An Empirical Analysis of Flaky Tests

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, Darko Marinov
Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{qluo2, hariri2, eloussi2, marinov}@illinois.edu

ABSTRACT

Regression testing is a crucial part of software development. It checks that software changes do not break existing functionality. An important assumption of regression testing is that test outcomes are deterministic: an unmodified test is expected to either always pass or always fail for the same code under test. Unfortunately, in practice, some tests—often called *flaky tests*—have non-deterministic outcomes. Such tests undermine the regression testing as they make it difficult to rely on test results.

We present the first extensive study of flaky tests. We study in detail a total of 201 commits that likely fix flaky tests in 51 open-source projects. We classify the most common root causes of flaky tests, identify approaches that could manifest flaky behavior, and describe common strategies that developers use to fix flaky tests. We believe that our insights and implications can help guide future research on the important topic of (avoiding) flaky tests.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Measurement, Reliability

Keywords: Empirical study, flaky tests, non-determinism

1. INTRODUCTION

Regression testing is a crucial part of software development. Developers use regression test suites to check that software changes do not break existing functionality. The result of running a regression test suite is a set of test outcomes for the tests in the suite. The outcomes are important for developers to take actions. If all the tests pass, developers typically do not inspect the test runs further. If any test fails, developers reason about the cause of failure to understand whether the recent changes introduced a fault in the code under test (CUT) or whether the test code itself needs to be changed [9]. The key assumption behind this process is that a test failure indicates that the recent changes introduced a problem in the CUT or the test code.

FSE'14 , November 16–22, 2014, Hong Kong, China

Unfortunately, test outcomes are not reliable for tests that can intermittently pass or fail even for the same code version. Following practitioners [12, 13, 20, 32, 34], we call such tests *flaky*[1]: their outcome is non-deterministic with respect to a given software version. Flaky tests create several problems during regression testing. First, test failures caused by flaky tests can be hard to reproduce due to their non-determinism. Second, flaky tests waste time when they fail even unaffected by the recent changes: the developer can spend substantial time debugging only to find out that the failure is not due to the recent changes but due to a flaky test [20]. Third, flaky tests may also *hide* real bugs: if a flaky test fails frequently, developers tend to ignore its failures and, thus, could miss real bugs.

Flaky tests are not only problematic but also relatively common in large codebases. Many practitioners and researchers have pointed out that flaky tests can be a big and frequent problem in general [7,12,13,21,26,29,32,34,37], but the only specific numbers we could obtain[2] are that the TAP system at Google had 1.6M test failures on average each day in the past 15 months, and 73K out of 1.6M (4.56%) test failures were caused by flaky tests.

The current approaches to combat flaky tests are rather unsatisfactory. The most common approach is to run a flaky test multiple times, and if it passes in any run, declare it passing, even if it fails in several other runs. For example, at Google, a failing test is rerun 10 times against the same code version on which it previously failed, and if it passes in any of those 10 reruns, it is labeled as a flaky test [15,27]. Several open-source testing frameworks also have annotations (e.g., Android has *@FlakyTest* [2], Jenkins has *@RandomFail* [17], and Spring has *@Repeat* [31]) to label flaky tests that require a few reruns upon failure.

Another approach would be to remove flaky tests from the test suite, or to mentally ignore their results most of the time (in the limit, ignoring the failure every time is equivalent to removing the test). In JUnit, the *@Ignore* annotation is used to exclude a test from the test suite to be run. However, developers are reluctant to use this approach, because flaky tests may still provide some coverage and could help find regression bugs. Although the current approaches used to deal with flaky tests may alleviate their impact, they are more "workarounds" rather than solutions. They do not address the root causes of flaky tests and can potentially waste a lot of machine resources (with test reruns) or reduce the effectiveness of the test suite (with flaky test exclusion).

[1]"Flaky" (sometimes spelled "flakey") means "unreliable".
[2]Personal communication with John Micco.

Findings about flaky test causes	Implications
F.1 The top three categories of flaky tests are Async Wait, Concurrency, and Test Order Dependency.	**I.1** Techniques for detecting and fixing flaky tests should focus on these three categories.
F.2 Most flaky tests (78%) are flaky the first time they are written.	**I.2** Techniques that extensively check tests when they are first added can detect most flaky tests.
Findings about flaky test manifestation	**Implications**
F.3 Almost all flaky tests (96%) are independent of the platform (i.e., could fail on different operating systems or hardware) even if they depend on the environment (e.g., the content of the file system).	**I.3** Techniques for manifesting flaky tests can check platform dependence lower in priority than checking environment dependence (e.g. event ordering or time), especially when resources are limited.
F.4 About third of Async Wait flaky tests (34%) use a simple method call with time delays to enforce orderings.	**I.4** Many Async Wait flaky tests can be simply manifested by changing time delays of order-enforcing methods.
F.5 Most Async Wait flaky tests (85%) do not wait for external resources and involve only one ordering.	**I.5** Most Async Wait flaky tests can be detected by adding one time delay in a certain part of the code without the need of controlling the external environment.
F.6 Almost all Concurrency flaky tests contain only two threads or their failures can be simplified to only two threads, and 97% of their failures are due to concurrent accesses only on memory objects.	**I.6** Existing techniques of increasing context switch probability, such as [10], could in principle manifest most Concurrency flaky tests.
F.7 Many Test Order Dependency flaky tests (47%) are caused by dependency on external resources.	**I.7** Not all Test Order Dependency flaky tests can be detected by recording and comparing internal memory object states. Many tests require modeling external environment or explicit reruns with different orders [37].
Findings about flaky test fixes	**Implications**
F.8 Many Async Wait flaky tests (54%) are fixed using `waitFor`. which often completely removes the flakiness rather than just reducing its chance.	**I.8 For developers:** Explicitly express the dependencies between chunks of code by inserting `waitFor` to synchronize the code. **For researchers:** Comparing the order of events between correct runs and failing runs, techniques could automatically insert order-enforcing methods such as `waitFor` to fix the code.
F.9 Various Concurrency flaky tests are fixed in different ways: 31% are fixed by adding locks, 25% are fixed by making code deterministic, and 9% are fixed by changing conditions. Our results are consistent with a study on concurrency bugs [24].	**I.9** There is no one common strategy that can be used to fix all Concurrency flaky tests. Developers need to carefully investigate the root causes of flakiness to fix such tests.
F.10 Most Test Order Dependency flaky tests (74%) are fixed by cleaning the shared state between test runs.	**I.10 For developers:** Identify the shared state and maintain it clean before and after test runs. **For researchers:** Automated techniques can help by recording the program state before the test starts execution and comparing it with the state after the test finishes. Automatically generating code in `setUp/tearDown` methods to restore shared program state, such as static fields [7], could fix many Test Order Dependency flaky tests.
F.11 Fixing flaky tests in other categories varies from case to case.	**I.11** There is no silver bullet for fixing arbitrary types of flaky tests. The general principle is to carefully use API methods with non-deterministic output or external dependency (e.g., time or network).
F.12 Some fixes to flaky tests (24%) modify the CUT, and most of these cases (94%) fix a bug in the CUT.	**I.12** Flaky tests should not simply be removed or disabled because they can help uncover bugs in the CUT.

