Can We Trust Test Outcomes?

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

Abstract—Software development is an on-going process. To ensure that the changes made do not break previous functionality, a regression test suite is maintained along with the source code. The correctness and the side effects of code changes are judged based on the output of the tests. However, are these tests’ outcomes always correct? Does a test failure imply a bug and do all tests passing imply a bug free code? The answer is no. Experienced researchers and practitioners know that there is an inherent non-determinism in testing.

This paper contributes the first comprehensive study on test outcome non-determinism. We target 153 open-source projects with a novel methodology starting from the commit logs, which enabled us to identify many unreported cases of tests with non-deterministic outcomes (also called flaky tests). We were able to detect 1129 commits from 51 projects that are related to flaky deterministic outcomes (also called flaky tests). We have been able to detect 1129 commits from 51 projects that are related to flaky tests, out of which we study 486 commits which are about distinct flaky tests. 61% of these flaky tests do not have associated bug reports. In our study, we focus on 81 commits to analyze in depth. We identify 11 causes of flakiness, provide examples for each, and give recommendations on how to fix them. We also discuss other interesting findings such as the effectiveness of common fixes, the evolution of flaky tests, how hard it is to reproduce a flaky test, and how hard it is to patch it.

I. INTRODUCTION

A. Motivation

Software development is an on-going process that never ends. Even after the product is released, developers keep changing it for many purposes: adding new features, fixing bugs, refactoring old code, etc. To ensure that the changes made do not break previous functionality, a regression test suite is maintained along with the source code.

The correctness and the side effects of code changes are judged based on the output of the tests. The testing scenario goes as follows: If all tests pass, the developer enjoys a restful night’s sleep. If some test fails, the developer would be alarmed and starts debugging, questioning the correctness of the code modified. But is this how the practices should be? The answer is no. Experienced researchers and practitioners know that there is an inherent non-determinism in testing.

We call a test non-deterministic (also known as flaky) if it passes or fails intermittently for the same code version and the same inputs. In order to submit a change, the developer expects the whole test suite to pass. Therefore, such tests are a big pain even if only few of them exist. Developers do not have a principled way for handling flaky tests. Sometimes, when a test fails, it is run many times trying to see if it is flaky. For example, Android has the @FlakyTest annotation and JUnit has the @Repeat annotation that can mark a flaky test to be run multiple times in case of failure. This would waste a lot of time and resources. Some other times failing tests are disabled completely. This is also not a good solution since we are disregarding the functionality that the disabled test covers and hence reducing from the value of the test suite. Lacoste [20] describes practical problems with flaky tests that he and his team faced. For example, one of the side effects of flaky tests that he mentions is that some branches miss a release because of an intermittent failure. In that case, the whole test suite has to be rerun after discovering the flaky test and fixing it.

Flakiness is as much of a problem for researchers as it is for practitioners. A lot of research in the literature has been done to study the effectiveness of regression test suites. Techniques such as test suite selection [6], [10], [28], [35], [37] test suite minimization [18], [29], [36], [38] and test suite prioritization [9], [17], [19], [30] try to run the suite faster or detect faults earlier. Other techniques [24] aim at enhancing the fault detection capabilities of test suites based on some predefined coverage criteria. Some work has also been done to make sure that tests are updated with the code [13], [16]. Many existing tools as well tackle the problem of test generation [1], [7], [14], [31], [34]. However, none of these testing techniques question the tests’ outcomes, and hence they may give misleading results, e.g., claiming that a test fails when it actually passes, or vice versa.

B. Contributions

This paper contributes the first comprehensive study on test outcome non-determinism. We target all projects of the Apache Software Foundation. Our novel methodology of starting from commit log messages rather than bug reports has enabled us to identify many flaky tests that were not reported in bug databases. In our analysis, we answer the following research questions:

RQ1: What are the root causes of flakiness? When a test fails, we don’t know if the failure is due to a bug introduced by the changes made or simply a flaky test that found the adequate setting to manifest. Knowing the sources of flakiness lays down the basic programming practices for deterministic test generation. Moreover, identifying the causes of non-determinism in a test suite is the first step in the road of developing automatic tools for flaky test detection.
RQ2: How are flaky tests usually fixed? Once the flaky tests are categorized, the next interesting question to answer is: what are the fixes for these sources of non-determinism. Based on the study we conducted, we analyze common ways of fixing each type of flakiness and give recommendations.

The rest of the paper is organized as follows: section II shows the setting of the study and the methodology used, section III answers RQ1 discussing the sources of flakiness with examples for each category, section IV answers RQ2 where fixes of flaky tests are discussed, section V is a discussion of other interesting findings such as the effectiveness of common fixes, the evolution of flaky tests, how hard it is to reproduce a flaky test, and how hard it is to patch it, section VI is a discussion of the threats to validity, section VII is related work, and finally section VIII is future work.

II. METHODOLOGY

Many empirical studies about bugs start from bug reports [5], [15], [22], [23], [25]. In our study, we adopt a novel methodology that considers commit messages as the starting point. In order to analyze the problem of flaky tests, we mined the complete commit log of all projects from the Apache Software Foundation. We identify commit messages that may indicate a fix of a flaky test by searching for the keywords “flak” and “intermit”. The search yielded 1129 results. Figure 1 shows the results of the initial labeling phase. Each result was inspected by two of the authors separately, then they merge the results. The first part of Figure 1 shows the results of the initial labeling phase.

