A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests

Valerio Terragni
USI Università della Svizzera italiana
Switzerland
valerio.terragni@usi.ch

Pasquale Salza
University of Zurich
Switzerland
salza@ifi.uzh.ch

Filomena Ferrucci
University of Salerno
Italy
fferrucci@unisa.it

ABSTRACT

Intermittent test failures (test flakiness) is common during continuous integration as modern software systems have become inherently non-deterministic. Understanding the root cause of test flakiness is crucial as intermittent test failures might be the result of real non-deterministic defects in the production code, rather than mere errors in the test code. Given a flaky test, existing techniques for root causing test flakiness compare the runtime behavior of its passing and failing executions. They achieve this by repetitively executing the flaky test on an instrumented version of the system under test. This approach has two fundamental limitations: (i) code instrumentation might prevent the manifestation of test flakiness; (ii) when test flakiness is rare passively re-executing a test many times might be inadequate to trigger intermittent test outcomes. To address these limitations, we propose a new idea for root causing test flakiness that actively explores the non-deterministic space without instrumenting code. Our novel idea is to repetitively execute a flaky test, under different *execution clusters*. Each cluster explores a certain non-deterministic dimension (e.g., concurrency, I/O, and networking) with dedicated software containers and fuzzy-driven resource load generators. The execution cluster that manifests the most balanced (or unbalanced) sets of passing and failing executions is likely to explain the broad type of test flakiness.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computer systems organization** → Cloud computing; • **Hardware** → Testing with distributed and parallel systems.

KEYWORDS

Test Flakiness, Non-Determinism, Concurrency, Cloud, Root-Causing Analysis, Fuzzy Analysis, Software Containers

ACM Reference Format:
Valerio Terragni, Pasquale Salza, and Filomena Ferrucci. 2020. A Container-Based Infrastructure for Fuzzy-Driven Root Causing of Flaky Tests. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381742

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381742

1 INTRODUCTION

Software constantly evolves to add and improve features, remove faults, change the design etc. As software evolves, developers have to ensure that changes do not modify software behaviors in unintended ways [13]. They often achieve this by automatically run the existing regression tests with the continuous integration pipeline. If all tests pass the developers have confidence that the code changes do not break existing functionalities [13]. If some tests fail, developers assume that the changes introduced one or more faults and they will start the debugging process.

However, developers sometimes realize that there is nothing wrong with the code changes, and tests fail intermittently because testing became inherently non-deterministic [1, 5]. This kind of tests are called **"flaky"** (*non-deterministic*), that is, a test that passes or fails intermittently for the same code version, the same inputs, and the same configuration [9]. Test flakiness has become a serious and acknowledged issue in modern software development, as the complexity and pervasiveness of non-deterministic components (e.g., concurrency, networking an I/O) have increased in software [7]. In recent years, many developers regard test flakiness as one of the primary concern for industrial software testing [1, 5]. Large organizations such as Facebook, Google, Huawei, Microsoft, and Mozilla are increasingly reporting problems with flaky tests [8]. A key issue is that flaky tests lead to unreliable signals that can erode the trust of developers in their regression testing [1, 5, 8, 10].

To identify and fix flaky tests, the most common approach is to execute tests many times to see if they fail intermittently. For example, using the JUnit `@RepeatedTest` annotation. In some cases, the cause of the intermittent test failure is obvious, and a developer can fix the test easily. Nevertheless, the problem arises when it is not immediately obvious why a test is flaky. In this case, developers need to acquire more information about the failure itself. More specifically, the developers need to identify the **"root cause of test flakiness."** Root causing flaky tests is both difficult and time-consuming, mainly because often one has to execute a flaky test many times before observing a non-deterministic behavior [7]. As such, developers may decide to simply ignore and disable flaky tests [9] assuming that the intermittent failures are tests problems [8].

