De-Flake Your Tests
Automatically Locating Root Causes of Flaky Tests in Code At Google

Celal Ziftci
Google Inc.
New York, USA
celal@google.com

Diego Cavalcanti
Google Inc.
New York, USA
diegotc@google.com

***Abstract*—Regression testing is a critical part of software development and maintenance. It ensures that modifications to existing software do not break existing behavior and functionality.**

One of the key assumptions about regression tests is that their results are deterministic: when executed without any modifications with the same configuration, either they always fail or they always pass. In practice, however, there exist tests that are non-deterministic, called *flaky* tests. Flaky tests cause the results of test runs to be unreliable, and they disrupt the software development workflow.

In this paper, we present a novel technique to automatically identify the locations of the root causes of flaky tests on the code level to help developers debug and fix them. We study the technique on flaky tests across 428 projects at Google. Based on our case studies, the technique helps identify the location of the root causes of flakiness with 82% accuracy. Furthermore, our studies show that integration into the appropriate developer workflows, simplicity of debugging aides and fully automated fixes are crucial and preferred components for adoption and usability of flakiness debugging and fixing tools.

***Index Terms*—Software maintenance, Diagnostics, Debugging aids, Debuggers, Tracing, Test management, Flaky tests**

I. Introduction

Regression testing is a critical part of software development [1], [2]. When developers add new functionality to a system, they run the regression test suite to ensure that their current changes did not inadvertently change existing functionality. If all tests in the regression test suite pass upon a code change, developers typically consider the test results as a success and continue to submit their change. However, if any of the tests fail, they typically investigate the reasons for the failure [3]. As a result, regression tests provide developers with a critical signal on whether they can submit their changes safely. The same signal is typically used in additional steps downstream in the development workflow, e.g. during release time, the new version of the system is rolled out only if all tests in the regression suite pass.

It is important that this signal is consistent and deterministic, i.e. if the test suite is executed without any changes with the same configuration parameters, they should either always pass or always fail. Unfortunately, there might be non-deterministic, so called *flaky* [4]–[9], tests in the test suite. Flaky tests are problematic because they introduce noise into the signal produced by the execution of the test suite [10]–[12].

First, flaky tests can be hard to debug because they are non-deterministic, so it might be difficult to reproduce their behavior during debugging. Second, they may cause developers to waste time by failing on unrelated code changes [7], [10], [12]. When developers change the code, run the test suite, and observe failures, they typically try to debug it to understand what caused the failures. If the failure is due to a flaky test, and not their changes, the debugging time is wasted. Such false signals can cause continuous wasted efforts across many developers who change code that results in the execution of the test suite containing flaky tests, especially in large monolithic codebases like those used at Google [3], [12]. Third, flaky tests decrease developers' perception of trust on the test suite, and may mask actual failures. Whenever a test fails, if developers observe that test to have failed due to flakiness before, they might ignore the results of that execution, and might actually ignore real failures and accidentally introduce bugs into the system.

Flaky tests along with the problems they cause have been reported to exist in many systems, both by practitioners [4], [5], [8], [9], [12]–[16] and researchers [6], [7], [11], [17]–[26]. According to a recently published study by Luo et al. [11], 4.56% of all test failures across test executions at Google's continuous integration (CI) system, named TAP [12], [27], were reported to be due to flaky tests during a 15-month window. Another study by Herzig and Nagappan [16] reported that Microsoft's Windows and Dynamics product had an estimated 5% of all test failures due to flaky tests. Similarly, Pivotal developers estimated that flaky tests were involved in almost half of their test suite failures [28], while another study by Labuschagne [24] reported that 13% of test failures in TravisCI were due to flaky tests.

There are different strategies to deal with flaky tests and their disruption of the developer workflow. A commonly used one is to run a flaky test several times with the same configuration, and declare it to pass if at least one execution passes, and to fail if all runs fail. This strategy is used by Google's TAP system [4], [29], [30], and several open source testing frameworks support a similar notion through various strategies, such as annotations on flaky tests inside code, e.g. `@FlakyTest` in Android [31], `@Repeat` in Spring [32], `@RandomlyFails` in Jenkins [33], `rerunFailingTestsCount` property in Maven [34]. However, this is not ideal, especially for large codebases, since it wastes machine resources, and results can still be noisy depending on "how flaky" tests are (i.e. test suites that have a high probability of flaking require many more runs to get a passing run) [35], [36].

Another strategy is to ignore flaky tests completely (e.g. `@Ignore` in JUnit [37]) or separate tests that are known to be flaky to a different test suite, and treat the results of the execution of that test suite as "optional" for various software development activities [4]. Developers then investigate if any of the tests in the "optional" test suite start passing consistently, i.e. they are now deterministic, and move them back to the original test suite. This is also not ideal, because it requires manual work from developers, and deterministic failing tests can get lost in the noise among the other flaky tests in the optional test suite causing real failures to be ignored.

These strategies have serious downsides, and work around the problems introduced by flaky tests instead of fixing the root causes. It is important to fix flaky tests as quickly as possible to keep development velocity high and to produce a reliable signal for software development and maintenance activities. In this paper:

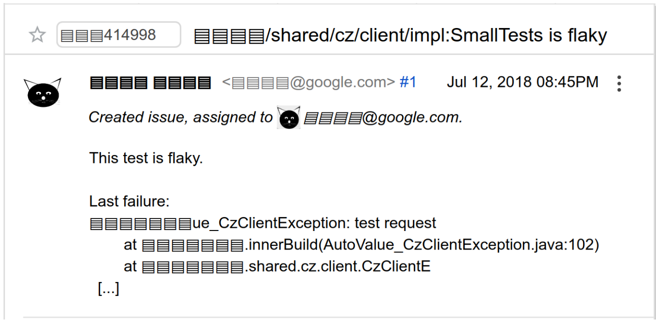
- We present a novel technique to automatically identify the locations of the root causes of flakiness in code,
- We implement this technique in a tool and deploy it across several products' flaky tests at Google to notify developers about the code location causing the flakiness,
- We assess the success of the technique on multiple case studies,
- We report our learnings on user perception and expectations of the tool and the technique.