Analysis Phase After this initial labeling, we study some of the commits that attempt to fix a flaky tests in more depth. Out of the 486 commits, we examined 81. Each commit is examined in detail by one of the authors who answers the following questions:

- what is the category of this flaky test?
- what is the root cause of its flakiness?
- how was it fixed?
- how can the flakiness be reproduced?
- when was the flaky test first introduced?
- was the test flaky the first time it was written?

The result is then inspected by a second author.

Figure 2 shows some statistics collected from analyzing the commit logs of the Apache Software Foundation. 51 out of the 153 projects in Apache have at least one flaky test. For each project, we show the number of commits fixing a flaky test, the number of bug reports associated with these flaky tests, and the total number of lines of code. Notice that from the 486 commits about unique flaky tests, only 297 (61%) mention an associated bug report. From the 486 commits about distinct flaky tests, we chose 81 to study in more detail.

III. SOURCES OF FLAKINESS

In this section, we answer research question RQ1. During our study, we came across 11 different root causes for flakiness. Figure 3 shows the distribution of the 81 tests we examined between the different categories.

### A. Asynchronous Wait

Asynchronous wait occurs when the current execution depends on data provided by a separate service like a remote server or another thread, and there is no proper synchronization made to wait for that service to be available before proceeding with the execution. This category lies in the intersection of two categories: concurrency and network. In both cases, the common problem is improper synchronization between the producer and the consumer of data.

Let us walk through a couple of examples to get a flavor of flaky tests due to asynchronous wait. Asynchronous wait for a server

The first example we give is for a flaky test that fails intermittently because of a server not responding fast enough. It is found in Hbase project. The fix revision number is 952837 and there exists an associated bug report with ID HBASE-2684. In the test class TestMasterWrongRS, the test testRsReportsWrongServerName() is waiting for a server to ping back. The code is shown below:

```java
public void testRsReportsWrongServerName() throws Exception {
    MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
    MiniHBaseClusterRegionServer firstServer = (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
    HRegionServer secondServer = cluster.getRegionServer(1);
    Intermit flak
      total 854 275
      yes, uniq 460 97
      yes, uniq, fix 398 88
    
    Async wait 20 7
    Concurrency 14 2
    Test order dependency 11 2
    GUI 6 1
    Network 5 1
    Randomness 0 2
    Resource leak 5 0
    Unordered collections 0 2
    IO 1 0
    Floating Point operations 1 0
    Time 1 0

    Fig. 1: Keywords used to find commit messages about flaky tests
```
Lines 11 and 12 show that the thread sleeps for two seconds waiting for the sever to respond, and then an assertion checks that the server is online. The test fails intermittently whenever the server does not respond within two seconds. This could be due to many reasons: network congestion, server overloaded, no network connection, etc... Here is how the problem is fixed:

```java
@Timeout(180000)
public void testSleepWhileRegionServerPingsBack() throws Exception {
  MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
  HRegionServer firstServer = (MiniHBaseClusterRegionServer) cluster.getRegionServer(0);
  HRegionServer secondServer = (MiniHBaseClusterRegionServer) cluster.getRegionServer(1);
  HServerInfo hsi = firstServer.getServerInfo();
  hsi.getInfoPort(), hsi.getHostname());
  assertEquals(2,
    cluster.getLiveRegionServerThreads().size());
  }
}
```

The sleep statement is replaced with a call to `waitOnRegionServer` which waits until server responds back and removes its corresponding thread. In order to make sure that the test does not run indefinitely, a timeout of 180 seconds is set in the `@Test` annotation.

### Asynchronous wait for a thread

The second example we give is for a test that intermittently fails because of a thread not waiting long enough for another thread to finish. Below, we show the code of the test `testContinuousScheduling` from the test class `TestFairScheduler` in the Hadoop project.

```java
public void testContinuousScheduling() throws Exception {
  MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
  FairScheduler fs = new FairScheduler();
  Configuration conf = createConfiguration();
  conf.setBoolean FAIR_SCHEDULER_CONTINUOUS_SCHEDULING, true); // set continuous scheduling enabled
  conf.setBoolean FAIR_SCHEDULER_CONFIGURATION, CONTINUOUS_SCHEDULING_ENABLED, true); // set scheduler configuration
  fs.reinitialize(conf, resourceManager.getRMContext());
  Assert.assertTrue("Continuous scheduling should be enabled.",
    fs.isContinuousSchedulingEnabled());
  }
```

---

**Fig. 2:** Many projects contain flaky tests

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of flaky tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Async wait</td>
<td>27</td>
</tr>
<tr>
<td>Concurrency</td>
<td>15</td>
</tr>
<tr>
<td>Test order dependency</td>
<td>16</td>
</tr>
<tr>
<td>GUI</td>
<td>7</td>
</tr>
<tr>
<td>Network</td>
<td>5</td>
</tr>
<tr>
<td>Randomness</td>
<td>2</td>
</tr>
<tr>
<td>Resource leak</td>
<td>4</td>
</tr>
<tr>
<td>Unordered collections</td>
<td>2</td>
</tr>
<tr>
<td>IO</td>
<td>1</td>
</tr>
<tr>
<td>Floating point operations</td>
<td>2</td>
</tr>
<tr>
<td>Time</td>
<td>1</td>
</tr>
</tbody>
</table>

