Root Causing Flaky Tests in a Large-Scale Industrial Setting

Wing Lam
University of Illinois at Urbana-Champaign
Urbana, IL, USA
winglam2@illinois.edu

Patrice Godefroid
Microsoft Corporation
Redmond, WA, USA
pg@microsoft.com

Suman Nath
Microsoft Corporation
Redmond, WA, USA
sumann@microsoft.com

Anirudh Santhiar
Microsoft Corporation
Redmond, WA, USA
ansanthi@microsoft.com

Suresh Thummalapenta
Microsoft Corporation
Redmond, WA, USA
suthumma@microsoft.com

ABSTRACT

In today's agile world, developers often rely on continuous integration pipelines to help build and validate their changes by executing tests in an efficient manner. One of the significant factors that hinder developers' productivity is *flaky tests*—tests that may pass and fail with the *same* version of code. Since flaky test failures are not deterministically reproducible, developers often have to spend hours only to discover that the occasional failures have nothing to do with their changes. However, ignoring failures of flaky tests can be dangerous, since those failures may represent real faults in the production code. Furthermore, identifying the root cause of flakiness is tedious and cumbersome, since they are often a consequence of unexpected and non-deterministic behavior due to various factors, such as concurrency and external dependencies.

As developers in a large-scale industrial setting, we first describe our experience with flaky tests by conducting a *study* on them. Our results show that although the number of distinct flaky tests may be low, the percentage of failing builds due to flaky tests can be substantial. To reduce the burden of flaky tests on developers, we describe our end-to-end *framework* that helps identify flaky tests and understand their root causes. Our framework instruments flaky tests and all relevant code to log various runtime properties, and then uses a preliminary *tool*, called RootFinder, to find differences in the logs of passing and failing runs. Using our framework, we collect and publicize a *dataset* of real-world, anonymized execution logs of flaky tests. By sharing the findings from our study, our framework and tool, and a dataset of logs, we hope to encourage more research on this important problem.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330570

KEYWORDS

flaky tests, debugging, regression testing

ACM Reference Format:
Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. 2019. Root Causing Flaky Tests in a Large-Scale Industrial Setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330570

1 INTRODUCTION

Both the industry and the open-source community have embraced the *Continuous Integration* (CI) model of software development and releases [26, 45]. In this model, every check-in is validated through an automated pipeline, perhaps running as a service in the cloud, that fetches source code from a version controlled repository, builds source code, and runs tests against the built code. These tests must all pass in order for the developer to integrate changes with the master branch. Thus, tests play a central role in ensuring that the changes do not introduce regressions.

To ensure that developers deliver new features at a high velocity, tests should run quickly and reliably without imposing undue load on underlying resources such as build machines. On the other hand, it is also critical that no regressions are introduced into the existing code. That is, in an ideal world, test failures would reliably signal issues with the developer's changes and every test failure would warrant investigation. Unfortunately, the reality of CI pipelines today is that tests may pass and fail with the *same* version of source code and the *same* configuration. These tests are commonly referred to as *flaky tests* [19, 34]. In a recent keynote [37], Micco observed that 1.5% of all test runs in Google's CI pipeline are flaky, and almost 16% of 4.2 million individual tests fail independently of changes in code or tests. Similarly, we find a non-negligible fraction of tests to be flaky; over a one-month period monitoring of five software projects, we observed that 4.6% of all individual test cases are flaky. Given their prevalence, there have even been calls to adopt the position that *every* test is potentially a flaky test [24].

The presence of flaky tests imposes a significant burden on developers using CI pipelines. In a survey conducted on 58 Microsoft developers, we find that they considered flaky tests to be the second most important reason, out of 10 reasons, for slowing down software deployments. A further detailed survey using 18 of the developers showed that they value debugging and fixing existing

flaky tests as the third most important course of action for Microsoft to take regarding flaky tests. These debugging and fixing efforts are often complicated by the fact that the test failures may only occur intermittently, and are sometimes reproducible only on the CI pipeline but not on local machines. When we re-run flaky tests locally 100 times, we find that 86% of them are only flaky in the CI pipeline. This result is not surprising since reproducing flaky behavior entails triggering a particular execution among many possible non-deterministic executions for a given flaky test. Non-determinism can arise from the non-availability of external I/O resources, such as network or disk, or from the order of thread and event scheduling. Prior work [34, 42, 51] uncovering these factors has examined in detail the extent to which these factors contribute towards flakiness, and developers often find it too difficult and expensive to identify the root cause of flakiness.

In most cases of flaky test failures, developers often resort to rerunning the tests after spending considerable effort debugging the failures [36, 37]. In some cases, developers may request that the flaky test should be rerun up to n times with the hope that at least one of the runs passes. For large repositories with multiple flaky tests that may fail independently, this process could add substantial time to the developer feedback loop, and can also result in scarcity of resources shared across teams. For example, Google ends up using ≈ 2-16% of its testing budget just to rerun flaky tests [37]. Ignoring flaky test failures can be dangerous since a rerun test that passes might hide a real bug in the production code. One study [43] found that when developers ignored flaky test failures during a build, the deployed build experienced many more crashes than builds that did not contain any flaky test failures. Another recent study [49] also found developers treating the signals from tests as unreliable, sometimes choosing to simply ignore test failures.

Debugging flaky test failures is a difficult and time-consuming activity, yet important to ensure high-quality production code. Therefore, there is a pressing need to develop automated scalable techniques to help developers debug flaky tests. This work takes the first step in this direction. More specifically, we study the prevalence of flaky tests in an industrial setting using production data obtained within Microsoft. We then describe an end-to-end framework that identifies the flaky tests during regular test executions and helps understand the root causes of the flakiness. This framework can be replicated for other companies and open-source projects.

Our experience with flaky tests includes dynamically instrumenting flaky test executions. Even though we focus here on unit test executions, we observe 335k methods calls, 5 threads, and 55418 objects on average per test run. Future efforts on flaky tests should hopefully be able to handle programs of this scale. We also describe detailed case-studies that illustrate some of the difficulties any automated technique will need to solve to debug flaky tests arising in production. Finally, we develop new tools and techniques to reduce the burden of flaky tests on developers, and present interesting challenges that can help guide future research in the area of flaky tests. Our experience with flaky tests provide three main lessons: (1) there is no correlation between the number of flaky tests and the number of builds that fail due to flaky tests, (2) excessive runtime overhead due to factors such as instrumentation can affect the reproducibility of flaky tests, and (3) finding differences in the runtime behavior of flaky tests can be an effective way to help developers root cause flakiness in tests. More details about the lessons we learned are presented in Section 5.

Overall, this paper consists of two parts. The first part contains a motivational study and a dataset of flaky tests. The second part contains a preliminary tool and framework to help identify and debug flaky tests. The paper makes the following contributions:

Study: A large-scale study of flaky tests in an industrial setting. Our quantitative results demonstrate that flaky tests are prevalent, and that although the number of distinct flaky tests is comparatively low, the number of validation runs that could fail due to flaky tests is high, emphasizing the problematic nature of flaky tests to developers. Along with the quantitative results, we also provide qualitative insights with in-depth examples that demonstrate the root causes of flaky tests.

Dataset: A collection of publicly available real-world, anonymized execution logs of flaky tests. We hope that the data will spur the development of new research techniques to root cause flaky tests and evaluate the effectiveness of tools that can help investigate flaky tests. These logs are publicly available online [8].