II. Infrastructure Related to Flaky Tests At Google

To prevent and avoid the negative impact of flaky tests during the development workflow, several systems have been developed at Google. These systems have been integrated together to bring attention to flaky tests inside the developer workflow. Below we discuss a relevant subset of these systems.

TAP: This is the CI system at Google that runs unit-tests [12], [27]. TAP runs tests continuously during the course of a day at different versions. Developers are allowed to label their tests as flaky, and TAP runs such tests several times to check if at least one passing run can be obtained [38]. If so, the test is deemed to have passed.

Flakiness Scorer: This system assigns a flakiness score to each flaky test. It obtains information from TAP related to all executions of flaky tests, with the more recent executions having more weight. It then assigns a score indicating how likely those tests are to fail due to flakiness in the future and provides a web-based user interface to present that information. We use flakiness scores in our tool discussed in the next section.


☆ [redacted]414998 [redacted]/shared/cz/client/impl:SmallTests is flaky

[redacted] <[redacted]@google.com> #1 Jul 12, 2018 08:45PM

Created issue, assigned to [redacted]@google.com.

This test is flaky.

Last failure:

```
[redacted]ue_CzClientException: test request
      at [redacted].innerBuild(AutoValue_CzClientException.java:102)
      at [redacted].shared.cz.client.CzClientE
  [...]
```

Fig. 1. Web-based user interface for Google Issue Tracker. Users typically check Flakiness Scorer to see if they have flaky tests, then create an issue in Google Issue Tracker to investigate the test.

Google Issue Tracker: This is the issue tracking system used at Google [39]. Developers have the option to check if they have any flaky tests identified by Flakiness Scorer. If they do, they can then manually create issues and assign them to a team member to debug and identify the root cause of flakiness, shown in Figure 1. Once the root cause is identified and the flaky test is handled (e.g. fixed or removed entirely), they resolve the issue. In one of our case studies, we notify developers on how to debug/fix flaky tests by automatically commenting on the issues they created.

Critique: This is the web-based code review tool at Google [40]. When a developer changes code, they create a *change* with the modifications, and send it to other developers for review. Upon sending for review, several automated tools and regression tests run on the changed version of the code to help the owner and the reviewers with suggestions and fixes by showing notifications on relevant parts of the code, or the entire change itself. If the owner of the change finds any of these analyses unhelpful, they can provide feedback by clicking a *Not useful* button. In one of our case studies, we notify developers about flaky tests and how to debug/fix them.

III. Flakiness Debugger

In this section, we introduce a novel technique that can automatically identify the location of the root cause of a flaky test, explain the tool that implements this technique, and discuss its deployment across Google by integrating it into the developers' daily workflow.

A. Non-Determinism and Flakiness

There are many causes of non-determinism in tests, e.g. concurrency and test order dependency [6], [11], [38]. In this paper, we don't target classifying the type of flakiness into a taxonomy, distinguishing whether flakiness is in test code or system code, or finding the root cause of any *specific* type of flaky test. Instead, we propose to find the location of flakiness for any flaky test either in test or system code, and to show a report to developers to aid in debugging. Since developers are

```java
1  import java.util.Random;
2
3  final class RandomNumberGenerator {
4    private static final Random R = new Random();
5
6    /* Generate 0 or 1 with this one weird trick. */
7    public static int getRandomZeroOrOne() {
8      final int randomBtw0and99 = R.nextInt(100);
9      if (randomBtw0and99 < 50) {
10       return 0;
11     } else {
12       return 1;
13     }
14   }
15 }
```

Listing 1. Working example code.

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

final class RandomNumberGeneratorTest {

  /* Test that it always generates 0. */
  @Test
  public void testGetRandomZeroOrOne() {
    assertEquals(0, RandomNumberGenerator.
    getRandomZeroOrOne());
  }
}
```

Listing 2. Working example unit-test.

the domain experts of their own code, we leave it to them to identify and fix the root cause based on the report.

Listing 1 shows a working example used in the discussions throughout this paper. In Listing 1, `getRandomZeroOrOne()` generates a random number that is either 0 or 1, using another random number generator internally. There is non-determinism in the code between lines $8 - 13$. Assuming `R` is a good uniform random number generator, `getRandomZeroOrOne()` is expected to return 0 or 1 approximately 50% of the time respectively. In Listing 2, `testGetRandomZeroOrOne()` tests the functionality of `getRandomZeroOrOne()` from Listing 1 by asserting it always returns 0. This test is expected to fail 50% of the time, making it a flaky test.

The full list of all flaky tests at Google are determined and known by Flakiness Scorer.

B. Divergence

In this section, we propose the novel `DIVERGENCE` algorithm to identify where flakiness first occurs in code. The algorithm compares the execution traces of each failing run to all passing runs to find the first point of divergence in the control flow of the failing run from any of the passing runs, i.e. the point where a failing run's control flow has never been observed in a passing run, described in Algorithm 1. `DIVERGENCE` takes in a list of tests t and their corresponding passing and failing executions. For a failing execution f, we find the passing execution, p, that has the longest common prefix with f, and extract the common lines and the first diverging lines between f and p. We store this information for each flaky test.

Algorithm 1 `DIVERGENCE` algorithm

Require: T: $\{(t, P^t, F^t)$: Test t, execution traces sorted by control-flow time from passing runs P^t and failing runs $F^t\}$