**Fig. 3:** Flaky Tests Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of flaky tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>HBase</td>
<td>Java</td>
</tr>
<tr>
<td>ActiveMQ</td>
<td>C++/Java/Scala</td>
</tr>
<tr>
<td>Hadoop</td>
<td>Java</td>
</tr>
<tr>
<td>Derby</td>
<td>Java</td>
</tr>
<tr>
<td>Harmony</td>
<td>C/C++/Java</td>
</tr>
<tr>
<td>lucene</td>
<td>Java</td>
</tr>
<tr>
<td>tomcat</td>
<td>Java</td>
</tr>
<tr>
<td>servicemodel</td>
<td>Java</td>
</tr>
<tr>
<td>zookeeper</td>
<td>Java</td>
</tr>
<tr>
<td>quartz</td>
<td>Java</td>
</tr>
<tr>
<td>flume</td>
<td>Java</td>
</tr>
<tr>
<td>maven</td>
<td>Java</td>
</tr>
<tr>
<td>openpoll</td>
<td>Java</td>
</tr>
<tr>
<td>ooze</td>
<td>Java</td>
</tr>
<tr>
<td>aties</td>
<td>Java</td>
</tr>
<tr>
<td>continuum</td>
<td>Java</td>
</tr>
<tr>
<td>subversion</td>
<td>C/Python</td>
</tr>
<tr>
<td>tapestry</td>
<td>Java</td>
</tr>
<tr>
<td>mesos</td>
<td>C++</td>
</tr>
<tr>
<td>flex</td>
<td>Java</td>
</tr>
<tr>
<td>httpcomponents</td>
<td>Java</td>
</tr>
<tr>
<td>accumulo</td>
<td>Java/Python</td>
</tr>
<tr>
<td>kafka</td>
<td>Scala</td>
</tr>
<tr>
<td>hive</td>
<td>Java</td>
</tr>
<tr>
<td>ambra</td>
<td>Java</td>
</tr>
<tr>
<td>jena</td>
<td>Java</td>
</tr>
<tr>
<td>apr</td>
<td>C</td>
</tr>
<tr>
<td>jackrabbit</td>
<td>Java</td>
</tr>
<tr>
<td>sling</td>
<td>Java</td>
</tr>
<tr>
<td>openneb</td>
<td>Java</td>
</tr>
<tr>
<td>mahout</td>
<td>Java</td>
</tr>
<tr>
<td>avro</td>
<td>Java</td>
</tr>
<tr>
<td>npanday</td>
<td>C#</td>
</tr>
<tr>
<td>cassandra</td>
<td>Java</td>
</tr>
<tr>
<td>uima</td>
<td>Java</td>
</tr>
<tr>
<td>roller</td>
<td>Java</td>
</tr>
<tr>
<td>portals</td>
<td>Java</td>
</tr>
<tr>
<td>ode</td>
<td>Java</td>
</tr>
<tr>
<td>buildr</td>
<td>Ruby</td>
</tr>
<tr>
<td>pig</td>
<td>Java</td>
</tr>
<tr>
<td>camel</td>
<td>Scala</td>
</tr>
<tr>
<td>archive</td>
<td>Java</td>
</tr>
<tr>
<td>xminbeans</td>
<td>Java</td>
</tr>
<tr>
<td>spanmasqain</td>
<td>C</td>
</tr>
<tr>
<td>shindig</td>
<td>Java</td>
</tr>
<tr>
<td>mina</td>
<td>Java</td>
</tr>
<tr>
<td>katal</td>
<td>Java</td>
</tr>
<tr>
<td>commons</td>
<td>Java</td>
</tr>
</tbody>
</table>

Total Multiple 486 280 7290890
The cause of flakiness is in lines 37-43. The current thread is waiting for some resources generated by another thread. To achieve that, a sleep is used in line 35, and that causes flakiness. The fix to the problem was to add a busy wait loop that stalls until at least one of the resources needed is available.

The following code is added before line 37.

```java
    Thread.sleep(fs.getConf().getContinuousSchedulingSleepMs() + 500);
```

To make sure that the test does not hang forever, a timeout of 5 seconds is added in the `@Test` annotation.

### B. Concurrency

Some flaky tests are due to data races, atomicity violation, or deadlock. In our study, concurrency was the third most common cause for flaky tests. Examples of flaky tests in this category can be found in Hive starting from revision 1520663 up to revision 1523177. The issue was fixed in revision 1523178, and has an associated bug report (HIVE-5290).

Multiple tests in HCatalog were failing intermittently because of non-determinism in the source code (due to concurrency). Below we show the code snippet from the function `HCatContext.setConf()` that is responsible for the bug. The method iterates over the elements in a map which can be modified by different threads. A ConcurrentModificationException is thrown when that happens.

```java
    for (Map.Entry<String, String> entry : conf) {
        if (!conf.containsKey(entry.getKey())) {
            conf.add(entry.getKey(), entry.getValue());
        }
    }
```

The fix was to add a synchronized block as shown in the code below (from revision 1523178).

```java
    synchronized (conf) {
        for (Map.Entry<String, String> entry : conf) {
            if (!conf.containsKey(entry.getKey())) {
                conf.add(entry.getKey(), entry.getValue());
            }
        }
    }
```

### C. Test order dependency

Some tests might fail intermittently because of the lack of isolation between tests. Dependencies between tests result in unpredictable behaviors when the test order changes.