Table 1: Summary of findings and implications

Despite the pervasiveness of flaky tests in practice, they have not drawn much attention from the research community. The few recent efforts focus on only one category of flaky tests due to test-order dependency [7, 29, 37]. In this paper, we present the first extensive study of flaky tests. We analyze in detail 201 commits that likely fix flaky tests from 51 open-source projects from the Apache Software Foundation. For each flaky test, we inspect the commit log message, the corresponding bug report (if any), and the corresponding patch to determine the root cause of the non-deterministic outcome and the way it was fixed. We also examine how the flakiness was introduced in the test suite and how it could be manifested.

We focus our study on the following questions that we believe could provide insights for practitioners and researchers:

(1) What are the common causes of flakiness? By studying the root causes of flaky tests, we reveal the most prominent categories of flaky tests for developers and researchers to focus on. Our study of how flakiness is introduced also suggests the best stage to identify flaky tests.

(2) How to manifest flaky test failures? By studying the possible ways to manifest flaky test failures, we suggest how automated techniques could detect unknown flaky tests.

(3) What are the common fixing strategies for flaky tests? By studying how developers fix flaky tests in practice, we provide insights for both developers about some principled ways for avoiding certain kinds of flaky tests and for researchers about the potential techniques that could automatically fix flaky tests.

Table 1 shows the summary of our findings and implications. The remaining sections discuss these in more detail.

2. METHODOLOGY

Our goal is to provide actionable information about avoiding, detecting, and fixing flaky tests. To that end, we focus on identifying and analyzing version-control *commits that likely fix flaky tests*. One can view each flaky test as a bug in the test code whereas it can produce a non-deterministic rather than deterministic outcome. Most empirical studies of bugs start from bug reports [8, 14, 22, 24, 28]. However, we start from commit logs, because we are mostly interested in flaky tests that are *fixed*. Starting from commits gives us a larger dataset than we would get starting from bug-report databases [5]. First, some fixes are made without ever being reported in bug-report databases. Second, some reports from bug-report databases are open and not fixed. In brief, every fixed flaky test is reflected in the version-control system but may not be reflected in its bug-report database.

To identify commits that likely fix flaky tests, we choose to search through the central SVN repository of the Apache Software Foundation [3]. This repository hosts a diverse set of over 150 top-level projects, written in various programming languages, having varying sizes, and being actively de-

	"intermit"	"flak"	Total	Total w. Bug Reports	HBase	ActiveMQ	Hadoop	Derby	Other Projects
All commits	859	270	1,129	615	134	86	90	118	701
Commits about flaky tests	708	147	855	545	132	83	83	102	455
LDFFT commits	399	87	486	298	72	68	56	49	241
Inspected commits	167	34	201	124	23	20	29	12	117
Async Wait	62	12	74	43	10	11	7	3	42
Concurrency	26	6	32	19	2	3	3	1	23
Test Order Dependency	14	5	19	16	3	0	10	2	4
Resource Leak	9	2	11	8	2	2	0	1	6
Network	10	0	10	6	1	1	2	0	6
Time	5	0	5	2	0	1	1	0	3
IO	4	0	4	3	0	0	1	1	2
Randomness	2	2	4	4	1	0	3	0	0
Floating Point Operations	2	1	3	2	0	0	1	1	1
Unordered Collections	1	0	1	1	0	0	0	0	1
Hard to classify	34	6	40	21	4	2	2	3	29

Table 2: Summary of commit info and flaky test categories

veloped for varying amount of time (from months to years) by a large open-source community.

We first extract the complete commit history of all projects from the Apache Software Foundation. To identify commit messages that may indicate a fix of a flaky test, we search for the keywords "intermit" and "flak". One could search for more keywords, but these two already find enough commits for several months of inspection. More precisely, the search yielded 1,129 commit messages. Table 2 shows the distribution of these between the two keywords. After collecting these commit messages, our study proceeds in two phases.

Filtering Phase. The first phase in studying these 1,129 commits is to identify those that are likely about fixing flaky tests. We manually inspect each commit, and if needed, the bug report(s) associated with this commit. To increase confidence in our inspection, two of the paper authors separately inspect each commit and then merge their results. The upper part of Table 2 shows the summary of this initial labeling phase. (We discuss the lower part of Table 2 in Section 3.)

Our goal is to determine for each commit whether: (1) it is likely about (reporting or fixing) a flaky test (labeled 'Commits about flaky tests' in Table 2), and (2) it attempts to fix a distinct flaky test (labeled 'LDFFT commits' in Table 2). We find 855 commits that are likely about flaky tests; the other 274 commits match "intermit" or "flak" but are either about the CUT not about the test code or just incidental matches (e.g., a username that contains "flak"). Of these 855 commits, 486 are likely distinct fixed flaky tests (*LDFFT commits*); the other 369 commits either report that certain tests are flaky but do not provide fixes for those tests, or are duplicates (e.g., multiple attempts to fix the same flaky test or multiple copies of commits from one development branch to another). For duplicates, we keep only the *latest* commit about the distinct flaky test, because we want to study the most recent change related to the flaky test (e.g., the most effective fix is usually the latest fix).

Comparing the results across the two keywords, we find that "intermit" is used more often than "flak" to indicate a flaky test. The numbers of commits for "intermit" are larger than the corresponding numbers for "flak" both in absolute terms (399 vs. 87) and in relative terms (46% vs. 32%).

Comparing the column for all commits and the column for the commits that have a bug report, we can see that a large fraction of fixes for flaky tests have no bug report. For example, 188 LDFFT commits have no bug reports, while 298 such commits have bug reports. Hence, we could have missed a large number of commits related to flaky tests if our methodology relied solely on bug reports.

The key result of this phase is a set of 486 LDFFT commits. Table 3 shows the number of these LDFFT commits across various projects. At least 51 projects out of the 153 projects in Apache likely have at least one flaky test. For each project, we tabulate the programming language(s) that it uses, the number of LDFFT commits, the number of commits that have at least one associated bug report, and the total number of lines of code in the project (computed by the "cloc" script). We can see that flaky tests occur in a diverse set of projects, using various languages, ranging over various sizes, and implementing code for various domains.

Analysis Phase. We study in more depth a subset of the 486 LDFFT commits, selected as follows. First we sort the projects according to their number of LDFFT commits and split them into two groups: small (less than 6 LDFFT commits) and large (greater than or equal to 6 LDFFT commits). Table 3 shows a line separating these two groups. Then we select to inspect all the commits from the small group and sample one third of the commits from each project from the large group. We are thus covering all the projects from the Apache Software Foundation where we identified some LDFFT commits, and our results are not overly biased toward the projects with the largest number of LDFFT commits. This sampling gives us a total of 201 out of 486 LDFFT commits to inspect.