Tool: We develop and make publicly available a preliminary tool [8], called RootFinder, that analyzes the logs of passing and failing executions of the same test to suggest method calls that could be responsible for the flakiness.

Framework: An end-to-end framework developed within Microsoft that uses RootFinder to root cause flaky tests. This framework can be easily replicated for other companies and open-source projects.

2 OUR EXPERIENCE WITH FLAKY TESTS

Microsoft makes a CI pipeline available to its developers as a modern build and test service framework on the cloud, called CloudBuild. CloudBuild is an incremental and distributed system for building code and executing unit tests, similar to other engineering systems such as Bazel [2] and Buck [3].

When a developer sends a build request to CloudBuild with a change, CloudBuild constructs a dependency graph of all modules in the project and identifies the modules that are impacted by the given change. CloudBuild executes unit tests only in those impacted modules, and skips the remaining modules' unit tests, since none of their dependencies were changed. Note that, within a module, CloudBuild always executes all tests in the same order. Also, CloudBuild executes unit tests as soon as their dependencies are ready, rather than waiting for the entire build to finish. This is a major advantage that helps reduce the overall build time since tests are executed while building modules. This feature also helps with an effective utilization of resources [47]. Today, CloudBuild is used by ≈1200 projects inside Microsoft and executes ≈350 million unit tests per day across all projects. Therefore, CloudBuild is certainly an ideal system to conduct large-scale studies.

Table 1 provides statistics about the flaky tests collected across 30 days in five projects that use CloudBuild. Due to confidentiality reasons, we anonymized the names of the projects. The goal of this data is to show the prevalence of flaky tests in Microsoft. We collected this data using a feature of CloudBuild, where failing tests are automatically reran, and if the rerun passes, we identify the test as flaky. However, there is no guarantee that rerunning a failure

Table 1: Statistics showing the prevalence of flaky tests in five projects using CloudBuild.

Projects	# Tests	# Builds	# Test Executions	# Flaky Test Failures	# Distinct Flaky Tests	# Builds with Flaky Test Failures	% of Builds with Flaky Test Failures
ProjA	26,404	302	6,670,299	6,106	2,165	43	14%
ProjB	5,675	430	2,433,452	537	125	224	52%
ProjC	23,651	575	4,596,490	3,530	190	173	30%
ProjD	5,693	741	1,449,233	328	98	126	17%
ProjE	3,390	1,823	786,898	1,564	286	429	24%

from a flaky test will result in a pass. Therefore, the actual number of failures from flaky tests could be higher than the reported values.

In the table, Column 2 shows the number of distinct unit tests in each project. Column 3 shows the number of failing builds of each project. Column 4 shows the number of unit tests executed in all builds. Note that CloudBuild does not execute all unit tests in each build, rather it executes only those tests that are within the modules impacted by the change. Column 5 presents the number of unit test failures due to flaky tests, and Column 6 shows the number of distinct flaky tests that failed at least once. Finally, Columns 7 and 8 show the number and percentage, respectively, of builds that contained at least one flaky test failure. Note that each of these builds can have more than one flaky test failure.

Our results show that, although the number of distinct flaky tests is low, the percentage of builds that include flaky test failures is substantial. For example, **ProjB** has the most builds (≈52%) failed because of one or more flaky test failures, but the project's number of flaky tests is substantially lower than other projects (e.g., **ProjA**). Upon further investigation into ProjA's flaky tests, we find that its large number of flaky tests can be attributed to a single change causing a large number of tests to be flaky for a few builds, and once the flakiness was fixed, these tests no longer caused builds to fail due to flaky test failures. Overall, our results illustrate the prevalence of flaky tests on these five projects, and how often developers are burdened by flaky tests causing their builds to fail.

3 END-TO-END FRAMEWORK

We next present our framework to identify the root causes of flaky test failures. The framework consists of multiple steps.

First, CloudBuild identifies flaky tests by rerunning failing tests to see if the retry passes or not. If the retry passes, then the test is identified as flaky. CloudBuild then stores information about all of these flaky tests in a scalable storage.

Next, for each flaky test specified in this storage, we collect all dependencies (i.e., test binary, its dependent source binaries, and relevant test data) from CloudBuild, so that the test can be executed locally on any machine independent of CloudBuild.

Next, for each flaky test, we produce instrumented versions of all of its dependencies using an instrumentation framework, called Torch [27, 33]. The instrumentation helps us log various runtime properties of the test execution. Note that instrumented binaries retain the same functionalities as the original binaries, and therefore, tests can seamlessly run on Torch-instrumented binaries.

Using the instrumented version, we next run the test **100** times on a local machine in an attempt to produce logs for both passing and failing executions. The logs generated by Torch contain various runtime properties at different execution points. More details are described in Section 3.1. We run the test 100 times as doing so represents a good trade off between gaining logs for both passing and failing executions, and the time spent on test runs. The test is run offline at a later time instead of on CloudBuild machines, since running the test 100 times under instrumentation is expensive, thereby increasing the build times for the users and causing resource contention. Nevertheless, our future work plan includes to not only collect the passing/failing logs of a flaky test during minimal workload times on the CI machines, but also to analyze the logs on the machines. Doing so would not only notify developers of a flaky test but also the possible root causes of the flakiness as part of their normal continuous integration workflow.

Since tests may exercise many methods during their execution, these logs can be very large and highlighting the differences among the passing and failing executions is beneficial for the majority of them. Therefore, we also present a tool, RootFinder, that automatically analyzes these logs to highlight the differences to provide developers insight on why the test is flaky.

3.1 Torch Instrumentation

Torch [27, 33] is an extensible instrumentation framework for .Net binaries. It takes a .Net binary and a set of target APIs, and instruments each target API call in the binary. The exact nature of instrumentation depends on what Torch *instrumentation plugin* is used. For instance, the profiling plugin instruments the binary to track latencies of APIs executed on critical paths. Torch comes with plugins for profiling, logging, fault injection, concurrency testing, thread schedule fuzzing, etc. One can extend these plugins or write new plugins to suit one's instrumentation goals.

During instrumentation, Torch replaces each API call with an automatically generated proxy call, as shown in Figure 1. Note that Torch does not instrument the implementation of a target API; only the call to the API is instrumented. The proxy generated by Torch calls the original API; in addition, as shown in Figure 1(c), it calls three Torch callbacks—(1) `OnStart`, called immediately before calling the original API, (2) `OnEnd`, called immediately after the original API returns, and (3) `OnException`, called when the original API throws an exception. `OnStart` returns a context that is passed to `OnEnd` and `OnException`; the context is used to stitch together information tracked by callbacks for the same API call.

For identifying the root causes of flaky tests, we use Torch's logging plugin to passively track and log various runtime properties. We find that some APIs will behave differently in passing and failing executions, and analyzing the differences will provide us insights on the root causes of flakiness. Since such APIs are not known beforehand, we opt for logging information for *all* API calls. Specifically, we log the following properties for all API calls.