```
1:  $Results \leftarrow \emptyset$
2:  for all $(t, P^t, F^t) \in T$ do
3:      for all $f \in F^t$ do
4:          $p \leftarrow argmax_{p_i \in P^t} longestCommonPrefix(p_i, f)$
5:          $commons \leftarrow findCommonPrefixLines(p, f)$
6:          $divergents \leftarrow findFirstDivergentLines(p, f)$
7:          $Results \leftarrow Results \cup (t, commons, divergents)$
8:      end for
9:  end for
10: return $Results$
```

For an example, consider that the method `testGetRandomZeroOrOne()` from Listing 2 is executed several times to obtain a passing run and a failing run. Figure 2 shows sample executions of that flaky test with a passing run on the left and a failing run on the right. Lines $7 - 9$ in `getRandomZeroOrOne()` are common to both passing and failing runs. Line 10 for the passing run and line 12 for the failing run are the points of divergence between the executions.

`DIVERGENCE` algorithm proposes to show the common lines along with the first divergence point across passing and failing runs to developers to help them understand where flakiness first gets introduced. There is further divergence in the control flow after the first divergence point, but those are ignored, as they don't add as much additional valuable information as the first point.

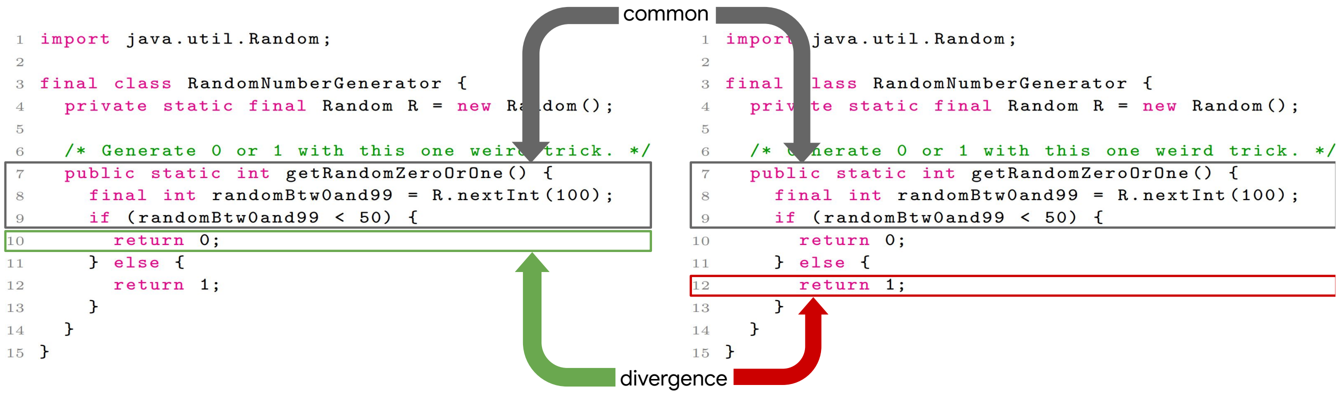
C. Finding Divergence for Flaky Tests

We implemented a tool at Google that uses `DIVERGENCE`, called Flakiness Debugger (FD). FD takes several steps to find root causes of flakiness for tests across many product groups at Google, summarized in Algorithm 2.

First, for FD to work for a project's flaky tests, it needs to be enabled by the owners of the project. FD finds all projects, P, where it is enabled, and identifies their tests, T^P.

Second, for all tests in T^P, FD queries Flakiness Scorer to check which of those tests are flaky, called T^F, and what their flakiness scores are. On a high level, for a given test t, Flakiness Scorer calculates the flakiness score $f(t)$ for t by checking how many times it flaked recently during its executions by TAP [38]. Using this score, FD skips t if $f(t) < M$, where M is a flakiness threshold that prevents running tests that are rarely flaky, and would use too many resources to get at least one failing run. We set $M = 0.1$.

Third, FD instruments the test t and the respective non-test code owned by the same team that owns t. This excludes any code that is owned upstream (e.g. `assertEquals` in



```java
import java.util.Random;

final class RandomNumberGenerator {
  private static final Random R = new Random();

  /* Generate 0 or 1 with this one weird trick. */
  public static int getRandomZeroOrOne() {
    final int randomBtw0and99 = R.nextInt(100);
    if (randomBtw0and99 < 50) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

```java
import java.util.Random;

final class RandomNumberGenerator {
  private static final Random R = new Random();

  /* Generate 0 or 1 with this one weird trick. */
  public static int getRandomZeroOrOne() {
    final int randomBtw0and99 = R.nextInt(100);
    if (randomBtw0and99 < 50) {
      return 0;
    } else {
      return 1;
    }
  }
}
```

Fig. 2. Common and divergence lines found by DIVERGENCE algorithm. When testGetRandomZeroOrOne() in Listing 2 is executed, if it passes, the execution will follow the flow on the left, while when it fails, it will follow the flow on the right. There are common lines in both executions, and diverging lines for passing and failing runs.