For each of these 201 LDFFT commits, we first have one of the authors examine the commit in detail. Our goal is to answer the following set of questions: (1) Is the commit indeed fixing a flaky test? (2) What is the root cause of the flakiness for the test? (3) How can the flakiness be manifested? (4) How is the test fixed? The answers are then inspected and confirmed by another author. The following sections discuss these answers.

3. CAUSES OF FLAKINESS

We first analyze the root causes of test flakiness and classify them into 10 categories. We then study when flakiness is introduced in tests.

3.1 Categories of Flakiness Root Causes

We analyze in detail 201 LDFFT commits to classify the root causes of the flakiness likely fixed by these commits. We precisely classify the root causes for 161 commits, while the remaining 40 commits are hard to classify for various reasons as described later.

We split the root causes of flakiness into 10 categories. Some of these categories have been previously described by

Project	Language	LDFFT commits	Bug reports	LOC
HBase	Java	72	61	3199326
ActiveMQ	C++/Java/Scala	68	17	323638
Hadoop	Java	56	56	1348117
Derby	Java	49	42	719324
Harmony	C/C++/Java	30	27	1155268
Lucene	Java	27	12	384730
Tomcat	Java	16	0	316092
ServiceMix	Java	15	12	156316
ZooKeeper	Java	14	14	119139
Qpid	C++/Java	14	8	359968
CXF	Java	11	3	441783
Web Services	C++/Java	9	1	859462
Tuscany	Java	7	1	181669
Flume	Java	7	6	60640
Maven	Java	7	0	23000
OpenJPA	Java	7	4	484054
Oozie	Java	6	6	149315
Aries	Java	6	4	125563
Continuum	Java	5	0	130765
Subversion	C/Python	4	0	562555
Tapestry	Java	4	0	233382
Mesos	C++	4	0	219485
Flex	Java	4	1	2909635
HttpComponents	Java	3	0	52001
Accumulo	Java/Python	3	1	309216
Kafka	Scala	3	2	139736
Hive	Java	3	3	715967
Ambari	Java	2	2	115725
Jena	Java	2	0	349531
APR	C	2	0	64279
Jackrabbit	Java	2	2	295961
Sling	Java	2	2	226613
OpenEJB	Java	2	1	534262
Mahout	Java	2	1	121312
Avro	Java	2	2	131384
NPanday	C#	1	0	41947
Cassandra	Java	1	1	130244
UIMA	Java	1	1	218881
Roller	Java	1	0	75560
Portals	Java	1	0	228907
ODE	Java	1	1	114807
Buildr	Ruby	1	0	31062
Pig	Java	1	1	378181
Camel	Scala	1	0	580015
Archiva	Java	1	1	107994
XMLBeans	Java	1	0	211425
SpamAssassin	C	1	0	65026
Shindig	Java	1	0	151743
MINA	Java	1	0	417571
Karaf	Java	1	1	360403
Commons	Java	1	1	21343
Total	Multiple	486	298	20654322

Table 3: Many projects contain flaky tests

practitioners [13,33] and others we have identified by studying a number of similar flaky tests. The lower part of Table 2 shows the summary of our analysis. The first column lists the 10 categories, and the last row shows the 40 commits that are hard to classify. The remaining columns tabulate the distribution of these categories over various types of commits and various projects. We highlight the number of flaky tests for the top four projects, and the last column sums up the numbers for the remaining 47 projects.

We next discuss in more detail the top three categories that represent 77% of the 161 studied commits. We finally briefly summarize the other seven categories.

3.1.1 Async Wait

74 out of 161 (45%) commits are from the Async Wait category. We classify a commit into the Async Wait category 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. For example, a test (or the CUT) can spawn a separate service, e.g., a remote server or another thread, and there is no proper synchronization to wait for that service to be available before proceeding with the execution. Based on whether the result becomes available before (or after) it is used, the test can non-deterministically pass (or fail).

While our focus is on understanding flaky tests, we point out that lack of synchronization can also lead to bugs in the CUT even if it does not manifest in flaky tests. In fact, bugs caused by asynchronous wait fall in a subcategory of concurrency bugs that Lu et al. [24] call "order violation" because the desired order of actions between multiple threads is not enforced. We classify Async Wait as a separate category of root causes from Concurrency because it represents a large percentage of flaky tests with some common characteristics that are not shared among all tests that are flaky due to Concurrency.

We next describe an example Async Wait flaky test. This snippet is from the HBase project:

```
1 @Test
2 public void testRsReportsWrongServerName() throws Exception {
3   MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
4   MiniHBaseClusterRegionServer firstServer =
5     (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
6   HServerInfo hsi = firstServer.getServerInfo();
7   firstServer.setHServerInfo(...);
8
9   // Sleep while the region server pings back
10  Thread.sleep(2000);
11  assertTrue(firstServer.isOnline());
12  assertEquals(2,cluster.getLiveRegionServerThreads().size());
13  ... // similarly for secondServer
14 }
```

The test uses a cluster to start a server `firstServer` and then uses `Thread.sleep(2000)` to wait for it to ping back. If the server does not respond in a timely manner, e.g., because of thread scheduling or network delay, the test will fail. (The test has similar code for `secondServer`.) Basically the test intermittently fails based on how fast the server responds.

So far we have described the root cause of this flaky test, but as a preview of our overall analysis, we describe how we found this commit and how the developers fixed the flakiness. This commit matches the keyword "flak" as its commit message reads "HBASE-2684 TestMasterWrongRS flaky in trunk", which refers to the bug report ID HBASE-2684 [4]. The test was added in revision[3] 948632 and was flaky ever since. The developers fixed this test in revision 952837 by making two changes. First, they replaced each `Thread.sleep(2000)` statement (for `firstServer` and `secondServer`) with a call to `cluster.waitOnRegionServer(0)` that waits until the server responds back and then removes its corresponding thread. Second, to ensure that the test does not run indefinitely in case that the server cannot start for some reason, the developers added `(timeout=180000)` to the `@Test` annotation, which fails the test after 180 seconds. These changes completely removed the flakiness of this test: the developers removed the assumption that the server will respond within two seconds and explicitly expressed the condition to wait for before resuming the execution. Table 4 further categorizes how many fixes completely removed flakiness and how many just decrease the probability of flakiness.

[3]All the revision numbers refer to the Apache Software Foundation SVN repository [3].

3.1.2 Concurrency

32 out of 161 (20%) commits are from the Concurrency category. We classify a commit in this category when the test non-determinism is due to different threads interacting in a non-desirable manner (but not due to asynchronous calls from the Async Wait category), e.g., due to data races, atomicity violations, or deadlocks.

The source of non-determinism can be either in the CUT or in the test code itself. 10 out of 32 (30%) cases are due to non-determinism in the CUT and manifest by the intermittent failure of a corresponding test. Note that non-determinism in the test (or code) *execution* may or may not be a bug: the code could indeed have several correct different behaviors. However, if the test incorrectly accepts only a subset of these as passing behaviors, then the test has non-deterministic *outcome* and is definitely flaky.