Many of the tests that we collected in this category were found in Hadoop. An example of such tests can be found in any revision of Hadoop starting from 1390159 (where the flaky tests were introduced) and before 1513368 (where they were fixed). This particular issue that revision 1513368 solved was occurring when running the test class TestDFSIO. In this test class, there are four tests that read (testWrite, testRead, testReadRandom, testReadBackward, testReadSkip) and one that writes(testWrite). The developers wrongly assumed that testWrite would always run first but JUnit does not guarantee any particular test ordering so, sometimes, one (or more) read tests will run before testWrite in which case they will fail.

The developers fixed this issue by adding a call to testWrite in the `@BeforeClass` as shown in line 13 below.

```java
    @BeforeClass
    public static void beforeClass() throws Exception {
        bench = new TestDFSIO();
        bench.setConf().setBoolean(DFSConfigKeys.DFS_SUPPORT_APPEND_KEY, true);
        cluster = new MiniDFSCluster.Builder(bench.getConf()).
            .numDataNodes(2)
            .format(true)
            .build();
        FileSystem fs = cluster.getFileSystem();
        fs.createControlFile(fs, DEFAULT_NR_BYTES, DEFAULT_NR_FILES);
        /** Check write here, as it is required for other tests */
        testWrite();
    }
```

### D. GUI

Some GUI tests can also be flaky. In our study, we have decided to classify them in a separate category. The specific reasons for flakiness in GUI tests vary. So far, we have encountered 7 flaky GUI tests. An example of a GUI flaky test can be found in Tapestry between revisions 890911 and 929531 (it was fixed in 929532). In the class FormTests, `regexp_validator()` makes an assertion of the text in a pop-up window as shown in the code snippet below. The test fails intermittently depending on whether the pop-up window is shown before or after the assertion is executed.
The developers fixed the issue by adding a call to the function `waitForCondition()` as shown in the code below.

```java
waitForCondition()
```

Note that the method `waitForCondition` takes as a second argument a timeout value, which was set to 5 seconds in this example. This means that if the pop-up window fails to show within 5s, the test will fail. This is an example of a fix that only reduces the probability of failure. It does not remove the source of non-determinism.

### E. Network

Network is one common reason for flakiness that is hard to control. Most of today’s software cannot work without communication. They either need to load data from the web, exchange information with a remote server, use an online service, or even have multiple processes on the same machine communicate via sockets. Such activities impose external dependencies on the running test and may cause it to intermittently fail. In that case, the failure does not mean that the code under test is buggy, but rather that the developer does not account for such uncertainties.

From the results of our inspection, we distinguish two types of flaky tests whose root cause is the network. The first type is due to network connection failure that the developers do not account for, and the second type is due to bad socket management. Let’s walk through a couple of examples, one from each type.

#### Network Connection Unreliability Example

Avro is a software for data serialization written in Java.

In its revision 1196217, a flaky test related to network is fixed. This test was flaky since it was written in revision 1136342. There is a bug report AVRO-934 associated with it as well. The tests `TestProtocolNetty`, `TestNettyServer`, and `TestNettyServerWithCallbacks` fail intermittently by hanging and never terminating. The reason of the failure is that all of these tests are trying to access the network with an unbounded wait. To solve the problem, timeouts with different values were added to the tests.

![Fig. 4: Types of fixes](http://avro.apache.org/docs/current/)

In general, there are no default or common timeout values. In practice, it is left to the wisdom of the developer how much wait he/she finds needed. Sometimes, the waiting time is decreased (like in this case) and some other times the waiting time is increased. Hence, there is no rule to be followed on the waiting value.

#### Socket Management Flakiness Example

One interesting example on socket management flakiness was found under APR (Apache Portable Runtime) project, which aims at providing a reliable interface for library developers leveraging platform dependences. They want to relieve programmers from unpredictable (flaky) platform dependent behaviors. Ironically, APR itself is found to have many flaky tests. The language used is C.

In its revision 1531884, the test `client` is failing. Below is the code snippet that shows the reason of the intermittent failure.

```c
while (connect_tries--) {
    rv = apr_socket_connect(sock, destsa);
    if (rv == APR_EACCES) {
        apr_errno = APR_EACCES;
        apr_sleep(connect_retry_interval);
        connect_retry_interval *= 2;
    } else {
        break;
    }
}
if (rv != APR_SUCCESS) { aprerr("apr_socket_connect()"); rv; }
```

As we can see, there is a while loop in which the client keeps trying to establish a connection to a socket on the server on the same machine in order to exchange a file. If the connection fails (lines 3 to 5), the client retries after a certain interval to reconnect to the same server’s socket again. This is actually the main problem. We are trying to connect to the same socket which might be busy sometimes and hence causes failure.

To fix the problem, we need simply to close the socket in case of failure to connect, and retry connecting to another one.

#### F. Randomness

Another reason for tests to intermittently fail is randomness. Out of the 81 commits that we thoroughly examined, 2 examples of randomness were encountered in the study.

The first one was found in Hbase. The corresponding revision number is r1181450 and the associated bug report ID is HBASE-2884. The flailing test is `TestHFileOutputFormat`. The tester wants to write into files. To make sure that unique
file names are generated, the function `public static long currentTimeMillis()` from the class `java.lang.System` is called. This function returns the current time in milliseconds, which is the difference between the current time and the time January 1, 1970 (UTC). However, the precision of the value returned depends on the operating system.

What happens is that two of the executing threads call the function `currentTimeMillis()` in the same second. The problem is that the millisecond precision of time returned by the system is too small to detect the time difference between the two threads and the function `currentTimeMillis()` returns the same value in both cases. Hence, the two threads write to the same file and we end up with one file instead of two.