Algorithm 2 Flakiness Debugger (FD) algorithm

```
Require: FS: Flakiness Scorer
Require: $M$: Minimum flakiness score threshold for a test
Require: $E$: # times a flaky test is executed

 1: $P \leftarrow \{p : \text{FD is enabled for project } p, \forall p \text{ at Google}\}$
 2: $T^P \leftarrow \{t : \text{Test } t \text{ belongs to } p, \forall p \in P\}$
 3: $T^F \leftarrow \{(t, f(t)) : \text{FS reports } t \text{ is flaky}$
        $\text{with flakiness score} f(t), \forall t \in T^P\}$
 4: $T^E \leftarrow \{t : f(t) \geq M, \forall (t, f(t)) \in T^F\}$
 5: $Results \leftarrow \emptyset$
 6: for all $t \in T^E$ do
 7:     $P \leftarrow \emptyset$, $F \leftarrow \emptyset$
 8:     for $i = 0$ to $E$ do
 9:         $(trace, passed) \leftarrow$ Run $t$ with instrumentation
10:         if $passed$ then
11:             $P \leftarrow P \cup trace$
12:         else
13:             $F \leftarrow F \cup trace$
14:         end if
15:     end for
16:     if $P \neq \emptyset \wedge F \neq \emptyset$ then
17:         $Results \leftarrow Results \cup \text{DIVERGENCE}(t, P, F)$
18:     end if
19: end for
20: return $Results$
```

```java
1  import java.util.Random;
2
3  final class RandomNumberGenerator {
4    private static final Random R = new Random();
5
6    /* Generate 0 or 1 with this one weird trick. */
7    public static int getRandomZeroOrOne() {
8      final int randomBtw0and99 = R.nextInt(100);
9      if (randomBtw0and99 < 50) {
10       return 0;
11     } else {
12       return 1;
13     }
14   }
15 }
```

Listing 3. Report generated by FD using DIVERGENCE algorithm. The common lines $7-9$ executed on both passing and failing runs are gray; line 10 is green since it was only in the passing execution; line 12 is red since it was only in the failing execution.

Listing 2 and `java.util.Random` in Listing 1 are not instrumented), because developers want to understand and debug their own code, and typically ignore the code in upstream dependencies. Then, FD executes the test t a total of E times and collects dynamic execution traces for both passing and failing test executions. We set $E = 50$.

Finally, common and divergence points are found using the DIVERGENCE algorithm as shown in Figure 2, and stored in a backend database with a link to an html report to be shown to developers later, as shown in Listing 3. Once enabled, FD executes without any action from developers, prepares reports, stores and caches them.

IV. Case Studies and Discussion

To evaluate the effectiveness of FD, we performed several case studies. All of these case studies involve FD reports on flaky tests, similar to the one shown in Listing 3. FD uses an internal dynamic execution tracing technology at Google, with certain limitations:

- It only works for tests that take shorter to finish executing than a specific time limit, to prevent execution traces from getting too large.
- It limits the total size of collected execution traces, to prevent using too many resources.
- It only supports C++ and Java.

Due to these limitations, combined with the limit M used in DIVERGENCE (from Section III-C), we have FD reports only for a subset of all flaky tests across Google.

TABLE I
STATISTICS ON DEVELOPER INVESTIGATION OF FD REPORTS ON FLAKY TESTS.

	Developer 1	Developer 2
# FD reports analyzed	83	83
In C++	39	39
In Java	44	44
# Useful-Exact (UE)	43	36
# Useful-Relevant (UR)	25	32
# Not-Useful (NU)	15	15

A. Case Study 1: Usefulness of FD Reports

In this study, we found a total of 83 historical issues that had been opened about flaky tests (as shown in Figure 1), have since been resolved with one or more code changes tagged as fixes for the issue, and for which FD produces reports. 39 of the tests were in C++, 44 were in Java. These issues never received any feedback from FD, i.e. all had been manually investigated and fixed by developers already. We then ran FD on these flaky tests at the version they had been identified to be flaky. Then we asked two developers, who are not in any of the teams to which these 83 issues belong, to independently inspect the FD report only (they did not have access to the issue reports or how each issue has been resolved by the original teams), and predict the root cause of flakiness based on that report in three categories:

1) **Useful-Exact** (UE): Flakiness is due to the exact lines pointed to by the FD report and can be fixed by changing those lines.
2) **Useful-Relevant** (UR): Flakiness is relevant to the lines pointed to by the FD report, but should be fixed in another location in the code (e.g. the issue is due to an RPC timeout, FD points to the RPC call site, the fix is to increase the timeout defined as a constant in another file).
3) **Not-Useful** (NU): FD report is inconclusive, hard to understand, or not useful.

Developer's responses are summarized in Table I. Both developers marked identical FD reports as NU, agreed on the reports they found useful ($UE + UR$), but disagreed on the categorization of the potential fix on 7 FD reports. We investigated those reports and found that both developers are correct in their categorization, as there may be several ways to fix a flaky test. As an example, in Listing 4, either an order-preserving `Map` implementation can be used in `ItemStore`, or `testGetItems` can be changed to assert on equality of unordered `Collections`.

After we gathered the two developers' responses, we compared their predictions with the original fixes submitted by the developers on each issue report. The fixes for all of the 68 issues where our survey developers marked FD reports to be useful ($UE + UR$) were fixed according to at least one of their predictions, i.e. FD reports were useful predicting the fix in 81.93% of the cases.