(1) Call information, including signature of the API, and its caller API (the API calling the instrumented API). We also track location

```
1  WebClient client = new WebClient();
2  string data = client.DownloadString (url);
```

(a) Original code

```
1  WebClient client = new WebClient();
2  TorchInfo ti = Torch.GetInstrumentationInfo();
3  string data = Torch_DownloadString(ti,client,url);
```

(b) Instrumented code

```
1  public static string Torch_DownloadString(TorchInfo torchInfo,
       WebClient instance, string url) {
2    string returnValue = null;
3    var context = Torch.OnStart(torchInfo, instance, url);
4    try {
5      returnValue = instance.DownloadString(url);
6    } catch (Exception exception) {
7      Torch.OnException(exception, context);
8      throw; // rethrow original exception
9    } finally {
10     Torch.OnEnd(returnValue, context);
11   }
12   return returnValue;
13 }
```

(c) Proxy method

Figure 1: Torch instrumentation

of the API call in the binary and/or source code. Signatures and locations let us uniquely identify each API call.

(2) Timestamp at each call. Timestamps at `OnEnd` and `OnStart` give the latency of the API call.

(3) Return value at `OnEnd` and exception at `OnException`, if any.

(4) A unique Id of the receiver object of the API. This helps identifying APIs operating on the same object and thus uncovering potential concurrency issues. (5) Ids of the process and thread executing the API.

(6) Id of the parent thread, i.e., the thread that spawned the thread executing the API. The information is important to understand dependencies of different threads and their activities [33].

It is important for the instrumentation to have a small runtime overhead. Excessive overhead can change runtime behavior by e.g., removing existing flakiness, introducing new flakiness, or timing out. The overhead comes from two different sources. First, computing some of the runtime properties on the fly can be expensive. For example, finding signatures of an API and its object type through reflection, or finding the parent API through stack trace can be expensive. We avoid this cost by computing these static properties during instrumentation and passing them to the `OnStart` callback as static parameters (as a `TorchInfo` object in Line 3 in Figure 1(b)). Second, since we collect runtime information for all APIs, the size of logs they generate can be prohibitively large. To avoid the overhead, we compress the logs in memory and asynchronously write them to disk.[1]

3.2 Log Analysis Tool

We develop a simple tool called RootFinder to parse Torch logs of passing and failing executions to identify potential root causes of certain types of flaky tests. At a high level, RootFinder takes as input a method name that is likely to be the cause of the flakiness and two directories containing Torch logs of passing and failing test runs, and determines if the method exhibits anomalous runtime behavior in the failing runs. Examples of input method names that may be of interest include methods that return non-deterministic values such as `System.Random.Next` (returns a non-negative random integer) or `System.DateTime.Now` (returns a DateTime object representing the current date and time). By default, our framework will use RootFinder with a predefined set of nondeterministic method calls. Developers can also add or remove method calls as they wish.

RootFinder works in two steps. In the first step, it processes each log file independently and evaluates a set of predicates at each line of the log file. The predicates, similar to the ones used in statistical debugging [32], determine if the behavior of the callee method in a log line is "interesting" (several example predicates will be given shortly). The outputs of the predicates are written to a *predicate file*. Each line in the predicate file contains the following information about a predicate: (1) The logical *epoch* when the predicate is evaluated, (2) name of the predicate, (3) value of the predicate at the current epoch. We currently consider predicates that are *local* to specific code locations, and therefore use logical epochs that can identify partial orders of predicates evaluated at the same code location. More specifically, the epoch is given by a concatenation of the unique code location of the method call,[2] current thread id, and a monotonically increasing sequence number that is incremented every time the method is called at the current location. For instance, in Figure 3, the method `Random.Next()` at unique location `9` is called multiple times (e.g., perhaps the line is in a loop or is called by multiple threads)—once in log line `2` and again in log line `5`. Assuming that they both are executed in the same thread with id `10`, the first call has the epoch `9:10:1`, the second call has the epoch `9:10:2`, and so on. Partial orders of the epochs can be derived from their ids along with the threads' parent-child relationship, which Torch dynamically tracks and logs.

Predicates: A predicate evaluates the state of the method call at the current epoch. RootFinder currently implements the following boolean predicates:

- *Relative:* The predicate is true if the return value of the current epoch is the same as that of any previous epochs. This predicate is useful to identify if a non-deterministic method is returning the same value in successive calls.
- *Absolute:* The predicate is true if the return value of the current epoch matches a given value. This predicate is useful to check if a method returns an error value (e.g., `null` or an error code).
- *Exception:* is true if the method throws an exception.
- *Order:* The predicate is true if an ordering of method calls matches a given list of methods and optionally, whether a specified amount of time occurred between the methods. This predicate is useful to identify thread interleavings.
- *Slow:* This predicate is true if the method call takes more than a specified time. (The threshold can be determined based on domain knowledge of the called method, or by analyzing latencies of passing test runs.)

[1] We also experimented with more lightweight Event Tracing for Windows (ETW) logging [1]; however, at a high logging event rate, ETW may skip logging randomly chosen events. We observed a high loss rate, and hence did not use ETW for logging.

[2] *Unique code location* uniquely identifies the location of a method call in the code. An example is the name of the program source/binary file plus the line number/binary offset of the method call within the file. For simplicity we use `Source#` as the unique location in the rest of the paper.

- *Fast:* This predicate is true if the method call takes less than a specified time.

After the predicate files are generated, RootFinder compares all predicate files (from passing and failing runs) to identify ones that are true/false in all passing executions, but are the contrary in all failing executions. Intuitively, these predicates are strongly correlated to test failures and hence are useful to understand the underlying root cause of failures. Specifically, RootFinder labels each predicate in the predicate files with one of the following categories:

(1) Inconsistent-in-passing: Such a predicate either appears in only a subset of all passing test runs or appears in all passing runs but with more than one value. The log line corresponding to such a predicate is likely irrelevant as to why a test is flaky. This is because whether the predicate was true or false did not affect the outcome of the test runs (i.e., they always passed).

(2) Inconsistent-in-failing: Such a predicate either appears in only a subset of all failing test runs or appears in all failing runs but with more than one value. As in the case of the previous category, this predicate is also likely irrelevant as to why a test is flaky.

(3) Consistent-and-matching: Such a predicate appears in all passing and failing runs and with the same value. This predicate is also likely irrelevant as to why the test is flaky as it did not affect the final outcome of the test runs.

(4) Consistent-but-different: Such a predicate either (I) appears only in passing or only in failing runs, or (II) is true in all passing runs but false in all failing runs (or vice versa). (I) indicates that executions of a passing and a failing run diverge before the epoch where the predicate was evaluated (which is why the predicate appears in one set and not the other), while (II) indicates how a method consistently behaves differently in the passing and failing runs. This predicate is *highly likely* to explain why a test is flaky because it precisely shows how passing and failing test runs differ.

By default, the predicates outputted by RootFinder are sorted so that the ones that are most likely to explain why a test is flaky are shown first (i.e., Consistent-but-different predicates). Once the categories are sorted, RootFinder then sorts the predicates within each category so that the predicates with the lowest log line numbers are outputted before the ones with higher numbers. In our case studies with RootFinder as described in Section 4.2, we find that sorting predicates as stated enables RootFinder to output useful ones in the least amount of time.