The fix of the problem was to make the last byte of the file name (which is the current time in milliseconds) equal to the unique task ID associated with the thread. Hence, we are sure that file names will not overlap again.

On a more general note, we notice that the operating system is also one of the causes of flakiness. Such a test can never fail on some machines, while failing intermittently on other machines.

The second one was found in Hadoop. The corresponding revision number is r1399093 and the associated bug report ID is HDFS-4055. The failing test is `TestAuditLogs`. Let's take a look at the code of the method that fails.

```java
@Test
public void testAuditAllowed() throws Exception {
    final Path file = new Path(fnames[0]);
    FileSystem users = DFSUtil.getFileSystemAs(userGroupInfo, conf);
    setupAuditLogs();
    InputStream istream = users.open(file);
    int val = istream.read();
    verifyAuditLogs(val > 0);
}
```

This method opens a file, and reads it. To make sure that the read was successful, an assertion at the end checks that the first byte read is greater than zero. To understand the reason of flakiness, we examine the `@Before` annotation and locate the function `createFiles` that generated the file opened. Here is the code snippet from that function that shows how the bytes in the file are generated.

```java
if (filename > 0) {
    byte[] toWrite = new byte[bufferLen];
    Random rb = new Random(seed);
    long bytesToWrite = fileLen;
    while (bytesToWrite > 0) {
        rb.nextBytes(toWrite);
        int bytesToWriteNext = (bufferLen < bytesToWrite ? bufferLen : (int) bytesToWrite);
        out.write(toWrite, 0, bytesToWriteNext);
        bytesToWrite -= bytesToWriteNext;
    }
}
```

At line 3, we can see that the first byte is generated using the class `java.util.Random`. The number generated has a probability 1/256 to be zero which was not accounted for in the test assertion val > 0 and hence, the test fails intermittently with probability of failure of 1/256. The fix of the problem was to modify the assertion to be `val >= 0`.

### G. Resource leak

When a system suffers from resource leaks, tests can intermittently fail (i.e., the test that causes the resource to go over its limit will fail). By resource leak, we refer to any resource that the application has to manage by acquiring and releasing (e.g., memory and database connections). We have observed 4 flaky tests that are caused by resource leaks. One of them is in Sling from revision 1440446 to revision 1501672 (fixed in 1501673). Some tests in ErrorHandlingTest.java were failing intermittently with the error status code 503 which means that the server is overloaded. The server in question is Jenkins and overloading was due to the repeated delete node operations initiated by the tearDown() method.

The fix was to make the tearDown() method delete specific fields within a node rather than the whole load. That was successful in alleviating the load on the Jenkins Server.

### H. Unordered Collections

In general, when iterating over unordered collections (e.g., sets), developers should not assume a particular order. When they do, the test outcome becomes non-deterministic. One instance was found in Hadoop. The corresponding revision number is r1005581 and the associated bug report ID is HDFS-6933. The failing test is `TestListFiles`. This is one of the tests for Apache File system `org.apache.hadoop.fs.FileSystem`. Here is the code snippet that is causing flakiness.

```java
@Test
public void testAuditAllowed() throws Exception {
    itor = fs.listFiles(TEST_DIR, true);
    stat = itor.next();
    assertTrue(stat.isFile());
    assertEquals(fs.makeQualified(FILE2), stat.getPath());
    stat = itor.next();
    assertTrue(stat.isFile());
    assertEquals(fs.makeQualified(FILE3), stat.getPath());
    stat = itor.next();
    assertTrue(stat.isFile());
    assertEquals(fs.makeQualified(FILE1), stat.getPath());
    assertFalse(itor.hasNext());
}
```

The variable `itor` is an iterator of type `RemoteIterator <LocatedFileStatus>`. The developer wrote the assertions assuming that the files are stored in the order FILE1, FILE2, FILE3. So whenever he increments the iterator `itor`, it will point to the next file in that order, which is not the case.

To patch the flakiness, a HashSet is used to save the file paths of FILE1, FILE2, and FILE3. Then, the assertion would be changed to `assertThatPath("Path “ + stat.getPath() + “ unexpected”, filesToFind.remove(stat.getPath()));` where the developers check that the path pointed by the iterator exists and then we remove it from the HashSet.

### I. IO

Some tests may intermittently fail because of issues with I/O operations. We have studied one such test in the project Archiva. RepositoryServletProxiedRelocatedTest is flaky between revisions 649352 and 718156 (fixed in 718157). This flaky test has a bug report associated with it (MRM-1004).
The flakiness of that test was caused by a fault in the MavenXpp3Reader class. The class does not close the reader after reading the content of a POM file. So, the file stays open until the instance of MavenXpp3Reader is garbage collected. The test fails whenever it tries to open the POM file before the previous instance of MavenXpp3Reader has been garbage collected. The fix (or rather work-around since the bug is in a library) was to close the file reader in the Archiva code after each call to the read method of the MavenXpp3Reader. Below is the buggy code in ArchivaDavResourceFactory.java (in revision 718156):

```java
Model model = new MavenXpp3Reader().read( new FileReader( pom ) );
```

In revision 71857, the code was replaced with the following:

```java
// MavenXpp3Reader leaves the file open, so we need to close it ourselves.
FileReader reader = new FileReader( pom );
Model model = new MavenXpp3Reader().read( reader );
reader.close();
```

### J. Floating Point Operations

Dealing with floating point operations is known to be very tricky [2], [3], [8], [32]. A simple operation like calculating the average of two numbers requires thorough coding to avoid overflows, underflows, etc... [3]. Such problems can be a reason for test flakiness. In our study, we encountered a case in the Shindig project where the test oracle asserts that a floating point result is equal to 566. Due to floating operations being different on different machines, rounding can yield sometimes to 567 causing intermittent failures. The fix was simply to relax the assertion to accept both numbers.