Furthermore, at the end of the study, after looking at the actual fixes of the remaining 15 issues they marked as

```
/* System code to store items. */
final class ItemStore {
  final Map<String, String> items = new HashMap<>();

  // Some business logic code
  ...
  public void insertItem(String key, String value) {
    items.put(key, value);
  }

  public Map<String, String> getItems() {
    return items;
  }
}
------------------------------------------------------
/* Test ItemStore */
final class ItemStoreTest {

  @Test
  public void testGetItems() {
    Map<String, String> exp = {
      "key1": "item1",
      "key2": "item2"
    };
    ItemStore store = new ItemStore();
    for (Entry<String, String> e : exp.entrySet()) {
      store.insertItem(e.getKey(), e.getValue());
    }
    Collection<String> v = store.getValues();
    assertEquals(2, v.size());
    assertEquals("item1", v.iterator().next());
    assertEquals("item2", v.iterator().next().next()
    );
  }
}
```

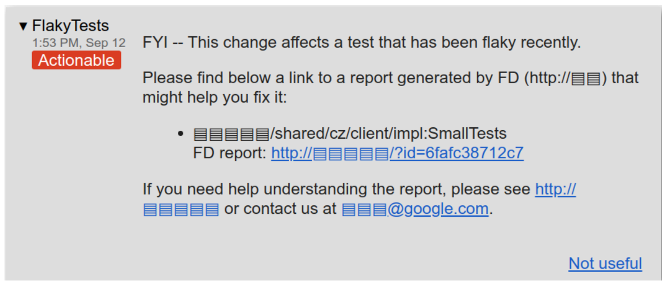
Listing 4. Flaky test `testGetItems()` can be fixed in two ways: (1) `ItemStore` can use an order-preserving `Map`; (2) the test can accept random traversal in its assertions.

NU, we determined that 4 of the reports were pointing to code locations that could have been relevant (UR) if they had more experience with the reports FD generates and the projects' codebase. The remaining 11 reports were for tests that had long execution times and were terminated by the test runner due to time limits, hence the locations pointed to by FD varied depending on the time of termination, i.e. they seemed random/unrelated in the FD reports, even though an experienced FD user could still understand that the generated reports might be related to terminations.

Finally, we asked the two developers for feedback on their experience with FD reports, and received the following responses:

> *"It takes some time to get used to the reports (colors, divergence etc.), but once you do, it is so easy to understand some of the root causes in subsequent reports."*

> *"You might be able to automatically classify some flakiness types and tell developers directly. This would make things much easier for them: instead of trying to understand how the tool works and what the colors are for, they can be told things such as 'you are likely using an unordered map' or 'your RPC call is timing out'."*


▾ FlakyTests
1:53 PM, Sep 12
Actionable

FYI -- This change affects a test that has been flaky recently.

Please find below a link to a report generated by FD (http://[redacted]) that might help you fix it:

- [redacted]/shared/cz/client/impl:SmallTests
  FD report: http://[redacted]/?id=6fafc38712c7

If you need help understanding the report, please see http://[redacted] or contact us at [redacted]@google.com.

Not useful

Fig. 3. Critique notification about existence of a flaky test displayed on code changes and a FD report url to help debug and fix it.

TABLE II
STATISTICS ON FD NOTIFICATIONS DISPLAYED IN CRITIQUE ON CODE CHANGES RELATED TO FLAKY TESTS.

# projects with FD enabled	428
# Critique notifications displayed	8182
# times a developer viewed a notification	501
# unique developers that viewed a notification	79
# times a developer clicked "*Not useful*"	2

"Some of these reports are very accurate, so accurate that you may be able to automatically generate fixes for them without involving developers, and that would be much better for everyone."

Based on these feedback, we conclude that, even though the reports are not too complicated, developers might have difficulty interpreting them when they see them for the first time. Certain root causes of flakiness can be identified based on the code fragments involved in the divergent lines, e.g. `for` loops on a `Map` may point to an assumption on the `Map` being order-preserving, and explicitly articulating it would be beneficial to developers. Developers prefer more automation, i.e. instead of helping debug flaky tests, it would be more beneficial if a tool automatically fixes them.

B. *Case Study 2: Critique Notifications*

In this study, we obtained a list of all projects for which FD has been enabled. For these projects, over 9 months, we showed notifications in Critique, shown in Figure 3, about existence of flaky tests and a url to debug them for every code change that modifies any code relevant to the flaky test or the flaky test itself. Table II summarizes metrics on the engagement of developers with these notifications.

During this study, FD has been enabled for 428 projects. We showed a notification for 8182 code changes to 79 unique developers. Developers viewed the notifications 501 times (6.12% of all notifications) and clicked on *Not useful* 2 times. Based on our discussions with developers, there were two common reasons for the low engagement rates:

- Notification was about a flaky test that is unrelated to their current changes.
- Notification required extra action, i.e. they had to click on the report and debug it. They preferred debugging and fixing flaky tests during specific periods, e.g. *fixit*s [35], when they dedicate specific days/weeks to work on maintenance activities in their projects.

These results agree with previous studies that reported debugging flaky tests is time consuming [7], [10], [12], therefore developers do not take action on them immediately, and prefer to debug them in boxed / dedicated time windows. We conclude that it is critical to integrate flaky test notifications into the appropriate development workflow journeys to obtain better engagement from developers.

TABLE III
STATISTICS ON FLAKY TEST RELATED ISSUES OPENED BY DEVELOPERS IN GOOGLE ISSUE TRACKER AND FD COMMENTS ON THOSE ISSUES.

# issues on flaky tests opened by developers	28250
# issues with a FD report	300
# issues still open at the end of the study	134
# issues fixed at the end of the study	166
# unique developers notified with a FD report	150

C. *Case Study 3: Google Issue Tracker Comments*

In this study, we obtained the full list of all Google Issue Tracker issues (shown in Figure 1) on flaky tests manually opened by developers during a 16-month window. Summarized in Table III, there were a total of 28250 such issues.