We next describe an example Concurrency flaky test, where the cause is non-determinism in the CUT. This snippet is from the Hive project:

```
1 if (conf != newConf) {
2   for (Map.Entry<String, String> entry : conf) {
3     if ((entry.getKey().matches("hcat.*")) &&
4         (newConf.get(entry.getKey()) == null)) {
5       newConf.set(entry.getKey(), entry.getValue());
6     }
7   }
8   conf = newConf;
9 }
```

The code iterates over a map shared by multiple threads; if the threads modify the map concurrently, a `ConcurrentModificationException` is thrown [23]. This code led to flaky failures in several tests. The developers fixed this by enclosing the lines 3 to 6 in a synchronized block, making them execute atomically. We classify this case in the atomicity violation subcategory of the Concurrency category.

Our study finds the main subcategories of Concurrency flaky tests to match the common bugs from concurrent programming [24]: data races (9 out of 32), atomicity violations (10 out of 32), and deadlocks (2 out of 32). But we also identify a new prominent subcategory that we call "*bug in condition*" (6 out of 32)[4]. We classify a commit in this subcategory when some multithreaded code has a condition that inaccurately guards what threads can execute the guarded code. The problem is when this condition is too tight or too permissive. An example is in the project Lucene with bug report LUCENE-1950. The test is flaky because a code portion should be executed only by the thread named `main`, but the condition does not guard for that, so when another thread executes the code, the test fails. The fix was to strengthen the condition to check if the thread name is `main`.

3.1.3 Test Order Dependency

19 out of 161 (12%) commits are from the Test Order Dependency category. We classify a commit into this category when the test outcome depends on the order in which the tests are run. In principle, all tests in a test suite should be properly isolated and independent of one another; then, the order in which the tests are run should not affect their outcomes. In practice, however, it is not the case.

This problem arises when the tests depend on a shared state that is not properly setup or cleaned. The shared state can be either in the main memory (e.g., the static fields in Java) or some external resource (e.g., files or databases). Either a test expects to find the state as it was initialized but meanwhile another test changed that state (i.e., running one "polluter" test before another test fails the latter test), or a test expects the state to be set by the execution of another test that was not run (i.e., *not* running one "setup" test before another test fails the latter test). Hence, dependencies among tests result in unpredictable behavior when the test order changes. For example, the update from Java 6 to Java 7 changed the order in which JUnit finds the tests in a test class (due to the change in the reflection library) [19].

We next describe an example Test Order Dependency flaky test. This snippet is from the Hadoop project:

```
1 @BeforeClass
2 public static void beforeClass() throws Exception {
3   bench = new TestDFSIO();
4   ...
5   cluster = new MiniDFSCluster.Builder(...).build()
6   FileSystem fs = cluster.getFileSystem();
7   bench.createControlFile(fs, ...);
8
9   /* Check write here, as it is required for other tests */
10  testWrite();
11 }
```

The snippet is from a test class where one test (`testWrite`) writes to a file via `fs` preparing data to be read by several other tests. The developers incorrectly assumed that `testWrite` would always run first, but JUnit does not guarantee any particular test ordering. If JUnit runs some read test before `testWrite`, the test fails. The developers fixed this by removing the original `testWrite` test and adding a call to `testWrite` in the *@BeforeClass* as shown in line 10.

3.1.4 Other Root Causes

We briefly discuss the other seven categories of root causes of flakiness. Due to space limitation, we do not give a detailed example for each category. The relative ratio of flaky tests in these categories can be computed from Table 2.

Resource Leak. A resource leak occurs whenever the application does not properly manage (acquire or release) one or more of its resources, e.g., memory allocations or database connections, leading to intermittent test failures.

Network. Tests whose execution depends on network can be flaky because the network is a resource that is hard to control. In such cases, the test failure does not necessarily mean that the CUT itself is buggy, but rather the developer does not account for network uncertainties. From the results of our inspection, we distinguish two subcategories of flaky tests whose root cause is the network. The first subcategory is due to *remote* connection failures (60%), and the second subcategory is due to *local* bad socket management (40%).

Time. Relying on the system time introduces non-deterministic failures, e.g., a test may fail when the midnight changes in the UTC time zone. Some tests also fail due to the precision by which time is reported as it can vary from one platform to another. We discuss platform-dependent tests later (Sec. 4), and while they are not frequent, it is easy for developers to overlook the differences among platforms.

IO. I/O operations (in addition to those for networks) may also cause flakiness. One example we encountered was in the project Archiva where the code would open a file and read from it but not close it until the `fileReader` gets garbage collected. So, a test that would try to open the same file would either pass or fail depending on whether the `fileReader` was already garbage collected or not.

[4] 5 out of 32 cases are hard to classify in any subcategory.

Randomness. The use of random numbers can also make some tests flaky. In the cases that we analyzed, tests are flaky because they use a random number generator without accounting for all the possible values that may be generated. For example, one test fails only when a one-byte random number that is generated is exactly 0.

Floating Point Operations. Dealing with floating point operations is known to lead to tricky non-deterministic cases, especially in the high-performance computing community [6]. Even simple operations like calculating the average of an array require thorough coding to avoid overflows, underflows, problems with non-associative addition, etc. Such problems can also be the root cause of flaky tests.

Unordered Collections. In general, when iterating over unordered collections (e.g., sets), the code should not assume that the elements are returned in a particular order. If it does assume, the test outcome can become non-deterministic as different executions may have a different order.

Other Cases. We did not classify the root causes for 40 commits; of those, 7 do not fix a (non-deterministic) flaky test but rather fix some deterministic bug, and 6 do not actually fix a flaky test. The other 27 are hard to understand. We inspected each commit for several hours, reading the commit message and associated bug reports if any, reasoning about the patched code, and occasionally even trying to compile and run the tests. However, certain commits turned out to be hard to understand even after several hours. If the test or code change proven to be too complex, and there is not much information in the commit message or bug report to help us understand the root cause of flakiness, we mark it as unknown. We exclude all 40 cases from our further study.

See Finding F.1 and Implication I.1 in Table 1.

3.2 Flaky Test Introduction

We also study when the tests became flaky. From the 161 tests we categorized, 126 are flaky from the first time they were written, 23 became flaky at a later revision, and 12 others are hard to determine (mostly because tests were removed or relocated to other files). We analyze in more detail the 23 cases in which the tests became flaky later and identify two main causes.

The first reason is the addition of new tests that violate the isolation between tests. For example, consider a test t_1 that requires a shared variable to be zero but relies on the initialization phase to set that value rather than explicitly setting the value before beginning the execution. A newly added test t_2 sets the value of that shared variable to 1, without cleaning the state afterwards. If t_2 is run before t_1, the latter would fail.

