 38 |
| `soabase/exhibitor` | 3 | 1 | 34 |
| `apache/hbase` | 2 | 1 | 19 |
| `RipMeApp/ripme` | 9 | 2 | 55 |

TABLE III
FAILURE RATIO

| Repository | flakies | no-stress | stress |
|---|---|---|---|
| `OpenHFT/Chronicle-Queue` | 6 | 0.15 | 0.17 |
| `soabase/exhibitor` | 3 | 0.04 | 0.15 |
| `apache/hbase` | 2 | 0.04 | 0.25 |
| `RipMeApp/ripme` | 9 | 0.07 | 0.25 |

## V. LIMITATIONS AND FUTURE WORK

Currently, SHAKER only supports Java projects using `maven` or Python projects using `pytest`. However, SHAKER is easily extendable and we plan to support other build automation tools and test frameworks in future releases.

Another current limitation of SHAKER refers to the stressing configurations used. SHAKER comes with a set of predefined stressing configurations that have proven to be effective in identifying flakiness in time-constrained tests. These configurations were selected after a large empirical evaluation conducted in our previous study [11]. Nevertheless, we acknowledge that different projects or different contexts might require a different set of stressing configurations. To address this limitation we plan to release a self-adaptive version of SHAKER that customizes the stressing configurations for the particular project using it, based on previous results.

## VI. CONCLUSION

In this paper we introduced SHAKER, an open-source tool for detecting flakiness in time-constrained tests by adding noise in the execution environment. SHAKER is available as a CLI and as a GitHub Action that can be seamlessly integrated with any GitHub project. We discussed the noise technique on which SHAKER relies, and we presented the usage, design, and implementation of SHAKER. We also discussed current limitations and highlighted some future research and development directions. Our evaluation of SHAKER showed encouraging results. In the evaluation using the CLI, SHAKER was able to discover more flaky tests than ReRun and in a faster way (less re-executions); besides, our approach revealed tens of new flaky tests that went undetected by ReRun even after 50 re-executions. An additional evaluation of the SHAKER Action reported similar results and the noisy runs conducted by SHAKER were up to 6.25 times more likely to find flaky tests than the standard ReRun strategy. Thanks to its flexibility and ease of use, we believe that SHAKER can be useful for practitioners and researchers. As future work, we intend to explore other ways of introducing noise in the execution environment, such as running tests in parallel, which might even reveal flakiness due to test order dependency, for instance. We also intend to perform qualitative assessments of SHAKER with developers.

## ACKNOWLEDGMENT

This work is partially supported by INES (www.ines.org.br), CNPq grant 465614/2014-0, CAPES grant 88887.136410/2017-00, and FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14. Denini was supported by a FACEPE fellowship number IBPG-1279-1.03/19.

## REFERENCES

[1] 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.

[2] 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.

[3] 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*, 2020, pp. 492–502.

[4] 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.

[5] 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)*. IEEE, 2019, pp. 312–322.

[6] 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.

[7] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021.

[8] 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.

[9] 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/.

[10] 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: ACM, 2019, pp. 101–111. [Online]. Available: http://doi.acm.org/10.1145/3293882.3330570

[11] D. Silva, L. Teixeira, and M. d'Amorim, "Shake it! detecting flaky tests caused by concurrency with shaker," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 301–311.

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

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

[14] GitHub, "GitHub Actions," https://github.com/features/actions.

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