### K. Time

Relying on the system time introduces many non-deterministic failures due to precision. The precision in which time is reported differs from a platform to another and that is mostly neglected by developers/testers. We only encountered a couple of instances of this type, one of which is shown in the randomness section (first example of Hbase). For space constraints, we will not go through the second one as it is very similar to the one shown in the randomness section.

## IV. Remedies for Flakiness

### A. Asynchronous Wait

Non-determinism due to asynchronous wait occurs when one thread needs the results of a certain service (thread, remote server, configuration, ...) and does not wait properly until those results are available. In general, to fix this problem, proper waiting/synchronization needs to be introduced into the test code. The source code is modified in very rare cases.

### Recommendations

We recommend synchronizing the test code by waiting for the needed conditions to be achieved. Note that thread sleeps are not a good practice. When using a sleep, the developer is expecting an upper bound on the wait time for a service to be available based on his own experience. However, this does not take into account platform dependence (different machines have different latencies), network congestion, server overloading, and many other uncertainties that are hard to estimate.

Waits are better than sleeps also in the opposite scenario, when the service is actually available earlier than expected. In the case of sleep, we will have to delay execution anyway for the specified amount of time. In order to prevent the test from hanging on the wait condition indefinitely, we set a an upper-bound for the test using a timeout in the @Test annotation.

Other common fixes Based on our study, we can summarize the popular fixes employed by developers for asynchronouse wait in three categories: adding a sleep, increasing the time of an existing sleep, and replacing a sleep with a wait for a condition (discussed in the recommendations above).

### B. Concurrency

Concurrency is a common and well known cause for non-determinism of test outcome. Data races, atomicity violation, and deadlock can all lead to tests intermittently failing. There are various ways to fix flaky tests in this category. Some of the tests we studied were fixed by changing the test code, others were fixed by changing the source code.

Other common fixes Based on our study findings, we can say that the common fixes for this category of flaky tests are either: adding synchronization, or removing the dependency between threads.

### C. Test Order Dependency

Some tests can become flaky because they depend on another test. These kinds of flaky tests are some of the hardest to detect. It is not obvious which other test(s) are interdependent with the intermittently failing test. Figure 5 shows the time that was needed to fix each of the flaky tests that we examined. The average time needed to fix flaky tests due to test dependencies is 339 days.

Other common fixes As shown in Figure 4, most of flaky tests of this kind can be fixed by changing the test code only. The most common fix for this type of flaky tests is add/modify the setUp() and tearDown() methods of the test class.
D. GUI

GUI tests can also be flaky. Determining the precise cause of their flakiness can be quite hard as it can be related to platform-dependence, asynchronous behavior, or libraries that are too sensitive to slight UI structure changes. From the data shown in figure 5, the average time that was needed to fix one of the GUI tests we encountered is 208 days. Figure 4 shows that the fixes were mostly restricted to the test code.

Recommendations Because the specific fixes of flaky GUI tests are so varied, it is hard to provide a general guideline for dealing with them. One thing to keep in mind however, is that some attempts to fix GUI flaky tests (like in the example we presented) may (significantly) reduce the probability of failure but do not make the test-outcome deterministic.

Other common fixes Common fixes for flaky GUI tests include: adding a wait, using a more robust UI element locator (css), and adding a check for the current browser.

E. 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. Therefore fixing flaky tests related to network can affect the source code only, the test code only, or both. It depends on where the network communication is used.

Recommendations Our recommendation is to alleviate this external dependency whenever possible and use mocks. [11]. A mock would behave exactly like the remote service and give us a reliable feedback. However, there is an issue of practicality. It is not an easy task to create mocks that mimic exactly the responses of the original server in all cases. Hence, it depends on the number of tests that communicate with the network and their importance whether to adapt the mock solution or simply add some wait/sleep statements.

Other common fixes When it comes to network connection failures, a quick fix would be to add/modify a timeout that waits for the connection establishment. This is typically useful in the case of slow networks. As for the socket management, it is very important to be careful with their usage: a check whether the socket is available or not, and properly set it or release it. Sometimes the API socket library does not take care of these cases and they have to be manually coded.

F. Randomness

Flakiness due to randomness is associated with random number generation. In the examples we encountered, random number generation was related to system time and concurrency but that cannot be generalized. The fixed encountered were in the test, but that is also not generalizable. It depends where the random number generator is used: if it was used in the source code, then the source code will have to be changed.

Recommendations The fixes of randomness can be summarized in two main recommendations. First the seed of the function that generates random numbers should be chosen wisely. Second, whenever dealing with random number generation, do not forget about the boundary cases like zero. The operating system’s uncertainties are also a concern that can be alleviated by not depending on the functions that OS provides (time for example).

G. Resource Leak

Flaky tests that are caused by Resource leaks can be particularly hard to detect as the failure will occur in the test that happened to pass the threshold of that resource. In the tests that we encountered, most of the time, changing the test code was enough to fix the flakiness. The fixes that we encountered were usually either decreasing the demand on the resource, or increasing the availability of resource. These two fixes may not always be possible.