The predicates outputted by RootFinder can aid the debugging efforts of nine out of ten categories of flaky tests mentioned in a survey [34]. More specifically, for categories such as Network, Time, IO, Randomness, Floating Point Operations, Test Order Dependency, and Unordered Collections, RootFinder can directly compare the return value of failing and passing Torch logs to identify predicates that are highly likely to explain why the test is flaky. For the Async Wait and Concurrency categories, our framework currently relies on Torch's ability to first fuzz delays, and then for RootFinder to identify latency-related predicates, such as Fast and Slow, to help developers root cause those categories. For the remaining category, Resource Leak, we plan to extend the set of predicates to include memory leak detection tools [5, 9] to help developers understand tests of this category with new predicates.

```
1  class TestAlertTest {
2    void TestUnhandledItemsWithFilters() {
3      TestAlert ta1 = CreateTestAlert();
4      TestAlert ta2 = CreateTestAlert();
5      ...
6      Assert.AreNotEquals(ta1.TestID, ta2.TestID);
7    }
8    TestAlert CreateTestAlert() {
9      int id = new Random().Next();
10     ...
11     return new TestAlert(TestID = id, ...);
12   }
13 }
```

Figure 2: Test method from a Microsoft product's test suite.

Line #	Source #	Seq #	Caller name	Callee name	Timestamp	Return Value
1	3	1	TestAlertTest.TUIWF()	TestAlertTest.CreateTestAlert()	1	null
2	9	1	TestAlertTest.CreateTestAlert()	Random.Next()	2	14
3	11	1	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	2	null
4	4	1	TestAlertTest. TUIWF()	TestAlertTest.CreateTestAlert()	3	null
5	9	2	TestAlertTest.CreateTestAlert()	Random.Next()	3	16
6	11	2	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	3	null
...	...	...	...	...	...	...
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	20	null
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	17	null
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	28	null

Figure 3: Torch logs for passing test executions of the test in Figure 2. `TUIWF` is `TestUnhandledItemsWithFilters`.

Line #	Source #	Seq #	Caller name	Callee name	Timestamp	Return Value
1	3	1	TestAlertTest.TUIWF()	TestAlertTest.CreateTestAlert()	1	null
2	9	1	TestAlertTest.CreateTestAlert()	Random.Next()	2	21
3	11	1	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	2	null
4	4	1	TestAlertTest. TUIWF()	TestAlertTest.CreateTestAlert()	2	null
5	9	2	TestAlertTest.CreateTestAlert()	Random.Next()	2	21
6	11	2	TestAlertTest.CreateTestAlert()	TestAlert.TestAlert(int)	2	null
...	...	...	...	...	...	...
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	18	null
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	24	null
8	6	1	TestAlertTest. TUIWF()	Assert.AreEquals(...)	16	null

Figure 4: Torch logs for failing test executions of the test in Figure 2. `TUIWF` is `TestUnhandledItemsWithFilters`.

An Example. Figure 2 shows the simplified version of a test of a confidential product in Microsoft (the corresponding code under test implementing `TestAlert` is omitted for brevity). `TestUnhandledItemsWithFilters` is flaky since `new Random().Next()` may actually return the same value if two consecutive calls are invoked close together. This is because if a `Random` object is instantiated without a seed, it takes the current system time as the default seed; therefore, two `Random` objects instantiated without a seed and within a short window of time may be initialized with the same seed, causing their `Next()` calls to return the same sequences of random numbers.

Figures 3 and 4 show a fragment of Torch logs from passing test runs and from failing test runs (respectively) of the test in Figure 2. As shown in Lines 2 and 5 of Figure 4, the `StartTimes` of the `Random.Next()` calls are the same in all failing logs, therefore the return value for both calls to `Random.Next()` is the same value (e.g., `21`, `17`, and `5`). In the passing logs such as the ones depicted in Figure 3, we can see that on Lines 2 and 5, `StartTimes` are different and consequently, the return values of `Random.Next()` are also different. The assertion on Line 8 of Figure 2 passes if the return values of `Random.Next()` are different and fails otherwise.

RootFinder can narrow down the above root cause when it is invoked for the method `Random.Next()`. In step 1 of its processing, it

Table 2: Characteristics of the specific flaky test examples in Section 4.2 and of all 44 flaky tests in our dataset.

	Duration / test	% of failed executions/test	# of method calls/test	# of unique method calls/test	# of threads / test	# of objects / test
Specific examples in Section 4.2						
Time	1s	29%	4.7k	463	3	1,151
Randomness	4s	91%	0.8k	182	1	249
Async Wait	2s	1%	0.2k	90	4	62
Concurrency	10s	1%	18k	882	8	8,200
Resource Leak	4s	1%	0.6k	218	6	246
All 44 flaky tests						
Median	5s	6%	2.5k	248	3	637
Average	45s	28%	335k	335	5	55,418

converts the passing logs in Figure 3 into predicate files containing the predicate `(9:2, Relative, False)`. This predicate means that the method `Random.Next()` in `Source#` 9 and `Seq#` 2 returns a value that is different from the return value of the immediate previous call of the same method at the same `Source#`. Similarly, it converts the failing logs in Figure 4 into predicate files containing the predicate `(9:2, Relative, True)`. In step 2, RootFinder compares the predicates across runs and identifies the predicate as *Consistent-but-different*. This predicate quickly points to a root cause (or a symptom that is strongly correlated to the root cause) of the flakiness, as well as its code location (encoded in the epoch of the predicate).

4 CASE STUDIES

We next present the results of applying our framework on large projects that use CloudBuild as their CI pipeline. To ensure that our results are not biased due to a single project, we collected all distinct flaky tests recorded during a day in our production environment. Among these flaky tests, we identified the tests that are compatible with the Torch instrumentation framework. More specifically, CloudBuild supports unit tests written in both managed (such as C#) and unmanaged (such as C++) code [4]. Also, CloudBuild supports tests written for various test frameworks such as MsTest [6], NUnit [7], and XUnit [10]. Our current implementation of the instrumentation framework supports only unsigned (binaries that do not include digital signatures) and managed code, and is also tailored for those tests that run using the MsTest framework.

Overall, we collected **315** flaky tests that matched the criteria described above. Our collected flaky tests belong to different projects that provide Microsoft services for both internal and external customers, and also fall into different categories such as database, networking, security, and core services. Among these flaky tests, we were only able to reproduce flakiness i.e., produce logs for both passing and failing executions in 100 runs for **44** tests. Among the remaining tests, 97 of them have all of their runs pass. For these tests, we also tried a fuzzing technique that introduces delays using Torch, however we were still unable to reproduce the flakiness. It is important to note that the focus of our framework is for identifying and debugging flaky tests. Our findings here that only 44 out of 315 flaky tests are reproducible suggests that improvements to reproducing flakiness can also be highly impactful.

For the remaining 174 tests, we find that all 100 runs failed. During our inspection, we find that for some of the cases, the tests failed with a different error signature than the one that resulted in the flaky failures. These failures can be broadly classified into three categories; (1) the Torch instrumentation resulted in a new failure, (2) there are some issue in the instrumentation part of our framework that needs to be fixed, or (3) there are differences in CloudBuild and the test machine where we ran the tests. Our future work plan is to investigate these issues and also try alternatives such as running the instrumented tests directly on CloudBuild.

4.1 Study Dataset

We use a dataset consisting of 44 flaky tests. They belong to 22 software projects from 18 Microsoft internal/external products and services. For each test, the dataset contains 100 execution traces, some of which are from failed executions. Each trace file consists of a sequence of records containing various runtime information about an executed method as described in Section 3.1.