The second reason for introducing flakiness after a test is first written is due to test code changes such patching a bug, changing the test functionality, refactoring a test, or incompletely patching some flakiness itself. For the 152 flaky tests for which we have evolution information, we calculate the average number of days it takes to fix a test to be 388.46. In sum, tools should extensively check tests when they are first written, but some changes can also introduce flakiness.

See Finding F.2 and Implication I.2 in Table 1.

4. MANIFESTATION

Manifesting the flakiness of flaky tests is the first step in fixing them. In practice, given a test failure that is suspected to be from a flaky test, the most common approach is to rerun the failing test multiple times on the same code to find whether it will pass (and thus is definitely flaky) or will not pass (and thus may be a real deterministic failure or might be still a flaky test that did not manifest in a pass in those multiple runs). While such rerunning can be useful in some cases, it has disadvantages. When the probability for a flaky test to change its outcome is low, rerunning it a few times may not be enough to manifest that it can change the outcome. Also, rerunning the tests multiple times is time consuming, especially if there are multiple test failures.

While inspecting each flaky test in our study, we have considered possible ways to automatically trigger the failure of the flaky test, either to definitely show that it can both fail and pass or at least to increase the probability of its failure (under the assumption that it passes majority of the time). Our analysis leads to findings that could help in developing automatic techniques for manifesting flaky tests.

4.1 Platform (In)dependency

To understand how flaky tests manifest in failures, the first question we want to answer is how many of those flaky failures only manifest on a particular platform. By a "platform", we refer to the underlying system the test is running on, including the hardware, operating system, JVM/JRE, etc. It differs from the *environment* (mentioned later), as it is not provided or controlled by the test or the application. An example of platform dependence is a test failing due to a different order of files on a specific file system.

From the flaky tests we categorized, we find that 154 out of 161 (96%) have outcome that does not depend on the platform. Namely, the test failures only depend on the events in the application code and not on any system call to the underlying platform. Of the 7 cases where the flaky failures can only be reproduced on a certain platform, 4 tests require a particular operating system to manifest the failure, 2 require a particular browser to manifest the failure, and only 1 requires a specific buggy JRE to manifest the failure.

See Finding F.3 and Implication I.3 in Table 1.

4.2 Flakiness Manifestation Strategies

For each of the top three categories of root causes of flaky tests, we next discuss how one could modify the tests from that category to manifest the flaky failures.

4.2.1 Async Wait

How many Async Wait flaky tests can be manifested by *changing* an existing time delay? To enforce a certain ordering in test execution, in many cases, developers use a `sleep` or `waitFor` method with a time delay. A `sleep` pauses the current thread for a fixed amount of time and then resumes its execution. With `waitFor`, we refer to a set of methods used to either let the current thread busy wait for some condition to become true or block the current thread until being explicitly notified. We find 25 out of 74 (34%) Async Wait flaky tests use `sleep` or `waitFor` with a time delay to enforce ordering. Their flaky failures can be simply manifested by *changing* the time delay of such method calls, e.g., decreasing the sleeping time in the test. For the other Async Wait flaky tests, developers do not use a common method call with a time delay to enforce certain ordering but either do not enforce any ordering at all or use some application-specific APIs to enforce the desired ordering.

See Finding F.4 and Implication I.4 in Table 1.

How many Async Wait flaky tests can be manifested by adding *one* new time delay? We find two factors that determine the difficulty of manifesting ASYNC WAIT flaky tests. The first is whether a test depends on external resources or not, because it is harder to control external resources. The second is whether the flaky failure involves only one ordering (where one thread/process is supposed to wait for an action from another thread/process to happen) or more orderings. Manifesting failures for multiple orderings would require carefully orchestrating several actions, which is much harder than just delaying one action.

Our study finds that the majority of ASYNC WAIT flaky tests, 67 out of 74 (91%), do not "wait" for external resources. The few that do wait include a test that waits for a specific process to be started by the underlying OS and a test that waits for a response from the network.

Our study also finds that most ASYNC WAIT flaky tests involve only one ordering. In fact, we find only 5 out of 74 (7%) cases with multiple orderings, e.g., a client waits for multiple servers to be started in a certain order.

Overall, 63 out of 74 (85%) ASYNC WAIT flaky tests do not depend on external resources *and* involve only one ordering. Their flaky failures can be manifested by *adding* only one time delay in the code, without the need of controlling the external environment. (Note that several tests can be manifested as flaky by either adding a new time delay or changing an existing time delay.) While finding the appropriate place to add the delay could be quite challenging, researchers can attempt to build on the heuristic or randomization approaches that were successfully used in manifesting concurrency bugs [10, 35].

See Finding F.5 and Implication I.5 in Table 1.

4.2.2 CONCURRENCY

How many threads are involved? For all the 32 flaky tests caused by CONCURRENCY, we also study how many threads are involved in the test failure. We find out that 13 cases involve more than two threads. Seemingly contradictory, Lu et al. [24] found out that most concurrency bugs *require* only two threads to manifest. However, our result does not contradict their finding because we study real tests that already contain multiple threads, whereas they study bugs in the code and reason about tests that could have been written to expose those bugs. To reproduce CONCURRENCY flaky failures, all the existing tests with multiple threads that we studied could be simplified into at most two threads. Interestingly enough, we even find one flaky test for which only one thread suffices to trigger a deadlock bug.
How many Concurrency flaky tests do not depend on external resources? We also find out that 31 out of 32 (97%) CONCURRENCY flaky tests do not depend on external resources. In other words, their failures are only caused by concurrent accesses to the objects in the main memory.

See Finding F.6 and Implication I.6 in Table 1.

4.2.3 TEST ORDER DEPENDENCY

We further study the source of dependency for each TEST ORDER DEPENDENCY flaky test. We identify three sources of dependency. We call the first one "Static field in TEST" (3 out of 19), which means that several tests access the same static field declared in the test code, without restoring the state of that field in a `setUp` or `tearDown` method. We call the second one "Static field in CUT" (6 out of 19), which means that the shared static field is declared in the CUT rather than in the test code itself. Test-order dependencies of the first two kinds can be exposed by recording and comparing object states. We call the third one "External dependency" (10 out of 19), which means that the dependency is caused by some external environment, such as shared file or network port, and not by a static field.

Of the 19 TEST ORDER DEPENDENCY flaky tests, we find that more than half are caused by an external dependency. Those tests cannot be easily manifested by recording and comparing internal memory object states but instead require more sophisticated techniques to model the state of external environment or to rerun tests with different order [37].

See Finding F.7 and Implication I.7 in Table 1.

Category	Fix type	Total	Remove	Decrease
ASYNC WAIT	Add/modify `waitFor`	42	23	19
	Add/modify `sleep`	20	0	20
	Reorder execution	2	0	2
	Other	10	9	1
CONCURRENCY	Lock atomic operation	10	10	0
	Make deterministic	8	8	0
	Change condition	3	3	0
	Change assertion	3	3	0
	Other	8	8	0
TEST ORDER DEPENDENCY	Setup/cleanup state	14	14	0
	Remove dependency	3	3	0
	Merge tests	2	2	0

Table 4: Flaky test fixes per category

5. FIXING STRATEGIES

Developers fix various root causes of flakiness in different ways. We identify the main strategies that they use and extract insights for practitioners to use in manually fixing flaky tests and for researchers to use as starting points to develop tools to automate this process. For the top three categories of flaky tests, we give a detailed description of the common fixes and discuss their effectiveness in removing the flakiness. For the other categories, we only briefly discuss the fixes. Some of the fixes for flaky tests change the CUT, so we also study those cases.