Recommendations When it is not possible to fix flaky tests of this nature by decreasing the demand on the resource or increasing its availability, it might still be useful to at least have some control on its manifestation. Fowler suggests to manage the relevant resources through resource pools. By reducing the pool size, the probability of failure can be increased. [11]

H. Unordered Collections

Flakiness due to unordered collections rises whenever the developer assumes that the API guarantees a certain order that the latter does not.

Recommendations In general, a good programming practice is to write tests that do not assume any specific ordering on collections unless an explicit convention is known to be enforced on the used data structure.

I. IO

Because they deal with external resources, I/O operations can cause intermittent test failures. In this study, we have only encountered one flaky test in this category so it is hard to generalize any findings. For the particular test we studied, the fix was in the source code. This might not be the case for all flaky tests caused by I/O. For example, some tests might be flaky because they depend on an external file. To fix those, the test code could be changed to mock the file system.

Recommendations As a general rule, file readers should always be closed after being used. The same goes for database connections and other types of I/O. We also recommend to mock external resources (such as files) as much as possible to avoid intermittent failures related to I/O operations.

J. Floating Point Operations

Floating point operations are non-deterministic by nature, and can cause a lot of problems if not handled correctly. The fixes to these problems can be either in the test or in the source code, depending on where the floating operation is encoded.

Recommendations In general, one has to be extremely careful when dealing with floating point operations. Imprecision, like the case we encountered, is not avoidable. We would like to have our assertions as independent as possible from floating point results.
Other common fixes A certain threshold for marginal error can be used if we decided to use floating point results in assertions.

K. Time

Relying on the system’s time is not a good practice, as the precision differs from one system to another. This issue becomes more critical when the time is correlated with concurrency and random number generation with seed taken from the system time.

More generally, developers are advised to avoid platform dependence.

V. DISCUSSION

In addition to answering the research questions, we would like to highlight other points worth discussing.

More flaky tests are out there. One important thing to point out is that our study has found thousands of flaky tests simply by searching for a couple of keywords. The novelty of our approach starting from the commit log has enabled us to catch many flaky tests that were not reported in bug report databases (this confirms what we will discuss next about flaky tests being hard to deal with). However, we are confident that a lot of flaky tests still exist hidden under a different keyword or simply unreported. This fact increases the importance of having a tool that automatically detects flaky tests and warns developers about them. To show even more the commonality of flaky tests, we briefly describe a few flaky tests encountered by students working on a project for their software engineering project. The students were asked to implement a simple library system and to write unit tests for it. Four student groups inadvertently wrote flaky tests. Three were due to test order dependencies. The students wrongly assumed that the test ordering is fixed. The fourth flaky test was caused by an unordered collection. The students wrongly assumed that the order of elements in a set was fixed.

Effectiveness of fixes. Fixing flaky tests sometimes addresses symptoms rather than the root cause. In that case, the patch would simply decrease the probability of a test failing but does not eliminate that possibility. For example in the network category, many fixes for hanging connections were to throw an exception after a certain timeout. This is a fix for the symptom (the test hanging) but not to the root cause (uncertainties due to dependence on the network). Another example is shown in section III.D about GUI.

Flaky is tricky. Dealing with flaky tests is not a straightforward thing to do. Even if the reason for flakiness is known, reproducing the test failure is hard. We have tried ourselves to reproduce flaky tests. We required having the buggy version, triggering the flaky execution, patching the buggy version, and running the scenario again expecting success. We faced a lot of challenges: dealing with open source code is not easy, compiling different versions of the code had different dependencies that are not met, patches provided do not always fix the problem, triggering flakiness is sometimes very tricky to obtain (especially in the case of platform dependence), etc. Some tests took us more than 10 hours (each) to be able to reproduce them.

Flaky test evolution. Figure 5 shows the time it took to fix each flaky test we analyzed. On average, to fix a flaky test, it takes 331.32 days for asynchronous wait, 338.72 days for concurrency, and 339.30 for test order dependencies. Although, the average time needed varies a lot across projects and between different flaky tests within the same projects, these statistics can provide an idea about the relative complexity of detecting and fixing flaky tests of each type.

Platform Dependence. One final point we want to make is about platform dependence which is a common issue across the different root causes of flakiness. Programmers need to pay a lot of care not to have assumptions about the running machine in terms of speed, functions provided by the system, precision of operations (time and floating point precisions for example), etc.

<table>
<thead>
<tr>
<th>Category</th>
<th>Time (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Async wait</td>
<td>313.32</td>
</tr>
<tr>
<td>Concurrency</td>
<td>338.72</td>
</tr>
<tr>
<td>Test order dependency</td>
<td>339.30</td>
</tr>
<tr>
<td>GUI</td>
<td>208.33</td>
</tr>
<tr>
<td>Network</td>
<td>223.25</td>
</tr>
<tr>
<td>Randomness</td>
<td>91.00</td>
</tr>
<tr>
<td>Resource leak</td>
<td>166.25</td>
</tr>
<tr>
<td>Unordered collections</td>
<td>44.00</td>
</tr>
<tr>
<td>IO</td>
<td>212.00</td>
</tr>
<tr>
<td>Floating Point operations</td>
<td>206.50</td>
</tr>
</tbody>
</table>

Fig. 5: Time needed to fix flaky test

VI. THREATS TO VALIDITY

The validity of our study and its results can be questionable regarding the following considerations:

- Keywords used. While searching the commit log, we greppe for a couple of keywords "flak" and "intermit". There is no guarantee on the recall of the results. We are sure that there are still flaky tests that our search did not capture. These tests can be hidden under words like "concurrency", "race", "stall", "fail", etc. In our future work we intend to search for such words and characterize the results having a better measure of recall.

