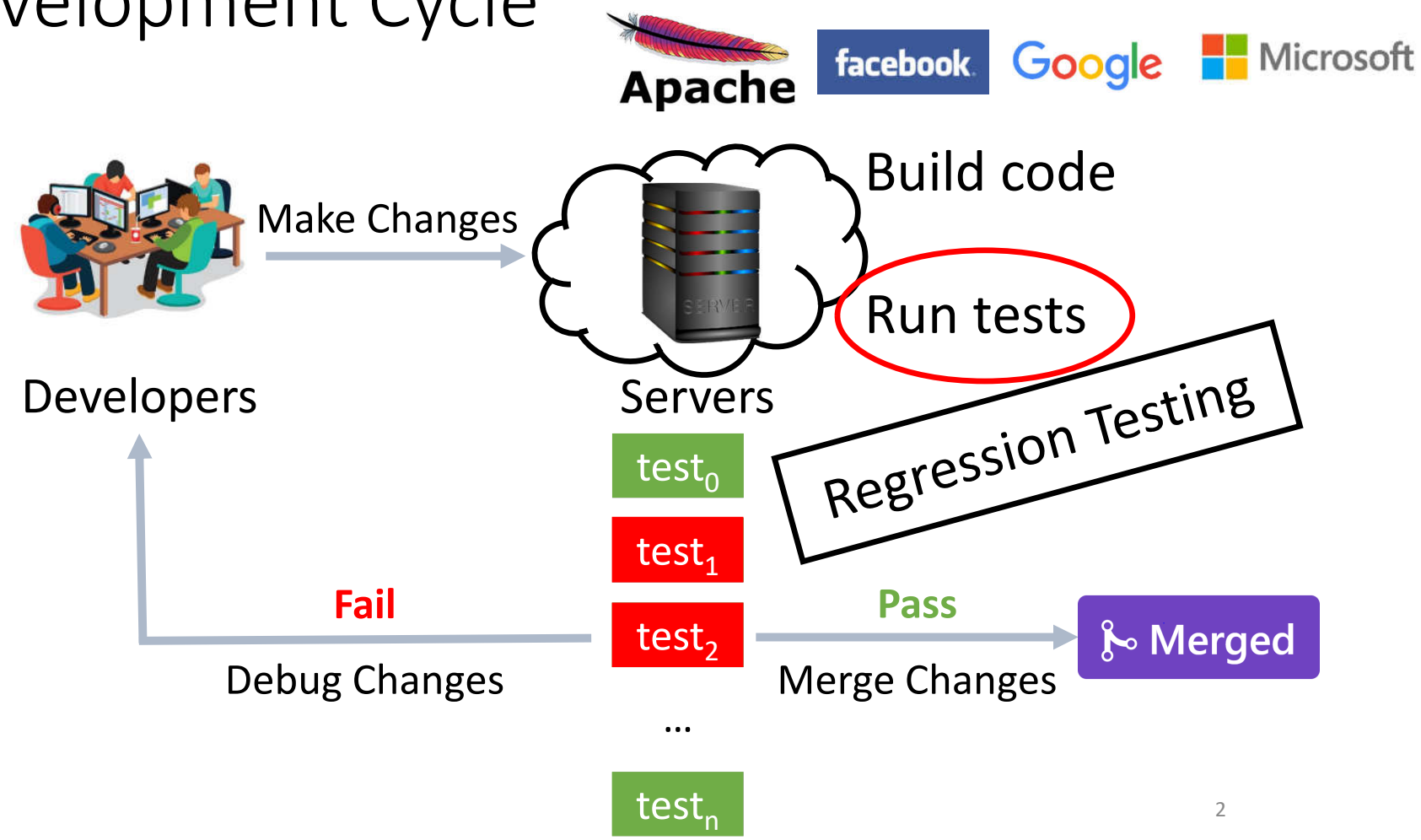
# CS 5154
# Flaky Tests

Spring 2021

Guest Lecture by
August Shi, Assistant Prof. at UT Austin

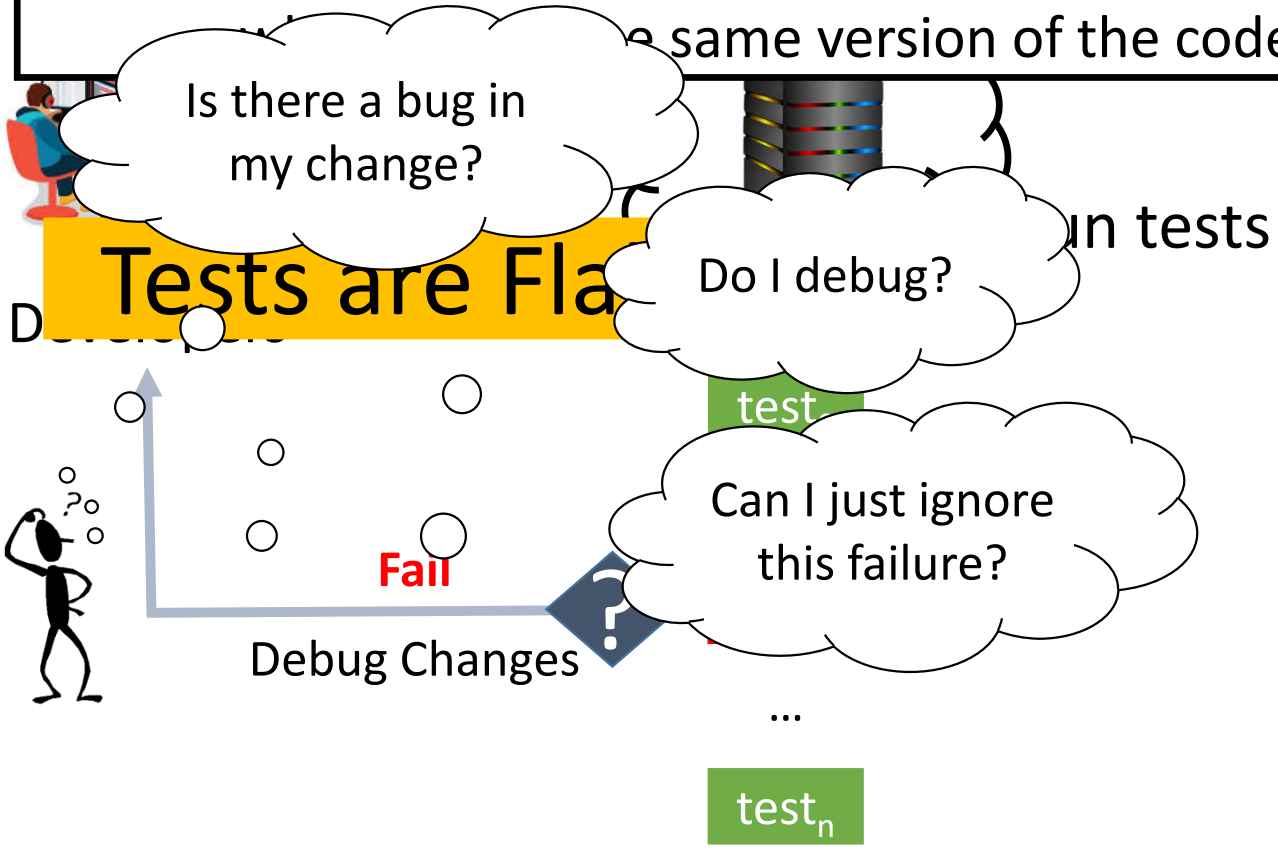