5.1 Common Fixes and Effectiveness

We describe the main strategies that developers use to fix flaky tests of different categories. Table 4 summarizes the fixes for the top three categories. An interesting property of these fixes is that they do not always completely eliminate the root cause, namely they do not turn a (non-deterministic) flaky test into a fully deterministic test. Rather, some fixes change the code such that the test is less likely to fail, although it could still fail. The column "Remove" shows the number of fixes that completely remove the flakiness, while the column "Decrease" shows the number of fixes that only decrease the chance of failures in the flaky tests.

5.1.1 ASYNC WAIT

Common fixes. The key to fixing ASYNC WAIT flaky tests is to address the order violation between different threads or processes. We describe the strategies used in practice and evaluate their effectiveness.

Fixes using `waitFor` calls: 42 out of 74 (57%) ASYNC WAIT flaky tests are fixed via some call to `waitFor`. Recall that those calls block the current thread until a certain condition is satisfied or a timeout is reached. The fixes add a

new call, modify an existing call (either the condition or the timeout), or replace an already existing `sleep` (36% of these fixes replace a `sleep` with a `waitFor` call). The latter shows that `waitFor` is a preferred mechanism that developers should use whenever possible. Also, out of all the `waitFor` fixes, 46% have a time bound, while the others are unbounded.

Fixes using `sleep` calls: 20 out of 74 (27%) ASYNC WAIT flaky tests are fixed by stalling some part of the code for a pre-specified time delay using `sleep`. 60% of these cases increase the waiting time of an already existing sleep, while the other 40% add a sleep that was missing (conceptually increasing the time delay from 0 up to the specified bound in the added `sleep`). This shows the effect that machine speed variation can have on flaky tests, especially for those 60% of the cases where the `sleep` was already there but the waiting time was not long enough on some slower (or faster) machines, leading to intermittent failures.

Fixes by reordering code: 2 out of 74 (3%) ASYNC WAIT flaky tests are fixed by reordering code. Instead of simply using `sleep` to wait for some time, the developer finds a piece of code that can be executed such that the execution of that code achieves the delay and hopefully achieves the particular event ordering. A benefit of executing some code rather than simply using `sleep` is that useful computation gets done, but the problem remains that the developer cannot precisely control the time taken for that computation.

Other: 10 out of 74 (14%) ASYNC WAIT flaky tests are very specific to the code and hard to generalize. For example, some of these fixes use an application-specific API method to trigger an event so the ordering is not violated.

Effectiveness of fixes in alleviating flakiness. Using a `sleep` is rarely a good idea when writing tests. As analyzed in our earlier work that focused on writing multithreaded tests [16], the use of `sleep` makes the test unintuitive, unreliable, and inefficient. It is hard to reason from the `sleep` calls what ordering among what events they try to enforce. Moreover, `sleep` calls cannot provide the guarantee that the ordering that the developer wants to enforce among events will indeed happen within the amount of time given in `sleep`. For that reason, developers tend to over-estimate the time needed in `sleep` calls, which makes the tests rather inefficient, because most of the time, the event can finish way before the time bound (and yet occasionally it does not finish before the time bound, thus intermittently failing the test). For all these reasons, we find that the fixes where `sleep` calls are used to fix ASYNC WAIT flaky tests are only decreasing the chance of a flaky failure: running tests on different machines may make the `sleep` calls time out and trigger the flaky failures again.

Using a `waitFor` is the most efficient and effective way to fix ASYNC WAIT flaky tests. Because `waitFor` makes explicit the condition that has to be satisfied before the execution can proceed, it becomes much easier to understand what ordering the test expects. Moreover, the execution is more efficient because it can proceed as soon as the condition is satisfied, rather than waiting for some time bound when the condition may or may not be satisfied. In fact, Table 4 shows that 23 out of 42 (55%) cases with `waitFor` completely remove the flakiness. The remaining cases only decrease the flakiness because those `waitFor` calls are bounded with a timeout. Even when there is no explicit timeout on some `waitFor`, the test itself can have a timeout. Such a timeout is practically useful for preventing a single test from hanging the entire test suite forever. We find that developers set much higher timeouts when using `waitFor` rather than `sleep`; in particular, the average waiting time for `waitFor` calls in our cases is 13.04 seconds, while the average waiting time for `sleep` calls is 1.52 seconds. The higher upper bound on `waitFor` makes them more robust against flakiness when the condition that is being waited for gets delayed; at the same time, `waitFor` is more efficient than `sleep` when the condition gets available earlier than expected. While the tests with `waitFor` may still fail when run on an extremely slow machine, using `waitFor` is much more efficient and reliable than using a `sleep`.

Reordering code is ineffective as using `sleep` calls, for the similar reasons. The only advantage of reordering code over sleeps is that the waiting time is not purely idle time but rather some useful computation happens. However, the developer does not have a precise control over the amount of time taken for that computation. Hence, the flaky tests can still fail after the code is reordered.

See Finding F.8 and Implication I.8 in Table 1.

5.1.2 CONCURRENCY

Common Fixes. Concurrency bugs are caused by undesired interleavings among different threads. The flaky tests from the CONCURRENCY category in our study are similar to common concurrency bugs in the CUT [24]. We find four main strategies of fixes for CONCURRENCY flaky tests.

Fixes by adding locks: 10 out of 32 (31%) CONCURRENCY flaky tests are fixed by adding a lock to ensure mutual exclusion for code that is supposed to be accessed by one thread at a time. 8 of these fixes address atomicity violation, and 1 each addresses deadlock and race condition.

Fixes by making code deterministic: 8 out of 32 (25%) CONCURRENCY flaky tests in the study are fixed by making the execution deterministic. The specific changes include modifying code to eliminate concurrency, enforcing certain deterministic orders between thread executions, etc.

Fixes by changing concurrency guard conditions: 3 out of 32 (9%) CONCURRENCY flaky tests in our study are fixed by changing the guard conditions in the test code or the CUT. For example, developers use a condition check to only allow certain threads to enter certain part of the code at the same time. If that condition does not take into account all the possible scenarios, the test may become flaky.

Fixes by changing assertions: 3 out of 32 (9%) CONCURRENCY flaky tests in our study are fixed by changing assertions in the test code. Although non-determinism is permitted by the concurrent program, the test assertion fails to accept all valid behaviors. The fix is to account for all valid behaviors in the assertion.