Table 2 shows some characteristics about the tests and traces. The characteristics show the overall complexities of the tests. The average run duration of the tests is nontrivial (45s), even though they are all unit tests and most of the I/O calls are mocked with fast proxy calls. Each test runs a large number of methods (335k total methods/test and 335 unique methods/test), mostly because a tested component often depends on many underlying components, each invoking many methods. 80% of the tests use more than one thread and on average, each test runs on 5 threads and operates on 55418 objects. Many of these elements can introduce nondeterminism that can make a test flaky. Moreover, the tests produce massive runtime logs, which can be extremely challenging to analyze.

Each runtime log in our dataset contains a wealth of information. For example, it contains all methods executed by the test, their latencies, return values, parent-child relationships of threads, activities of threads, etc. We believe that the dataset will be useful to the research community, not only to conduct research on various aspects of flaky tests and their root causes, but also for a general understanding of runtime behavior of tests in a production system. We have, therefore, made an anonymized version of the dataset available to the public [8]. In the anonymized dataset, sensitive strings (such as method names containing Microsoft product names) are replaced with hash values. The hashes are deterministic and hence can be correlated within and across trace files.

4.2 Case Studies of Finding Root Causes

As explained in Section 3.2, our framework in theory can address nine out of ten categories of flaky tests. In this section, we provide

```
1  [TestMethod]
2  public void TestReplicaService() {
3    ...
4    byte[] response = Service.SendAndGetResponse(payload);
5    byte[] replicaResponse = ServiceReplica.SendAndGetResponse(payload);
6    Assert.AreEqual(response, replicaResponse);
7  }
8  public class Service : NetworkService {
9    public byte[] SendAndGetResponse(Request req) {
10     ...
11     DateTime currentTime = DateTime.UtcNow;
12     Message message = new Message(req, currentTime);
13     return base.SendAndGet(message.Serialize());
14   }
15   ...
16 }
```

Figure 5: A test that is flaky due to getting the system time.

in-depth examples of flaky tests and how RootFinder assisted developers with debugging these particular flaky tests. The remainder of this section presents four examples of flaky tests that our framework can find root causes for and one example that our framework cannot. All examples are anonymized and simplified as needed.

4.2.1 Time. We find some tests to be flaky due to improper use of APIs dealing with time. These flaky tests rely on the system time, which introduces non-deterministic failures, e.g., a test may fail when time zones change.

Figure 5 shows a simplified version of a test case, which ensures that a service and its replica return the same response to a particular message. A developer may observe that the assertion on Line 6 occasionally fails. The failures are because calls to `Service` and `ServiceReplica`'s `SendAndGetResponse` may or may not use the same timestamps. If the invocations of `SendAndGetResponse` on Lines 4 and 5 happen within a short window of time, the timestamps produced by `DateTime.UtcNow` on Line 11 can be the same due to the limited granularity of the system timer. The granularity is seconds by default. If the timestamps are the same, then the test passes; otherwise, it fails. We find many flaky tests at Microsoft exhibiting similar behavior. This example fails 29% of the time in our experiments, but our experiments also find other tests exhibiting a similar root cause to fail up to 88% of the time.

A useful predicate for this example should indicate that the timestamp of `SendAndGetResponse` when invoked by Line 5 is always the same as the timestamp when invoked by Line 4 in the passing logs, but they are always different in the failing logs. When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, 11 seconds to run and outputted 1163 predicates total. The useful predicate was ranked at 81. In practice, when developers used RootFinder on this example, they were able to input suspicious method names to quickly find the useful predicate in a matter of minutes.

4.2.2 Randomness. Tests may pass or fail if their results depend on random numbers. More specifically, tests that use a random number generator without accounting for all the possible values that it may generate can be flaky.

One example of such a flaky test we find is shown in Figure 2. As explained in Section 3.2, the test in Figure 2 is flaky since

```
1  [TestMethod]
2  public void DelayedTaskStaticBasicTest() {
3    int delay = 1000; int i = 0;
4    Scheduler.ScheduleTask(
5      DateTime.UtcNow.AddMilliseconds(delay),
6      new LoggedTask(
7        "TestDelayedTaskFrameworkTask", () => { i = 1; },
8        new Dictionary<string, string> { { "test", "value" } }));
9    Thread.Sleep(500);
10   Assert.IsTrue(i == 0);
11   Thread.Sleep(delay);
12   Assert.IsTrue(i == 1);
13 }
```

Figure 6: A test that is flaky due to waiting for asynchronous calls.

`Random.Next()` actually uses the system time as the seed to generate a random number if no seed is provided by the user. Therefore, when the timing delay between the `CreateTestAlert()` calls and consequently the calls to `new Random().Next()` is too small for the system to record that it's different, then the test will fail. In our experiments, we find that the `TestUnhandledItemsWithFilters` test fails 91 out of 100 times. The fix for the flakiness of this test should be for the `CreateTestAlert` helper test method to not use random numbers for `testID`. Instead `CreateTestAlert` should take an `int` parameter and allow tests to pass in values for `testID` so that `TestAlerts` that should and shouldn't have the same `TestID` can be decided by the method calling `CreateTestAlert`. A useful predicate for this example is described in Section 3.2. When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, 2 seconds to run and outputted 408 predicates total. The useful predicate was ranked first.

4.2.3 Async Wait. Tests are flaky due to the Async Wait issues when the test execution makes an asynchronous call and does not properly wait for the result of the call to become available before using it. Depending on whether the asynchronous call was able to finish execution or not, such flaky test may pass or fail.

Figure 6 shows an example of an Async Wait flaky test. `DelayedTaskStaticBasicTest` schedules a task to run at a pre-defined time (the current time + 1000 milliseconds) on Line 4. In our experiments, we find that the `DelayedTaskStaticBasicTest` test fails 99 out of 100 times. The test can be flaky due to two main reasons.

(1) When the asynchronous task on Line 4 finishes executing before Line 10, then the test will fail. In passing executions of this test the value of `i` has not changed to `1`, but in the failing executions, delays before the assertion on Line 10 can actually be greater than the time it takes to execute the asynchronous task on Line 4. In such cases, the value of `i` when Line 10 executes is already `1` and the assertion will fail.

(2) When the assertion on Line 12 finishes executing before the asynchronous task on Line 4 finishes executing, then the test will fail. In passing executions of this test, the value of `i` is changed to `1` before Line 12 executes, but in the failing executions, the asynchronous task on Line 4 runs so slow that the delays on Lines 9 and 11 are not enough to prevent Line 12 from executing before the asynchronous task finishes. In all 6 test failures that we encounter, the flaky test failed due to this second reason.

A useful predicate for this example should indicate that the task on Line 4 always took longer in the failing runs. When we apply RootFinder to this example without any domain knowledge from

developers, it took, on average, one second to run and outputted 868 predicates total. The useful predicate was ranked at 17.

4.2.4 Concurrency. A flaky test's root cause is Concurrency when the test can pass or fail due to different threads interacting in a non-deterministic manner (e.g., data races, deadlocks).