# Development Cycle



# Are Tests Reliable?

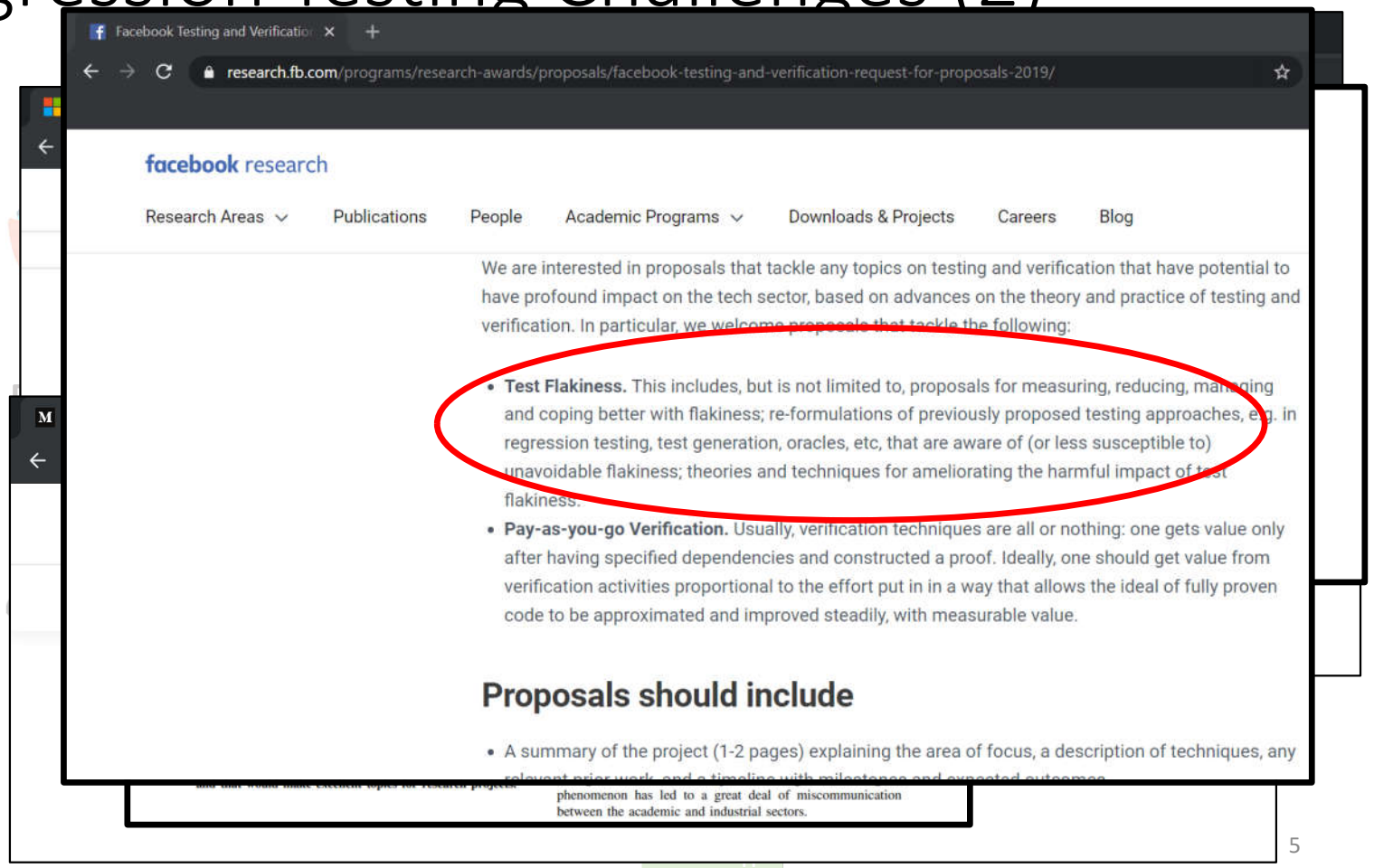
**Flaky Test**: a test that can non-deterministically pass or fail