Others: The remaining fixes for CONCURRENCY flaky tests vary from case to case, and they are usually specific to the application. For instance, developers may fix a CONCURRENCY flaky test due to race condition by making a specific shared variable to be thread local.

Effectiveness of fixes in alleviating flakiness. In our study we find that all the fixes for CONCURRENCY flaky tests completely remove flakiness in the test. As long as the root cause of the CONCURRENCY flaky test is correctly identified and understood by developers, the committed fix always resolves the problem completely.

Our finding seemingly contradicts a previous study [24] by Lu et al. who found a number of concurrency bugs to be

hard to fix and typically have at least one incomplete patch attempting to fix the bug but not completely fixing it. The difference is likely due to the different methodologies we use. In particular, we analyze one *committed* fix per a flaky test in the repository, and if the same flaky test has multiple commits, we pick the *last commit* to fully understand the flakiness and how it was resolved. It is quite possible that our study missed some incomplete fixes. Also, it is possible that some incomplete fixes were proposed with the bug reports, but developers rejected these incomplete fixes without committing them to the repository at all. In contrast, Lu et al. study concurrency bugs from bug reports and not from commits. Also, one reason that Concurrency flaky tests can be easier to fix than general concurrency bugs is that general bugs may have no tests, making it harder to debug than when a specific test is present.

See Finding F.9 and Implication I.9 in Table 1.

5.1.3 Test Order Dependency

Common Fixes. Test Order Dependency flaky tests are not easy to debug because it may be difficult to find out which other test is (or tests are) interdependent with the intermittently failing test. However, once developers figure out the dependency, the fix is usually simple and obvious. We classify the common fixes into three main strategies.

Fixes by setting up/cleaning up states: 14 out of 19 (74%) Test Order Dependency flaky tests are fixed by setting up or cleaning up the state shared among the tests. Basically, the test needs to set up the state before it executes, clean up the state after it finishes, or both.

Fixes by removing dependency: 3 out of 19 (16%) Test Order Dependency flaky tests are fixed by making local copies of the shared variable and removing the dependency on it.

Fixes by merging tests: 2 out of 19 (10%) Test Order Dependency flaky tests are fixed by merging dependent tests. For example, developers copy the code in one test into another one and remove the first test.

Effectiveness of fixes in alleviating flakiness. All the fixes we find in our study for Test Order Dependency flaky tests completely remove the flakiness. However, the first two strategies of fixes (Setup/cleanup state and Remove dependency) are better than the last strategy. Setting up or cleaning up state ensures that even if tests do depend on the shared state, that state is always found as expected, independent of the order in which the tests run. The second strategy remedies the case where two or more tests access and modify a common field that is expected to be local; this strategy is somewhat related to concurrency, even when tests are executed on one thread.

Merging dependent tests makes tests larger and thus hurts their readability and maintainability, although it does remove the flakiness. Moreover, merging smaller tests into large tests limits the opportunities for parallelizing the test suite or applying test selection and prioritization techniques.

See Finding F.10 and Implication I.10 in Table 1.

5.1.4 Others

We next discuss briefly the common ways of fixing the other categories of flaky tests.

Resource Leak. This category is one of the hardest for generalizing the specific fixes. Fowler suggests to manage the relevant resources through resource pools [13]. When a client needs a resource, the pool provides it. The pool can be configured to either throw an exception if all resources are in use, or to grow. When the client is done with the resource, it should return it to the pool. A resource leak occurs if a client does not return resources. These leaky clients can be detected by reducing the pool size so that requesting a resource triggers an exception, and fixed by properly returning the resources.

Network. Whether for internal communication between processes on the same machine through sockets, or getting data and services from remote servers, software relies extensively on the network. This dependency results in non-determinism in test and code execution. One of the best fixes for this category is to use mocks. Whenever the use of mocks is non practical, the flakiness can be remedied by using `waitFor`.

Time. Time precision differs from one system to another. In general, tests should avoid using platform dependent values like time.

IO. Because they deal with external resources, I/O operations can cause intermittent test failures. Therefore, the developer should make sure to close any opened resource (file, database, etc.) and to use proper synchronization between different threads sharing the same resource.

Randomness. This category is associated with random number generation. To avoid/fix flakiness due to randomness, the developers should control the seed of the random generator such that each individual run can be reproduced yet the seed can be varied across runs. The developer should also handle the boundary values that the random number can return, e.g., zero can be a problem in some scenarios.

Floating Point Operations. Floating-point operations are non-deterministic by nature, and can cause a lot of problems if not handled correctly. In general, one has to be careful when dealing with floating point operations. Imprecision is not avoidable, but one should aim for determinism, and it is good practice to have test assertions as independent as possible from floating-point results.

Unordered Collections. Flakiness due to unordered collections arises whenever the developer assumes that the API guarantees a certain order that it does not. In general, a good programming practice is to write tests that do not assume any specific ordering on collections unless an explicit convention is enforced on the used data structure.

See Finding F.11 and Implication I.11 in Table 1.

5.2 Changes to the code under test

38 out of 161 (24%) of the analyzed commits fix the flakiness by changing both the tests and the CUT. We classify the changes to the CUT as follows.

Deterministic Bug. In some cases, the source code is deterministic and contains a bug. The flakiness is in the test code, but the test, by failing even intermittently, helps in uncovering the real bug in the CUT.

Non-Deterministic Bug. The source code is non-deterministic and contains a bug (e.g., a race condition) that causes flakiness. The flaky failures again help to uncover the bug.

Non-Deterministic No Bug. The code is non-deterministic but contains no bug. However, the developers decide to make it more deterministic to avoid some cases. Such change can help in writing tests because the flaky test need not consider all the possible correct results.

See Finding F.12 and Implication I.12 in Table 1.

6. THREATS TO VALIDITY

Our study is empirical and has the common threats of internal, external, and construct validity as any empirical study. We focus on more specific issues.

Choice of projects. We study only a subset of all software projects, so our results may not generalize. To address this threat, we consider *all* the projects from the Apache Software Foundation. We examine a diverse set of projects with more than 20 million LOC (just in the projects with flaky tests) and 1.5 million commits. The projects use different languages and various types of applications (web server, databases, cloud, graphics, mail, build-management, etc.). However, Apache does not contain many mobile applications (such as Android Apps) or GUI tests [26], so some of our findings and implications, such as platform independence, may not apply to those cases.

Selection Criteria. In selecting the cases to study, we (1) rely only on the commit log, (2) use specific keywords to search it, and (3) study only the fixed tests. Section 2 explains in detail the benefits of choices (1) and (3), in particular finding fixed flaky tests that other approaches (such as relying on bug reports) could miss. However, we could still miss many flaky tests that are only reported in bug reports and never got fixed. Concerning choice (2), we search the commit messages using only two keywords, "intermit" and "flak", so there is no guarantee on the recall of our search. In fact, we believe our search could miss many flaky tests whose fixes could use words like "concurrency", "race", "stall", "fail", etc. In our future work we intend to expand our search for likely fixes of flaky tests. The large number of commits we already find with just two keywords shows that flaky tests are an important problem to study.