From this list, we generated FD reports for 300 supported tests, i.e. FD is enabled for the project of the test, the test is implemented in Java or C++, and the test is supported by the dynamic execution tracer. We then commented on each of the 300 issues with a link to the FD report. By the end of our study, 166 of the tests with a FD report were fixed by developers.

For the 166 fixed issues, we asked the 150 unique developers who were assigned to fix those issues to give us feedback on their experience regarding the integration of FD into Google Issue Tracker. 18 developers responded, and provided positive feedback summarized below.

"I liked it! Overall, this is pretty cool. [...] in this case, the test failure was pretty simple, [...] I hope to get these in more complex situations, they seem like they would really be helpful in the non-trivial cases."

"I liked the report in general. And, it was very pleasant to see someone chime in on [an issue] that I filed, with suggestions :) I'll try to file these [issues] for the rest of our [flaky tests] & see if we get generated reports for those. Overall, I think the way the suggestion comes (in form of the [issue] comment) is perfect. I liked it."

We did not receive any negative feedback from Google Issue Tracker users. We don't know the reason behind this, but we suspect that there may be responder bias where issues may get several automated comments (e.g. reminders to fix/close them if they are overdue), and only those developers that found the

TABLE IV
STATISTICS ON DEVELOPERS THAT PROVIDED FEEDBACK ABOUT USABILITY OF FD REPORTS.

# unique developers who were shown a notificaton in Critique in Case Study 2	79
# unique developers who were notified with a comment in their Google Issue Tracker issues in Case Study 3	150
# feedback reports we received	20
# unique developers that provided feedback	18

FD reports useful may have responded to our survey, and the rest of the users may have simply ignored it.

Based on the feedback, we conclude that, when integrated into Google Issue Tracker, where developers willingly open issues to debug and fix flaky tests themselves, notifications with FD reports can be useful to them by providing additional tooling support.

D. Case Study 4: Usability of FD Reports

In this study, we asked developers for feedback on their perception of the FD reports through a free-form text survey, summarized in Table IV. 79 unique developers have been shown a notificaton in Critique in Case Study 2, and 150 unique developers were notified with a comment in their Google Issue Tracker issues in Case Study 3. In total, 229 unique developers received a notification from FD and were asked to provide feedback through our survey. We received 20 feedback forms from 18 unique developers, covering different aspects of the usability of the reports. Below is a representative sample of these feedback.

Overall, several developers found the reports useful in debugging and identifying the root causes of flakiness.

> *"The [FD] report helped me to identify the problem much faster. Thank you!"*

> *"The report is awesome. It provided me with suspicious file and [. . .] suspicious line of code. It helped me locate the issue of the [. . .] failure."*

> *"[FD] was useful to have a starting point for debugging. It would have been harder to find the culprit without the report."*

> *"I really like the idea behind these reports and think it could help a lot of people."*

Furthermore, we received some negative feedback from several developers, specifically when the FD report is of type *Useful-Relevant* (from Section IV-A). They expected the FD reports to directly tell them where flakiness should be fixed, as opposed to where it manifests.

> *"The [report] pointed out the place where a system is checked to see if it's alive. The flakiness is in the system startup however [. . .]."*

> *"This is an integration test. The flaky failure is probably in another binary. So the report's info seems very wrong."*

We also received some negative feedback on the usability / comprehension of the FD reports from several developers.

> *"I have no idea what the tool is trying to tell me without reading the documentation."*

> *"[The report] needs superimposed word-bubbles [. . .] and tooltips – not colors."*

Finally, we received recommendations on improving FD reports, specifically by providing more automated insight into the root causes, similar to the feedback we received from the developers in Case Study 1.

> *"Could integrate with known testing code that is often used in flaky tests, such as WaitForCondition.loopUntilTrue(), with suggestions (e.g. increase deadline, reduce time to predicate evaluating true in tested code)."*

These feedback suggest that FD reports can be useful to help debug and fix flaky tests, can be improved and simplified for easier comprehension, and can help developers further by recommending fixes on a higher level than just pointing to code locations.

V. THREATS TO VALIDITY

Our studies are empirical, and carry the common threats of validity associated with such studies. Below we focus on specific ones.

Choice of projects: All case studies depend on the projects of the teams that enabled FD. The tool has been advertised broadly inside Google, but we had no control over which teams enabled it. However, overall, we assume that teams that put importance on fixing flaky tests may have enabled it, and this may have introduced bias in our results.

Programming languages: FD only supports C++ and Java. This might have introduced bias on the conclusions of our case studies if certain languages yield to less or more flaky tests, and if we could get FD reports for more languages.

Choice of parameters: In our case studies, we set the parameters M and E to specific values based on estimates and experience at Google. These specific values may not yield the same results outside Google, and they may yield different results if set to different values. Therefore, the values we chose may have introduced bias in our results.

Types of flaky tests: At Google, there are certain strategies to deal with specific types of flaky tests. For example, some test runners run tests in random order, so that test order related dependencies can be caught and fixed. Therefore, certain types of flaky tests have likely been fixed already and did not make it to any of our case studies, potentially limiting our conclusions to flaky tests outside that set.

Choice of flaky tests: FD reports are not generated for tests that are rarely flaky due to the parameter M we used in Section III-C to limit how many times we will run flaky tests to obtain at least one failing run. Rarely flaky tests might have different characteristics than the ones we obtained FD reports for.

Manual analysis by developers: Case Study 1 depends on the manual inspection of two independent developers to understand and classify the root causes of flakiness. Although these developers both have more than five years of experience each in programming, they may have made mistakes, both in identifying the root causes of flakiness, and classifying them.