- Choice of projects. In our study, we target ALL the projects under Apache Software Foundation. A threat to validity can be that these findings are not generalizable to other projects. We believe that our findings are general and not specific to Apache projects. Apache is just a license that does not imply any constraints on the projects. We examine mature projects with more than 1.5 million commits, and 7 billions LOC. The projects cover different languages (java, python, perl, ruby, c, c++, c#, php, sql, xml, javascript, etc) and different types of applications (web server, databases, cloud, graphics, mail, build-management, etc).

- Manual inspection. Labeling and characterizing commits manually can be a threat to the validity of the study. To minimize the probability of error, every commit is
inspected by two authors independently, and then the results are merged.

VII. RELATED WORK

Several researchers and practitioners have pointed out that non-deterministic tests are a big issue [11, 12, 20, 21, 27, 33]. The problem is especially exacerbated in the context of automated regression testing. Fowler [11] describes non-deterministic test outcomes as a recurring problem that his team runs into during automatic regression testing. He also explains the problems that arise from having flaky tests within a regression test suite. Lacoste [20] also describes some of the unfortunate side-effects of having flaky tests in an automated regression testing system: Some branches may miss a release because of an intermittent failure and the test suite has to be rerun after a flaky test is discovered and fixed.

Fowler outlines some ways in which flaky tests can be dealt with in practice [11]. He presents possible fixes for flaky tests that are due to lack of isolation, asynchronous behavior, remote services, time, and resource leaks. The first (temporary) way he suggests for dealing with flaky tests is quarantine. Fowler stresses that this should only be a temporary measure. The quarantined tests should be fixed as soon as possible in order to get back their fault-finding ability.

To avoid flaky tests caused by lack of isolation, Fowler recommends keeping tests isolated by either always rebuilding the starting state or by cleaning up the state after each test. For tests that fail intermittently because of asynchronous behavior, he suggests to avoid fixing them by adding sleep statements. He provides two justifications for that:

- The sleep time needs to be set to a large enough value to avoid intermittent failures which leads to a slower running test suite.
- No matter how large the sleep time is, there will still be a possibility for the test to fail (i.e., adding a sleep statement only reduces the probability of the test failing but it does not make its outcome deterministic).

The alternatives that he suggests for fixing flaky tests of this kind include:

- Providing a callback in the asynchronous service: This insures that the test only waits the needed amount of time.
- Polling: The advantage of this method is that the polling interval can be set to a small value and that will be the maximum amount of time that will be "wasted" while waiting for a response.

Another cause of flaky tests that Fowler considered is remote services. From tests that intermittently fail because of remote services, he recommends using of mocks (which he refers to as Test Doubles). As for fixing flaky tests that are caused by time (direct calls to the system clock), he suggests to wrap the system clock with routines that will allow for seeding a particular value during testing. The last category that Fowler addressed is flaky tests caused by resource leaks. For those, he suggests to manage the relevant resources through resource pools. By reducing the pool size, the probability of failure is increased and therefore faults can be detected early and corrected.

Aside from Fowler's work presented previously, we are not aware of any other published work that tries to study flaky tests in general. However, an extensive work has been done in studying some particular categories of flaky test causes. These studies do not necessarily mention test outcomes but they address a particular issue that we have identified as a possible cause for test outcome non-determinism.

A. Test Order Dependency

Wuttke et al. published a technical report that addresses the issue of test dependence. They provide a formalization of test dependence, prove that finding manifest dependencies is NP-complete, and present examples from real-world test suites where manifesting test dependency leads to exposing some faults within the systems under test. [11]

Muslu et al. found that isolating unit tests can be helpful in detecting faults but that enforcing it can be computationally expensive. [26]

Some researchers have also started to work on trying to remove or at least mitigate the effect of these test dependencies. Bell and Keiser propose an approach for isolating unit tests in Java applications by analyzing them to identify all potential side-effect causing code (in memory) and instrumenting the code to reinitialize the necessary parts. [4]

B. Concurrency

Lu et al. published a comprehensive characteristic study of concurrency bugs. [23]. Their study examines bug patterns, manifestation, and fixes of concurrency bugs. One of the study findings that they have listed is that reliable test cases that can reproduce a concurrency bug are very important for diagnosis. They gave an example of a bug report in Mozilla (73291) which was simply closed because the failure could not be reproduced. In this particular case, there was no test case to begin with, but even if there was, the other complication is to make it trigger the bug (i.e., fail) reliably.

VIII. CONCLUSION AND FUTURE WORK

This paper presents the first comprehensive study about test outcome non-determinism. With our novel methodology of starting with the commit log messages rather than the bug reports, we were able to identify 1129 commits related to flaky tests out of which 486 deal with distinct tests. Note that 61% of the 486 commits do not have an associated bug report. We identify 11 root causes of non-deterministic test failure and illustrate them with examples from open source. For each root cause, we give useful recommendations on how to avoid/remedy the flakiness. Finally, we discuss other issues and conclusions about flaky tests.

In the future, we plan to continue working on the study collecting more data. We also plan to develop tools for detecting and fixing flaky tests.
REFERENCES