Figure 7 shows an example of a test that is flaky due to concurrency issues. The `TestDirtyResource` method tests that a manager (created in Line 4) of a cluster properly recycles used resources. Once the manager is created, it is setup by adding a machine to it, that is in use, or "dirty" (Lines 5–6), along with more setup code (omitted for brevity). The test then repeatedly creates and sends requests to the cluster to execute jobs (Lines 9–12), and makes sure that the response obtained in line 12 is correct (Lines 13–16). Finally, the resources used by the cluster to process the particular request are released (Line 17), and the newly freed resource heartbeats its status to the manager (Line 18), and the manager processes it (Line 19). We observed in our experiments that the assertion on Line 13 failed occasionally.

Upon investigating the failure, we find that the creation of the resource manager (Line 4) also starts a background task that periodically marks resources as unavailable in case T milliseconds (ms) has elapsed since the last heartbeat was processed. Test failure occurs when this task runs T ms after another thread has processed Line 19, since the background task would mark the cluster resource as unavailable, which would then cause the subsequent resource allocation request made using `QueryAllocate` on Line 12 to return `null`. The null value will then cause the assertion on Line 13 to fail. This example demonstrates a subtle flakiness condition that only manifests on a particular thread interleaving, and moreover, only if the interleaving follows very precise timing constraints.

A useful predicate for this example should indicate that the background task from Line 4 always ran after Line 19 and it always runs T ms or longer after Line 19. When we apply RootFinder to this example without any domain knowledge from developers, it took, on average, 126 seconds to run and outputted 127187 predicates total. The useful predicate was ranked at 3231. In practice, when developers used RootFinder on this example, they provided some domain knowledge and RootFinder was eventually able to help them debug the root cause of this flaky test. The difficultly for RootFinder here is largely because this flaky test's root cause is not only thread interleaving, but also the timing of the interleavings. Our future work plan includes improving RootFinder to consider combinations of predicates (e.g., Order and Slow) to better root cause flaky tests such as this example.

4.2.5 Resource Leak. A flaky test's root cause is Resource Leak when the test passes or fails because the application does not properly acquire or release its resources, e.g., locks on files.

Figure 8 shows an example of a Resource Leak flaky test. `ResourceAllocation` tests whether the necessary resources are properly allocated for an application. The application internally uses a third-party database to store some information. The `TestCleanup` method is executed after every test case to delete the database, so that it may be re-initialized before the next test case. Although Line 12 tries to close the connection to the database, the third-party library requires the garbage collector to run prior to this step in order to release the file handle to the database file. If the garbage collector does not run first, then the file resource is held, causing the subsequent attempts to delete the file on Line 12 to throw an exception. Since our framework relies on Torch to capture information relevant to the root cause of the flaky test from the user's code, our framework is currently unable to assist developers for Resource Leak flaky tests such as the one in this example.

```
1  [TestMethod]
2  public async Task TestDirtyResource() {
3    ...
4    using (var emptyPoolManager =  CreatePoolManager(...)) {
5      Resource pm =   ResourceUtils.CreatePhysicalMachine(...);
6      await emptyPoolManager.AddOrUpdateResourceAsync(pm, HeartbeatStatus.
           InUse);
7      ...
8      for (int i = 0; i < 5; i++) {
9        var sessionId = Guid.NewGuid().ToString();
10       var request = new ResourceAllocateRequest(pm);
11       await emptyPoolManager.PreAllocateResourcesAsync(request.Yield(), pm.
             Specification, TenantId, sessionId, sessionPriority);
12       var response = (await QueryAllocate(emptyPoolManager, request.Yield()
             , TenantId, sessionId)).FirstOrDefault();
13       Assert.IsNotNull(response);
14       Assert.AreEqual(request.Id, response.RequestId);
15       Assert.IsNotNull(response.ResourceId);
16       Assert.AreEqual(pm.ResourceId,response.ResourceId);
17       await emptyPoolManager.ReleaseResourcesAsync(response.ResourceId);
18       await emptyPoolManager.HeartbeatResourceAsync(pm, HeartbeatStatus.
             Ready);
19       await emptyPoolManager.ProcessResourcesHeartbeats(CancellationToken.
             None);
20     }
21   }
22 }
```

Figure 7: A test that is flaky due to concurrency.

```
1  DatabaseProvider ssp;
2  String currentDirectory = ...;
3  [TestMethod]
4  public void ResourceAllocation() {
5    ssp = new DatabaseProvider(currentDirectory);
6    ...
7  }
8  [TestCleanup]
9  public void TestCleanup() {
10   ClearConnections();
11   if (File.Exists(dbPath)) {
12     File.Delete(dbPath);
13   }
14   ...
15 }
```

Figure 8: A test that is flaky due to resource leaks.

4.3 Applying RootFinder on to Case Studies

When we apply RootFinder onto the examples described in Section 4.2, we find that the root causes are concisely summarized as predicates in the output for four of the five examples, especially when developers provided domain knowledge to RootFinder. These predicates can assist developers in identifying what values are the same or not in the passing and failing executions, and provide them the code location that produced these values. When we present the findings of RootFinder to developers or use it ourselves, they/we are able to more quickly reproduce the failures of the flaky test.

For the remaining example, Resource Leak, our future work plan includes possibly extending the end-to-end framework, Torch instrumentation framework, and RootFinder to provide valuable, concise information to developers in a timely manner. For example,

one possible extension we plan to make is to extend the end-to-end framework to include memory leak detection tools [5, 9]. This extension will allow us to provide developers information for the Resource Leak example in Section 4.2.5. Integrating such an extension may impose significant performance, resource and time cost to our current framework though. Therefore, more research and experimentation may be required to determine the best solutions to address all of the possible root causes of flaky tests.

5 LESSONS LEARNED

This section summarizes the main lessons learned so far from our experience with flaky tests in our specific industrial setting.

(1) Reproducing flakiness locally is challenging. In our experiments, we tried 315 tests that showed flaky behavior in CloudBuild. However, when run on a machine outside CloudBuild, we did not observe flakiness in a vast majority of them (86%), even after running each test 100 times.

(2) Runtime overhead from instrumentation can affect the reproducibility of flaky tests. We used several runtime optimizations in Torch, including pre-computing static properties and compressing logs. Yet, when we randomly sampled 59 flaky tests to run 100 times locally on a machine with and without Torch instrumentation, we observed that 2 tests are flaky only with Torch instrumentation, while 3 tests are flaky only without Torch instrumentation.

(3) The number of flaky tests a project contains does not correlate to the frequency in which builds fail. E.g., as shown in Table 1, ProjB has a low number of flaky tests but a high number of failed builds, whereas the situation is the opposite for ProjA.

(4) Finding differences in the runtime behavior of flaky tests can be an effective way to help developers root cause these tests. In fact, our preliminary tool along with our lightweight instrumentation can already help developers in nine out of ten causes of flaky tests.

We emphasize that these results are preliminary and that the main purpose of this paper is instead to encourage more work in this specific problem area. With this goal in mind, we identify several open research challenges in Section 8.

6 THREATS TO VALIDITY

Internal Validity: Our threats to internal validity are concerned with our study procedure's validity. RootFinder and our framework can contain faults that impact our results and lessons learned. We attempt to mitigate this threat by having thorough code reviews and testing of RootFinder and our framework. Furthermore, we rely on various other tools in our framework, such as Torch. These tools could have faults as well and such faults could have also affected our results. To mitigate this threat, the logs produced by Torch and the root causes produced by RootFinder are manually analyzed and confirmed by at least two of the authors.