Ignoring flaky test failures is dangerous as they could be symptoms of real intermittent bugs in the production code, rather than mere non-deterministic tests [2]. Sometimes a flaky test is flawless and it reveals a non-deterministic fault in the production code [7, 10]. For example, a flaky behavior could be caused by concurrency faults, which often manifest non-deterministically under specific thread interleavings [12]. Even if developers do not write tests to intentionally validate concurrency behaviors, tests may invoke concurrent components that could trigger concurrency faults. As

such, ignoring flaky tests is dangerous because it may ignore real non-deterministic faults during the continuous integration [12].

Although root causing flaky tests is a labor and time-consuming activity, it is crucial to guarantee reliable production code. Given the prevalence of flaky tests in modern software systems [2, 3, 7], cost-effective tools and methodologies for automatically identifying the root cause of test flakiness are highly needed and demanded.

In this paper, we present a new idea to discover test flakiness and its root causes. It employs software containers to distribute several multiple executions of the test under analysis, each of them defining possible configurations and conditions that may lead to flakiness. In the rest of the paper, we present the state of the art, highlighting the limitations we believe should be addressed to improve root causing of flaky tests. Then, we describe our vision, together with the proposal for a novel infrastructure to cope with the problem.

2 STATE OF THE ART

Understanding the root cause of test flakiness is simple in theory: given a flaky test, an automated tool could compare the run-time behaviors of the passing and failing executions of the test. It is very similar to what techniques for fault localization do, such as Tarantula [6]. However, achieving this in a cost-effective way for test flakiness is difficult in practice [7]. Existing techniques, either rely on code instrumentation [7], which may be disruptive and incomplete, or passively explore the non-deterministic space by simply re-executing many times the flaky tests [2], which may be inadequate to reproduce the intermittent failures.

Limitations of instrumentation-based approaches. Lam et al. recently proposed RootFinder [7], which identifies the root cause of flaky tests by finding differences in the logs of passing and failing executions. It obtains such logs by running the flaky tests with an instrumented version of the production code. However, relying on code instrumentation entails three fundamental limitations.

First, the instrumentation may disrupt the test execution and therefore prevent the manifestation of flaky test failures (e.g., due to instrumentation runtime overhead). Indeed, Lam et al. observed that some flaky tests exhibit intermittent failures only when they run without instrumentation [7].

Second, it is impossible to know in advance all the execution points that we should instrument and what information should be collected to be able to understand the runtime differences. There are different sources of test flakiness [9], for example, concurrency, test ordering, I/O, and networking. Each source requires collecting different runtime information at different execution points [7]. Collecting all the information is often infeasible, because of the high runtime overhead [2, 7]. Moreover, the more information we collect, the more expensive the differential analysis becomes [6, 7].

Third, instrumentation often introduces significant slowdown. This is a problem because often one has to re-execute flaky tests many times before observing a test failure [4, 9].

To avoid such limitations and obtain a non-disruptive root cause analysis we need new methodologies that root cause flaky tests, but without relying on instrumentation. iDFlakies [8] is an example of an instrumentation-free approach. However, it can detect and root-causing only one kind of flaky tests, i.e., those that have non-deterministic behaviors due to test dependency orders [8].

Limitations of exploring the non-deterministic space passively. The standard practice to detect [2] and debug [7] a flaky test is to repetitively re-execute the test, hoping that it would manifest an intermittent test outcome [1]. This practice may need many re-executions of the test, when test flakiness occurs rarely [2, 7]. For example, because of rare non-deterministic execution orders among threads. Moreover, re-executing a flaky test many times manifests test flakiness only if the execution environment that causes intermittent test outcomes is properly exercised [8]. For example, a test flaky may fail only under certain network bandwidth [9].

This practice passively explores the non-deterministic space, and thus it may repetitively explore the same test execution environments and configurations. Thus, it may not be able to expose intermittent failures or it may lead to unbalanced sets of passing and failing executions. Having only a few failing executions to compare with the passing counterparts is known to compromise the effectiveness of differencing tools [6, 7].

An approach that actively explores the non-deterministic space increases the chance of exposing test flakiness and at the same time avoids to repetitively explore the same test execution environments.

3 METHODOLOGY

To address such challenges, we propose a new methodology for root causing flaky tests that explores the non-deterministic execution space actively, without relying on any form of instrumentation.