Developer behavior: Case Study 3 uses issues manually created by developers on flaky tests. Our results might be biased since such issues are typically created by developers who want to fix flaky tests, regardless of whether they are easy or hard to fix.

Responder bias: In Case Studies 2, 3 and 4, we surveyed developers. First, we don't know the qualities of the responders. Second, the number of responses we received to our survey is small. Therefore, there may be responder bias in feedback responses and our conclusions from them may not generalize.

VI. Related Work

Flaky tests. Flaky tests and various problems they cause have been reported and discussed by both practitioners [4], [5], [8], [9], [12]–[16] and researchers [6], [7], [11], [17]–[26]. Fowler [6] reported that regression testing had recurring issues with non-deterministic tests. Memon and Cohen [19] outlined several reasons that cause GUI tests to be flaky. Lacoste [7] reported several unwanted side effects of flaky tests. Memon and Cohen [12] reported several difficulties created by flaky tests in large scale continuous integration testing.

Flaky test categorization. There have been several recent studies on categorizing flaky tests. Luo et al. [11] reported an extensive study of flaky tests on 51 open source projects, and classified root causes into ten categories. Palomba and Zaidman [41] also studied flaky tests on 18 projects and classified root causes into ten categories. Lam et al. [25] reported a study where they classified flaky tests to be order dependent or non-order dependent. Gao et al. [20] studied open source projects focusing on concurrency related bugs and flakiness, and classified their root causes into three categories. In this paper, our focus is not on categorizing flaky tests, but on helping developers debug and fix them.

Flaky test detection & mitigation. Several recent studies focus on automatically detecting specific types of flakiness. Zhang et al. [22] discussed test order dependency, a common reason for flakiness, studied real-world tests with dependency issues, and proposed several techniques to detect such tests. Muşlu et al. [21] observed that isolating unit-tests during execution can be helpful in detecting flakiness, but it can be computationally intensive. Bell and Kaiser [17] proposed tracking side-effects on shared memory objects and reversing these between test-runs to detect flaky tests. Farchi et al. [42] investigated concurrency bugs and proposed static analysis to detect them. Lu et al. [43] also reported a study on concurrency bugs discussing their patterns, manifestation and fixes. Gyori et al. [44] proposed PolDet, a technique to detect tests that leave a different environment state than when they started, so they can detect order dependent flaky tests. Gambi et al. [45] proposed PraDet, a technique that can detect flaky tests due to test order dependencies. Bell et al. [23] proposed that it is expensive to re-run tests to identify if they are flaky, and instead proposed DeFlaker, a technique to use additional code coverage during code changes to mark failing tests that are unrelated to new code changes as flaky. Lam et al. [25] proposed iDFlakies, an approach to automatically detect flaky tests by reordering tests, and produced a corpus of flaky tests for further research. The focus of our paper is not on detection, but on debugging and fixing flaky tests instead.

Flaky test fixing. There have been recent studies that focus on fixing specific types of flaky tests. Palomba and Zaidman [41] investigated flaky tests that are caused by the code under test, and proposed that fixing code smells in tests indirectly helps fixing 54% of flaky tests. Our proposal in this paper applies to both test code and system code related flakiness. Shi et al. [26] proposed a technique, iFixFlakies, to use *helpers*, code that sets the state, to automatically generate patches to fix order dependent flaky tests. Our proposal in this paper is not only geared towards order dependency but also other reasons of flakiness.

Fault localization. There is a large body of literature on fault localization, i.e. identifying the locations of faults in a system. For an extensive survey in the field, we refer readers to the recent study by Wong et al. [46]. Broadly, these techniques use the results of multiple tests in a test suite to identify potential code locations that are the likely causes of deterministically failing tests. The work in this paper partially builds on the Spectrum Based Fault Localization techniques [47]. However, instead of using failure results from different deterministic tests, we re-run a single test several times, to find the root cause of a non-deterministic test. The first version of the work in this paper adapted a technique similar to Tarantula by Jones et al. [48] to show developers a ranked list of possible locations of flakiness. However, during our initial studies, developers reported confusion on how the ranked code locations were related, and had difficulty interpreting the output. Based on this feedback, we proposed the new `DIVERGENCE` technique, as it is similar to how developers typically debug code.

VII. Conclusion

Regression testing is a critical part of software development. Existence of flaky tests in the regression test suite can severely undermine several software development activities. Therefore it is important to have developers fix them quickly.

Prior work has studied existing software systems, shown there are common categories of flaky tests, some of these flaky tests can be automatically detected and some specific types of flaky tests, such as order dependent flaky tests, can be automatically fixed.

We present `DIVERGENCE`, a new technique that can automatically identify locations of root causes of flaky tests. We implemented and deployed this technique in a tool, FD, across several products' flaky tests at Google, and performed case studies to assess its success and usefulness.

Our evaluation of the accuracy of FD on 83 fixed flaky tests shows that it can point to the location of relevant code involved in flakiness with 81.93% accuracy when compared to the actual fixes submitted by developers. In another study, we observed that developers are not motivated to fix flaky tests that are unrelated to their current code changes, and prefer to fix such tests during dedicated time windows for maintenance such as fixits. In another study, we added FD reports as comments on open issues, and observed that developers were positive on the integration of FD reports into their workflow. Finally, our assessment of the expectations and perceptions of developers from a tool like FD shows that several developers found the reports hard to understand, some expected the tool to actually fix the flakiness automatically, while other developers found it beneficial in debugging and fixing their flaky tests, and wanted the reports to go further by providing suggestions on the type of flakiness and for potential fixes.