External Validity Our threats to external validity are concerned with all threats unrelated to our study procedure. Our lessons learned may not apply to projects other than the ones in our study. We attempt to mitigate this threat by including a diverse range of projects in our study. Our projects service for both internal and external customers of Microsoft, and also fall into different categories such as database, networking, security, and core services. Flaky tests are, by definition, tests that nondeterministically pass or fail with the same version of code. Due to the nondeterministic nature of these tests, the flaky tests from our dataset that we are able and unable to produce Torch logs for may not be true if the experiment was to be repeated. To mitigate this threat, we ran each test a nontrivial amount of times.

7 RELATED WORK

Prevalence and costs of flaky tests. Luo et al. [34] noted in first extensive study of flaky tests that specific numbers on the occurrence of flaky tests were hard to obtain. Since then, the problem has received more attention, with recent results indicating that flaky test failures are relatively common. We find that on average, 27.4% builds exhibit flaky tests while Labuschagne et al. [30] found that 12.8% of builds in their data set of 935 builds from 61 projects using Travis CI, a cloud-based CI tool, failed because of flaky tests. Palomba and Zaidman [42] found 8829 flaky tests out of 19532 JUnit tests from 18 projects - 45% of all tests they analyzed were flaky. Moreover, these tests showed both passing and failing behavior using only up to 10 reruns. In our experience, flaky behavior is rarer (around 4.6% of all tests flaked), as well as harder to reproduce. Of 311 tests in our dataset that failed on the cloud, we could not reproduce the failure locally in 97 cases, even with 100 reruns. Rahman and Rigby [43] showed, using Firefox as a case study, that ignored flaky tests can lead to crashes in production. In spite of this, developers do ignore flaky tests; Thorve et al. [49] examine 77 commits pertaining to flaky tests from 29 Android projects, finding that 13% of the commits simply skipped or removed flaky tests. Lam et al. [31] found 388 flaky tests in 683 projects, and publicized the list flaky tests they found online. Their dataset includes the names and partial classification of the flaky tests, but does not contain detailed execution logs, such as the ones provided in our dataset.

Causes of flaky tests. Zhang et al. [51] note that test suites can suffer from *test order dependency* where test outcomes can be affected by the order in which the tests are run. In our case, CloudBuild always runs tests of a test suite in the same order. Luo et al. [34] investigate 201 commits from 51 open-source projects, finding that the primary causes for flakiness are (i) async-wait, (ii) concurrency, including atomicity violation, data races and deadlocks, and (iii) test order dependency. Thorve et al. [49] find additional root causes for flaky tests in Android. These studies worked backwards from the commits fixing flaky tests to ascertain the root cause; in contrast, our RootFinder aims to diagnose flakiness before the flaky tests are fixed. Gao et al. [20] observe that it is difficult to reliably replay and reproduce outcomes from tests where a user interacts with a system, owing to test dependencies on system platform, library versions and tool configurations. Although we focus on unit tests, our experience has been similar.

Detecting and fixing flaky tests. A standard strategy in CI pipelines to deal with flaky tests is to rerun them—if a failing test passes on re-running, it no longer gates a build [34, 37]. However, running tests can be expensive especially when these tests are gating a build. DeFlaker [13] monitors the code coverage of recent changes and marks as flaky any newly failing test that did not execute any of the changes. On the other hand, RootFinder helps developers root cause flaky tests by finding the differences in the logs of passing and failing executions. Since Deflaker helps developers know which tests

may be flaky and RootFinder helps developers root cause flaky tests, we recommend developers to first use Deflaker to find flaky tests and then use RootFinder to root cause them once they are found. Herzig and Nagappan [25] propose an association rule mining based technique to tag a system/integration test with many steps that fails as a false alarm, based on whether each step failed or passed. Among approaches to detect root causes of flaky tests, detecting dependent tests has received the most research attention [31, 38, 51]. Palomba and Zaidman [42] established a relationship between test smells and flaky tests. Refactoring the tests fixes the flakiness in all the tests containing the code smell in their study. However, it is not clear how to generalize their approach to other root-causes for flaky tests. Shi et al. [46] detected nondeterministic flaky tests by detecting tests that assumes a deterministic implementation with a nondeterministic specification. In comparison to our work, we do not require specifications to detect flaky tests.

Concurrency is a major root-cause for flaky tests [34, 42, 49], therefore detecting and fixing concurrency issues could partially address the problem. There exist automated techniques to detect data-races [18, 39], deadlocks [21, 40, 44] and atomicity violations [17]. However, to integrate these techniques within modern CI pipelines would require overcoming significant challenges of meeting performance, resource and time constraints, providing multi-language support, and controlling rates of false positives and false negatives [14, 15, 28, 35]. Given the ubiquity of flaky tests in modern continuous integration environments, Harman and O'Hearn [24] advocate assuming all tests as flaky (with some probability).

Like RootFinder, several other systems use differences in runtime invariants in passing and failing tests to identify likely causes of failures [16, 22, 23, 32, 48]. Fault localization techniques and tools, such as Tarantula [29], Ochiai [12], Op2 [41], Barinel [11], and DStar [50], analyze different passing and failing tests in order to localize likely faults in programs. In contrast, RootFinder focuses only on non-deterministic tests that sometimes pass and sometimes fail, it depends on collecting a large volume of runtime logs (to not miss rare flakiness and to check for unplanned invariants), and it optimizes the process of log collection (to not affect flakiness and reproduce it even under instrumentation).

8 OPEN RESEARCH CHALLENGES

The preliminary results reported in this paper are just scratching the surface of identifying the root causes of flaky tests. By sharing our dataset of flaky tests, we hope to encourage more research in this area. Here are open questions that are left to be explored.

How to evaluate results. A major challenge is to determine the "*ground truth*" of why a test is flaky. Ultimately, this requires developers to look at each flaky test, examine the suggested candidate root causes, and then decide which root cause(s) is/are the most likely. For a large number of flaky tests, this evaluation is *expensive*. Moreover, results may vary depending on the developer's expertise and familiarity with the code being tested and with the tests being performed. How to prevent *biased evaluations* due to diversity in developer expertise is another non-trivial challenge.

What information to log. Earlier in the paper, we showed one way to log execution traces. But there are many other options. Should all method calls and returns be logged? In all the software components or only in some? If some, which ones and why? Should function input and output values be logged as well? Should intra-procedural execution information (e.g., which code instructions, blocks or branches were executed) also be recorded for a fine-grained analysis?

Logging versus analysis tradeoffs. The more data is being logged, the more computationally expensive it is to store and process all the data. The analysis itself becomes more complicated: non-essential differences may creep in if too much information is recorded, which makes it harder to identify meaningful differences. On the other hand, recording too little information may omit key events explaining the source of the test flakiness. How to strike the right balance between logging and analysis effectiveness is another key challenge in this space.

Fixed versus variable logging. Another dimension to the problem is whether the level of detail used for logging should be the same for all applications and tests, or whether it should be adjusted, automatically over time in an "iterative-refinement process", or using user-feedback. If the logging level can be adjusted, should it start *lazily*, at a high-level and be refined until meaningful differences are detected? Or, should logging proceed bottom-up, *eagerly* logging many events, then identifying irrelevant details (data noise) and eliminating those until meaningful differences are found?