Our core idea is to develop a root cause analysis approach that identifies the root cause of test flakiness by executing a (potential) flaky test under different **"execution clusters"**. Each execution cluster actively explores a specific non-deterministic execution space by fuzzing the execution environment of the test across a particular dimension (e.g., concurrency, networking, I/O). A (flaky) test will be executed a fixed number of times by each of the clusters. The execution cluster that manifests the most balanced (or unbalanced) sets of passing and failing executions likely characterizes the broad type of non-determinism that explains the test flakiness. This is because each execution cluster actively explores a particular non-deterministic dimension. Once our approach identifies the general type of test flakiness, one could perform a targeted and fine-grained root-cause analysis (if needed). By knowing the type of test flakiness, a developer should know which execution points to instrument and which information to collect. Thus alleviating the fundamental limitations of code instrumentation discussed earlier.

The execution clusters will rely on both software containers technology and fuzzy resource load generators to explore the non-deterministic space. The software containers will be used as a form of lightweight virtualization to allocate a wide spectrum of available resources (e.g., the number of cores, network bandwidth, disk size). Instead, fuzzy load generators will be used to occupy the available resources with dummy and random operations. Fuzzy load generators will be carefully designed to both free and occupy resources as test flakiness might occur either way. For example, data races might occur because a certain thread runs faster or slower [12]. The synergistic combination of available resources and dummy operations would help us to explore a wide spectrum of the non-deterministic execution space.

Figure 1 shows the overview of the proposed infrastructure. We define an execution cluster as a specific combination of a family of containers and a fuzzy-load generator. Each cluster explores a specific non-deterministic dimension, which can cause test flakiness. Each container runs the test under analysis in a specific environment, associated to a possible configuration in the non-deterministic execution space. A *Fuzzy Orchestrator* would be responsible to manage the execution clusters, dynamically changing the software containers and the load of the fuzzy-load generator. It would select the next container and fuzzy-load either randomly or adaptively (based on the behavior of the test under analysis).

In the next subsections, we define in more details our envisioned methodology.

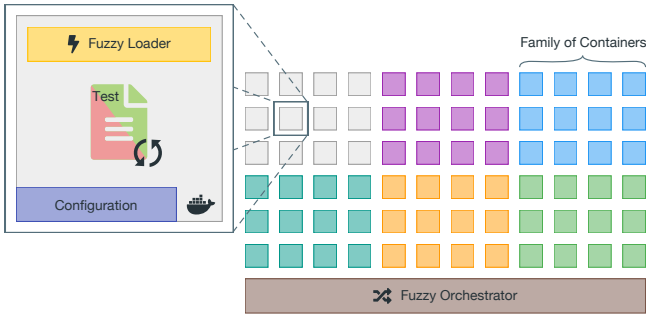

Figure 1: Overview of the proposed infrastructure

3.1 Execution Clusters

Our envisioned execution clusters are based on software container technology. We selected such technology because it offers a light-weight form of virtualization that allows a fast execution of the environments [11]. The environments themselves can be expressed in terms of a configuration file, e.g., `Dockerfile`. Then, the containers can be easily built in the form of images, which are ready to be executed. In addition, the existing platforms for software containers management, e.g., Docker, allow users to define the characteristics of the virtual machines to be run. In this way, we can define the hardware conditions on which the (flaky) test should be executed for exploring a specific non-deterministic dimension. The underneath behavior of containers is regulated by the presence of the fuzzy-load generators who characterize the clusters and perturb the normal execution of the test.

The study by Luo et al. [9] identified the most common test flaky root causes to be *async wait concurrency*, *test order dependency*, *resource leak*, *network*, *time*, *I/O*, *randomness*, *floating point operations*, and *unordered collections*. We envision the following five execution clusters, which are expected to cover the most popular root causes of test flakiness [9]. Of course, the list is not restricted to the clusters that we are presenting in the following. More clusters can be defined to cover additional types of flaky tests.

Multi-threaded execution cluster explores the non-deterministic interleaving space of concurrent executions. This execution cluster characterizes test flakiness due to internal concurrency, which is the most common type of test flakiness [9].