VIII. Future Work

Based on feedback from several users, there are several important future research directions we identified. First is to increase the number of respondents to our surveys and further study their traits (e.g. junior vs. senior developers, C++ vs. Java developers), which may provide further insight. Another is to further simplify the reports shown to developers and to make the information easier to comprehend. Another is to automatically classify flaky tests using more metadata, e.g. involved code fragments, and convey this information to developers directly on the reports. Finally, automatically fixing flaky tests is an important area of research, as developers prefer fully automated solutions over their personal involvement.

Acknowledgment

We thank Vivek Parikh for encouraging and supporting this work at Google. We thank Xuchen Ma for his contributions to the first prototype of FD. We also thank internal Google reviewers as well as ICSME committee reviewers for valuable discussions and feedback on this paper.

References

[1] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 426–437.

[2] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at facebook and oanda," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2016, pp. 21–30.

[3] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[4] J. Micco. (2016, May) Flaky tests at google and how we mitigate them. [Online]. Available: https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html

[5] T. C. Projects. (2019) Flakiness dashboard howto. [Online]. Available: https://bit.ly/2lBHId5

[6] M. Fowler. (2011) Eradicating non-determinism in tests. [Online]. Available: https://bit.ly/2PFHI5B

[7] F. J. Lacoste, "Killing the gatekeeper: Introducing a continuous integration system," in *2009 agile conference*. IEEE, 2009, pp. 387–392.

[8] P. Sudarshan. (2012) No more flaky tests on the go team. [Online]. Available: https://thght.works/2ko7qBD

[9] G. T. Blog. Tott: Avoiding flakey tests. [Online]. Available: https://bit.ly/2m5yF4h

[10] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 1–23.

[11] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 643–653.

[12] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 233–242.

[13] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 235–245.

[14] H. Jiang, X. Li, Z. Yang, and J. Xuan, "What causes my test alarm? automatic cause analysis for test alarms in system and integration testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 712–723.

[15] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 483–493.

[16] 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*. IEEE Press, 2015, pp. 39–48.

[17] J. Bell and G. Kaiser, "Unit test virtualization with vmvm," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 550–561.

[18] T. Lavers and L. Peters, *Swing Extreme Testing*. Packt Publishing Ltd, 2008.

[19] A. M. Memon and M. B. Cohen, "Automated testing of gui applications: models, tools, and controlling flakiness," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 1479–1480.

[20] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang, "Making system user interactive tests repeatable: When and what should we control?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 55–65.

[21] K. Muşlu, B. Soran, and J. Wuttke, "Finding bugs by isolating unit tests," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 496–499.

[22] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in

Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014, pp. 385–396.

[23] 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.

[24] A. Labuschagne, L. Inozemtseva, and R. Holmes, “Measuring the cost of regression testing in practice: a study of java projects using continuous integration,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 821–830.

[25] 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.

[26] 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*. ACM, 2019, pp. 545–555.

[27] J. Micco, “Tools for continuous integration at google scale,” *Google Tech Talk, Google Inc*, 2012. [Online]. Available: https://www.youtube.com/watch?v=KH2_sB1A6lA

[28] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, “Trade-offs in continuous integration: assurance, security, and flexibility,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 197–207.

[29] A. Kumar, “Development at the speed and scale of google,” *QCon San Francisco*, 2010. [Online]. Available: https://bit.ly/2kup7PZ

[30] J. Micco, “Continuous integration at google scale,” 2013. [Online]. Available: https://bit.ly/2SYY4rR

[31] A. Reference. (2019) Flakytest annotation. [Online]. Available: https://bit.ly/2kcW4jI

[32] S. Reference. (2019) Repeat annotation. [Online]. Available: https://bit.ly/2kC5fKG

[33] J. G. Repository. (2017) Randomlyfails annotation. [Online]. Available: https://bit.ly/2kst2gb

[34] A. Maven. (2018) Rerun failing tests. [Online]. Available: https://bit.ly/2lIsbrN

[35] J. Micco, “The state of continuous integration testing @google,” 2017. [Online]. Available: https://bit.ly/3glb8nw

[36] C. Ziftci and J. Reardon, “Who broke the build?: Automatically identifying changes that induce test failures in continuous integration at google scale,” in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 2017, pp. 113–122.

[37] JUnit. (2019) Ignore annotation. [Online]. Available: https://bit.ly/2mb6nFH

[38] J. Micco and A. Memon, “Test flakiness @google - predicting and preempting flakes,” *Google Test Automation Conference*, 2016. [Online]. Available: https://www.youtube.com/watch?v=CrzpkF1-VsA

[39] Google. (2018) Google issue tracker. [Online]. Available: https://developers.google.com/issue-tracker

[40] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, “Modern code review: a case study at google,” in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 181–190.

[41] F. Palomba and A. Zaidman, “Does refactoring of test smells induce fixing flaky tests?” in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 1–12.

[42] E. Farchi, Y. Nir, and S. Ur, “Concurrent bug patterns and how to test them,” in *Proceedings International Parallel and Distributed Processing Symposium*. IEEE, 2003, p. 286.

[43] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics,” in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1. ACM, 2008, pp. 329–339.

[44] A. Gyori, A. Shi, F. Hariri, and D. Marinov, “Reliable testing: detecting state-polluting tests to prevent test dependency,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 223–233.

[45] 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.

[46] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[47] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, “A critical evaluation of spectrum-based fault localization techniques on a large-scale software system,” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 114–125.

[48] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. IEEE, 2002, pp. 467–477.