Logging versus reproducibility. The more data one logs, the more intrusive the runtime instrumentation can be. As discussed earlier in this paper, reproducing flakiness is hard, and can become even harder when significant runtime slowdowns are introduced by expensive logging activities.

9 CONCLUSION

Flaky tests are a prevalent issue plaguing large software development projects. Simply ignoring flaky tests from a regression suite is a dangerous practice since it might mask severe software defects that will hurt users later on. To help software developers and testers, we clearly need better tools and processes for dealing with flaky tests. This paper presented two main efforts currently under way at Microsoft to address flaky tests. The first part consists of a motivational study and a dataset of flaky tests. The second part consists of a framework and preliminary tool to help debug flaky tests. More specifically, we described how centralized software building and testing can help detect and manage flaky tests. We also discussed preliminary work on new tools for root-causing flaky tests, in order to determine in a cost-effective manner what causes a test to be flaky and how to eliminate that root cause. The purpose of this paper is also to share with the research community a large data set of flaky tests and the tools we are currently building to root-cause these tests. We hope that this paper, dataset and tools will encourage more research on this important problem.

ACKNOWLEDGMENTS

Most of the work of Wing Lam was done while visiting Microsoft. We thank Ankush Das for his contributions to an early version of this work. We also thank August Shi, Owolabi Legunsen, Tao Xie, and Darko Marinov for their discussions about flaky tests. We also acknowledge support for research on flaky tests and test quality from Huawei and Microsoft.

REFERENCES

[1] 2011. Event Tracing for Windows (ETW) simplified. https://bit.ly/2NgEBzl.
[2] 2019. Bazel. https://bazel.build.
[3] 2019. Buck. https://buckbuild.com.
[4] 2019. Managed vs. Unmanaged development. https://bit.ly/2Or4C3p.
[5] 2019. MemorySanitizer. https://clang.llvm.org/docs/MemorySanitizer.html.
[6] 2019. MsTest framework. https://bit.ly/2NeZmLO.
[7] 2019. NUnit framework. https://nunit.org.
[8] 2019. RootFinder. https://sites.google.com/view/root-causing-flaky-tests/home.
[9] 2019. Valgrind. http://valgrind.org.
[10] 2019. xUnit framework. https://xunit.github.io.
[11] R. Abreu, P. Zoeteweij, and A. Gemund. 2009. Spectrum-based multiple fault localization. In *ASE*. Auckland, NZ, 88–99.
[12] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82 (2009), 1780–1792.
[13] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *ICSE*. Gothenburg, Sweden, 433–444.
[14] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53 (2010), 66–75.
[15] M. Christakis and C. Bird. 2016. What developers want and need from program analysis: An empirical study. In *ASE*. Singapore, 332–343.
[16] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69 (2007), 35–45.
[17] C. Flanagan and S. Freund. 2008. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming* 71 (2008), 89–109.
[18] C. Flanagan and S. Freund. 2009. FastTrack: Efficient and precise dynamic race detection. In *PLDI*. Dublin, Ireland, 121–133.
[19] M. Fowler. 2011. Eradicating non-determinism in tests. https://bit.ly/2PFHI5B.
[20] Z. Gao, Y. Liang, M. Cohen, A. Memon, and Z. Wang. 2015. Making system user interactive tests repeatable: When and what should we control?. In *ICSE*. Florence, Italy, 55–65.
[21] P. Godefroid. 1997. Model checking for programming languages using VeriSoft. In *POPL*. Paris, France, 174–186.
[22] A. Groce and W. Visser. 2003. What went wrong: Explaining counterexamples. In *International SPIN Workshop on Model Checking of Software*. Springer, 121–136.
[23] J. Ha, J. Yi, P. Dinges, J. Manson, C. Sadowski, and N. Meng. 2013. System to uncover root cause of non-deterministic (flaky) tests. In *Google Patent*. https://patents.google.com/patent/US9311220
[24] M. Harman and P. O'Hearn. 2018. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *SCAM, keynote*. https://bit.ly/2KzBnKQ
[25] K. Herzig and N. Nagappan. 2015. Empirically detecting false test alarms using association rules. In *ICSE*. Florence, Italy, 39–48.
[26] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE*. Singapore, 426–437.
[27] Y. Jiang, L. Sivalingam, S. Nath, and R. Govindan. 2016. WebPerf: Evaluating what-if scenarios for cloud-hosted web applications. In *ACM SIGCOMM*.
[28] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *ICSE*. San Francisco, CA, USA, 672–681.
[29] J. Jones and M. Harrold. 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE*. Long Beach, CA, USA, 273–282.
[30] A. Labuschagne, L. Inozemtseva, and R. Holmes. 2017. Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *ESEC/FSE*. Paderborn, Germany, 821–830.
[31] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*. Xi'an, China, 312–322.
[32] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. 2005. Scalable statistical bug isolation. *ACM SIGPLAN Notices* 40 (2005).
[33] L. Luo, S. Nath, L. Sivalingam, M. Musuvathi, and L. Ceze. 2018. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *USENIX ATC*.
[34] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. 2014. An empirical analysis of flaky tests. In *FSE*. Hong Kong, 643–653.
[35] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco. 2017. Taming Google-scale continuous testing. In *ICSE-SEIP*. Buenos Aires, Argentina, 233–242.
[36] J. Micco. 2016. Flaky tests at Google and how we mitigate them. https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html
[37] J. Micco. 2017. The state of continuous integration testing at Google. In *ICST, Keynote*. https://bit.ly/2OohAip
[38] K. Muşlu, B. Soran, and J. Wuttke. 2011. Finding bugs by isolating unit tests. In *ESEC/FSE*. Szeged, Hungary, 496–499.
[39] M. Naik, A. Aiken, and J. Whaley. 2006. Effective static race detection for Java. In *PLDI*. Ottawa, Canada, 308–319.
[40] M. Naik, C. Park, K. Sen, and D. Gay. 2009. Effective static deadlock detection. In *ICSE*. Vancouver, BC, Canada, 386–396.
[41] L. Naish, H. Lee, and K. Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20 (2011), 11:1–11:32.
[42] F. Palomba and A. Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSME*. Shanghai, China, 1–12.
[43] M. Rahman and P. Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In *ESEC/FSE*. Lake Buena Vista, FL, USA, 857–862.
[44] A. Santhiar and A. Kanade. 20167. Static deadlock setection for asynchronous C# programs. In *PLDI*. Barcelona, Spain, 292–305.
[45] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. 2016. Continuous deployment at Facebook and OANDA. In *ICSE-C*. Austin, TX, USA, 21–30.
[46] A. Shi, A. Gyori, O. Legunsen, and D. Marinov. 2016. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST*. Chicago, IL USA, 80–90.
[47] A. Shi, S. Thummalapenta, S. K Lahiri, N. Bjorner, and J. Czerwonka. 2017. Optimizing test placement for module-level regression testing. In *ICSE*. Buenos Aires, Argentina, 689–699.
[48] W. Sumner, T. Bao, and X. Zhang. 2011. Selecting peers for execution comparison. In *ISSTA*. Toronto, Canada, 309–319.
[49] S. Thorve, C. Sreshtha, and N. Meng. 2018. An empirical study of flaky tests in Android apps. In *ICSME, NIER Track*. Madrid, Spain, 534–538.
[50] E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63 (2014), 290–308.
[51] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. Ernst, and D. Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*. San Jose, CA, USA, 385–396.