Manual inspection. Labeling and characterizing commits manually can lead to incorrect labeling, e.g., our analysis phase found some false positives from our filtering phase. However, the number of false positives is relatively low. Further, to minimize the probability of error, two authors independently inspect every commit and then merge the results.

7. RELATED WORK

Several researchers and practitioners have pointed out problems with non-deterministic tests [1,7,12,13,20,21,26,29,32, 34,37]. For example, Fowler [13] described non-deterministic test outcomes as a recurring problem in regression testing and outlined some ways for avoiding and manually fixing flaky tests. Memon and Cohen [26] pointed out a few possible reasons that make GUI tests flaky. Lacoste [20] also described some of the unfortunate side-effects of flaky tests in automated regression testing, e.g., some features may miss a release deadline because of intermittent test failures. More recently, Marinescu et al. revealed a number of non-deterministic test suites in their study that analyzed evolution of test suite coverage [25].

Non-deterministic bugs and tests. Most existing work on non-deterministic bugs (either in the CUT or in the test code) focuses on *one* specific category of non-determinism causes. For example, several researchers focus on Test Order Dependency. Zhang et al. formalized the test dependency problem, studied real-world test suites with test dependency problems, and implemented several techniques to identify these tests by reordering test runs [37]. Muşlu et al. found that isolating unit tests can be helpful in detecting faults, but enforcing isolation can be computationally expensive [29]. Bell and Kaiser proposed an automatic approach for isolating unit tests in Java applications by tracking all side-effects on shared memory objects and undoing these effects between tests [7].

Concurrency was also well studied. For example, Farchi et al. summarized common bug patterns in concurrent code and employed static analysis to detect some of them [11]. Lu et al. published a comprehensive characteristic study [24] examining bug patterns, manifestation, and fixes of concurrency bugs. Compared to prior work, our study focuses on characterizing flaky tests across *all* categories, and we start from commit logs rather than just bug reports. Our results show that some categories of flaky tests such as Async Wait and Concurrency are more prevalent than Test Order Dependency and should likely get more attention.

While we focus on flaky tests, some causes that we find are general non-determinism bugs in the CUT. However, we ignore the cases of bugs in the CUT that do not result in test flakiness, but we do include the cases of test flakiness that may have no bug in the CUT (e.g., Test Order Dependency). We believe that the causes and fixes for flaky tests differ enough from general bugs in the CUT to warrant more special focus.

Bug-fixes study. Researchers have also studied different characteristics of bug fixes. For example, Murphy-Hill et al. conducted a large study to find factors that influence how bugs get fixed [30]. Bachman et al. found that many bug-fix commits do not have corresponding bug reports [5]; their results motivate us to start from commits instead of bug reports. Automated techniques have also been proposed to fix concurrency bugs by Jin et al. [18]. Our study revealed a number of different strategies for fixing flaky tests in different categories, and we believe that our findings can help in developing more automated techniques for fixing bugs.

Fixing tests. Daniel et al. proposed an automated technique for fixing broken tests [9]. Yang et al. proposed a different technique using Alloy specifications to repair tests [36]. However, existing test repair only focuses on broken tests that fail deterministically, while we study flaky tests that fail non-deterministically.

8. CONCLUSIONS

Regression testing is important but can be greatly undermined by flaky tests. We have studied a number of fixes to flaky tests to understand the common root causes, identify approaches that could manifest flaky behavior, and describe common strategies that developers use to fix flaky tests. Our analysis provides some hope for combating flaky tests: while there is no silver bullet solution that can address all categories of flaky tests, there are broad enough categories for which it should be feasible to develop automated solutions to manifest, debug, and fix flaky tests.

9. ACKNOWLEDGMENTS

We thank John Micco for sharing personal experience about flaky tests at Google, and Sebastian Elbaum and Sai Zhang for the valuable discussions about this work. This research was partially supported by the NSF Grant Nos. CNS-0958199 and CCF-1012759, and the DARPA grant FA8750-12-C-0284. Farah Hariri was also supported by the Saburo Muroga Endowed Fellowship.

10. REFERENCES

[1] API design wiki - OrderOfElements. http://wiki.apidesign.org/wiki/OrderOfElements.

[2] Android FlakyTest annotation. http://goo.gl/e8PILv.

[3] Apache Software Foundation SVN Repository. http://svn.apache.org/repos/asf/.

[4] Apache Software Foundation. HBASE-2684. https://issues.apache.org/jira/browse/HBASE-2684.

[5] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *FSE*, 2010.

[6] E. T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *POPL*, 2013.

[7] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *ICSE*, 2014.

[8] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan. An empirical study on inconsistent changes to code clones at the release level. *SCP*, 2012.

[9] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *ASE*, 2009.

[10] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *CCPE*, 2003.

[11] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[12] Flakiness dashboard HOWTO. http://goo.gl/JRZ1J8.

[13] M. Fowler. Eradicating non-determinism in tests. http://goo.gl/cDDGmm.

[14] P. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *ICSE*, 2010.

[15] P. Gupta, M. Ivey, and J. Penix. Testing at the speed and scale of Google, 2011. http://goo.gl/2B5cyl.

[16] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *FSE*, 2011.

[17] Jenkins RandomFail annotation. http://goo.gl/tzyC0W.

[18] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[19] JUnit and Java7. http://goo.gl/g4crZL.

[20] F. Lacoste. Killing the gatekeeper: Introducing a continuous integration system. In *Agile*, 2009.

[21] T. Lavers and L. Peters. *Swing Extreme Testing*. 2008.

[22] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *ASID*, 2006.

[23] Y. Lin and D. Dig. CHECK-THEN-ACT misuse of Java concurrent collections. In *ICST*, 2013.

[24] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.

[25] P. Marinescu, P. Hosek, and C. Cadar. Covrig: A framework for the analysis of code, test, and coverage evolution in real software. In *ISSTA*, 2014.

[26] A. M. Memon and M. B. Cohen. Automated testing of GUI applications: models, tools, and controlling flakiness. In *ICSE*, 2013.

[27] J. Micco. Continuous integration at Google scale, 2013. http://goo.gl/0qnzGj.

[28] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of Unix utilities. *CACM*, 1990.

[29] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. ESEC/FSE, 2011.

[30] E. R. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *ICSE*, 2013.

[31] Spring Repeat Annotation. http://goo.gl/vnfU3Y.

[32] P. Sudarshan. No more flaky tests on the Go team. http://goo.gl/BiWaE1.

[33] 6 tips for writing robust, maintainable unit tests. http://blog.melski.net/tag/unit-tests.

[34] TotT: Avoiding flakey tests. http://goo.gl/vHE47r.

[35] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting where it hurts: An automatic concurrent debugging technique. In *ISSTA*, 2007.

[36] G. Yang, S. Khurshid, and M. Kim. Specification-based test repair using a lightweight formal method. In *FM*, 2012.

[37] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, M. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA*, 2014.