The family of containers varies the execution architecture by allocating different numbers of cores. It considers both few cores (e.g., 2) and many cores (e.g., 100), as the non-determinism due to concurrency manifests because thread runs faster or slower. The fuzzy load generator spawns and runs various concurrent threads that execute dummy operations to occupy resources. The number of spawned threads and the number of dummy operations will constantly change to apply a different degree of multi-core contention. Because the test under analysis runs in parallel with the fuzzy load generator, occupying CPU resources with dummy operations help to explore the interleaving space of the test without using instrumentation.

Network execution cluster explores the non-deterministic space of the network latency and response time, characterizing flaky tests due to sharing resources with other components on the same network, or on the web. It includes non-determinism caused by external concurrency and network connection failures [9].

The family of containers varies the execution architecture by increasing/decreasing the network bandwidth and by switching off the network for a certain time (e.g., 1 ms to 100 ms). A remote connection failure is one of the major cause of test flakiness [9]. The load generator occupies the network, causing traffic and congestion. For example, it can achieve this by downloading files or by randomly browsing the web using a web crawler. The load generator will choose from a wide spectrum of types of communication protocols: (i) basic data communication (TCP/IP and HTTP); (ii) network security (HTTPS, SSL and SFTP); and (iii) network management (SNMP and ICMP). Using different communication protocol may change the network occupancy, introducing congestion because different protocols communicate on different ports.

I/O execution cluster explores the non-determinism of I/O operations, which characterizes flaky tests due I/O and resource leaks [9].

The family of containers varies the execution architecture by allocating various disk space to explore the scenario that the disk has no free space or becomes full while a test is writing a large file. The load generator creates dummy I/O operations, for example, writing and reading text files. This would create resource contention to those I/O operations that the test indirectly or directly performs. The load generator will also activate the garbage collection at random intervals. This is because whether a resource was already garbage collected or not might affect the test outcome [9].

Test order execution cluster explores the non-determinism caused by test dependencies, as a common type of test flakiness is test dependency order. More specifically, a test execution might modify the state of the system, and thus the tests executed after this test might be affected by this new state [8].

The family of containers executes the other available tests in a different order before executing the test under analysis. The load generator does not perform operations.

Platform cluster explores the non-determinism caused by different execution platforms.

The family of containers executes the test under different operating system platforms and versions, library versions, and tool configurations. The load generator does not perform operations.

3.2 Root-Cause Analysis

Our root-cause analysis is based on the test outcomes reported by each execution cluster after a fixed number of executions ($N > 1$). In addition to the five execution clusters described in Section 3.1 ($\mathcal{E}_1, \mathcal{E}_2, \ldots \mathcal{E}_5$), the Fuzzy Orchestrator executes the test t with a *baseline execution cluster* $\mathcal{E}_0$, in which both the fuzzy loader and the containers do not perturb the normal execution of t. Our idea is to identify the root-cause of test flakiness by identifying the execution cluster $\mathcal{E}_k$ ($k > 0$) that most diverge from the baseline ($\mathcal{E}_0$).

More formally, let $exec_i(\mathcal{E}_k, t)$ denote the outcome of the i^{th} execution of t with the execution cluster $\mathcal{E}_k$. The outcome is either zero (the test passed) or one (the test failed). We now define the *failure rate* (λ) of the execution cluster $\mathcal{E}_k$ after it executes N times the test t, as:

$$\lambda_t^k = \frac{\sum_{i=1}^{N} exec_i(\mathcal{E}_k, t)}{N} \in [0; 1]$$

The failure rate is one (zero) when t failed on all (none) of the N test executions that $\mathcal{E}_k$ performed. We say that a test is flaky on the execution cluster $\mathcal{E}_k$, if λ_t^k is different from both zero and one. Our envisioned root-cause analysis works as follows.

If $\lambda_t^0 = 0$, the test t is not flaky on the baseline execution cluster. In this case, the most plausible root cause for test flakiness is given by the execution cluster with the highest failure rate ($\max_k \lambda_t^k$). For example, if the failure rate of the network execution cluster is higher than the failure rate of all other execution clusters, it is likely that t is flaky due to network non-determinism.

If $\lambda_t^0 \neq 0$, the test t is flaky on the baseline execution cluster. Here the root cause analysis becomes more tricky because presumably if a test is flaky on the baseline cluster it will be so also in other execution clusters. In this case, the most plausible root cause is given by the execution cluster with the most diverse failure rate with respect to λ_t^0, i.e., $\max_k |\lambda_t^k - \lambda_t^0|$. Intuitively, the fuzzy perturbation that belongs to the root cause could either decrease or increase the flakiness, and thus $\max_k |\lambda_t^k - \lambda_t^0|$ captures both scenarios. For example, let assume that a test is flaky because a thread always runs faster than another thread. If it occurs in the baseline cluster often, the fuzzy perturbations of the multi-threaded execution cluster are likely to perturb such ordering of threads as well. In this case, the multi-threaded execution cluster ($\mathcal{E}_1$) will be reported as the most plausible one since $|\lambda_t^1 - \lambda_t^0|$ would have the highest value.

Our root-cause analysis assumes that the each flaky test can have exactly one source of flakiness, which is the most common situation [2, 7]. Creating hybrid combinations of multiple clusters could be an effective approach to handle test flakiness caused by the interaction of two or more flakiness sources.

Given a (flaky) test t, the root-cause analysis reports: (i) the non-determinism associated with the most plausible execution cluster; (ii) the association of the configurations of the fuzzy-loader and container with the test outcome (pass or fail). Such information help developers to root-cause test flakiness. For instance, if a configuration with a low network bandwidth is often associated with test failures, this is the most likely reason for test flakiness.

Moreover, the exact configurations of the execution clusters that trigger test flakiness can also help developers to reproduce the test flakiness facilitating the debugging.

4 OUR VISION

In this paper, we propose a new idea for root-causing flaky tests. We envision our approach to be deployed in two scenarios: (i) *during continuous integration* for both detecting and root-causing flaky tests; (ii) *after continuous integration* for root causing only those tests that were deemed flaky during continuous integration. Developers can choose one of these scenarios, depending on their needs and the available budget.

We envision to use distributed software containers platforms, e.g., Docker Swarm, and Kubernetes, to implement our approach. These platforms transform physical hardware into distributed environments on which multiple software containers can run in parallel.

Note that, our infrastructure is general enough to allow root-cause analyses more sophisticated to the one presented in this paper. For example, machine learning or statistical approaches could be key tools for achieving precise results.

ACKNOWLEDGMENTS

This work is partially supported by the Swiss SNF project ASTERIx (SNF 200021_178742).

REFERENCES

[1] Nadia Alshahwan, Andrea Ciancone, Mark Harman, Yue Jia, Ke Mao, Alexandru Marginean, Alexander Mols, Hila Peleg, Federica Sarro, and Ilya Zorin. 2019. Some Challenges for Software Testing Research. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1–3.

[2] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 433–444.

[3] Zebao Gao, Yalan Liang, Myra B Cohen, Atif M Memon, and Zhen Wang. 2015. Making System User Interactive Tests Repeatable: When and What Should We Control?. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 55–65.

[4] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 223–233.

[5] Mark Harman and Peter W. O'Hearn. 2018. From Start-Ups to Scale-Ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–23.

[6] James A Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 273–282.

[7] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 101–111.

[8] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 312–322.

[9] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 643–653.

[10] Md Tajmilur Rahman and Peter C. Rigby. 2018. The Impact of Failing, Flaky, and High Failure Tests on the Number of Crash Reports Associated with Firefox Builds. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 857–862.

[11] Pasquale Salza and Filomena Ferrucci. 2019. Speed Up Genetic Algorithms in the Cloud Using Software Containers. *Future Generation Computer Systems* 92 (March 2019), 276–289.

[12] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. 2015. RECONTEST: Effective Regression Testing of Concurrent Programs. In *IEEE/ACM International Conference on Software Engineering (ICSE)*. 246–256.

[13] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification & Reliability* 22, 2 (2012), 67–120.