same version of the code



# Flaky Tests Fundamentally **Break** Regression Testing!

# Regression Testing Challenges (2)



# An Empirical Analysis of Flaky Tests

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, Darko Marinov



**Slides adapted from those by Farah Hariri**

# Flaky Tests: Why do we care?

- Flaky tests undermine the value of the test suite
  - Flaky failures are hard to be reproduced due to non-determinism
  - Flaky failures are hard to debug especially when not affected by recent modifications
  - Flaky failures may hide real bug
- Commonality of flaky tests in large code bases
  - Example: TAP system at Google has 1.6M test failures in the last 15 months, out of which 73K (4.56%) are failures causes by flaky tests

# Flaky Tests: Why do we care?

- Current solutions are unsatisfactory
  - Most common solution is rerunning the failed test multiple times, if it passes once we declare it passing
- Google for example reruns flaky tests for 10 times
  - Android annotation *@FlakyTest* and JUnit annotation *@Repeat* used to mark tests for rerunning multiple times before declaring its failure

# This Study

- First empirical study on flaky tests within open-source projects
- Goals
  - Characterize the root causes of flaky tests
  - Study the possible ways of manifesting flaky failures
  - Identify the common fixing strategies and provide actionable information for avoiding, detecting, and fixing flaky tests

# Methodology

- Extract the entire SVN repository of Apache Software Foundation
  - Focus on commits likely to have fixed flaky tests
    => larger dataset of **fixed** cases than from bug-report databases
- Grep for the keywords “flak” and “intermit”
  - 1129 commits
- Manual inspection of commits
  - **Filtering phase**: Is the commit actually about a flaky test? Distinct flaky test?
    => 855 commits about flaky tests, **486** about fixing distinct flaky tests
  - **Analysis phase**: What are the reasons for flaky tests?
    => Inspected **201** commits

# Analysis Findings

- 10 main root causes of flakiness

| **Root Cause** |
| --- |
| Async Wait |
| Concurrency |
| Test-Order Dependency |
| Resource Leak |
| Network |
| Time |
| I/O |
| Randomness |
| Floating-Point Operations |
| Unordered Collections |
| Unknown |

# Async Wait

- The largest category – 46%
- Def: Test makes an asynchronous call and does not properly wait for the result of the call to become available before using it
- Example:

```
@Test
public void testRsReportsWrongServerName() throws Exception {
  MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
  MiniHBaseClusterRegionServer firstServer =
    (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
  HServerInfo hsi = firstServer.getServerInfo();
  firstServer.setHServerInfo(...);
  // Sleep while the region server pings back
  Thread.sleep(2000);
  assertTrue(firstServer.isOnline());
  ... // similarly for secondServer
}
```

# Async Wait - Manifestation

- Manifestation Strategies:
  - 34% of Async Wait flaky tests use `sleep` or `waitFor` with time delay
    => manifestation can be as simple as changing the time delay value
  - 85% of Async Wait flaky tests do not “wait” on external resources and involve only one ordering (wait on only one other thread/process)
- Implication: Most Async Wait tests can be detected by adding **one** time delay

# Async Wait - Fixing

- Common Fixes:
  - Change to use `waitFor` calls (58%)
  - Modify `sleep` call parameters (26%)
  - Reordering code (3%)
  - Other (14%)
- Example:

```
@Test(timeout=180000)
public void testRsReportsWrongServerName() throws Exception {
  MiniHBaseCluster cluster = TEST_UTIL.getHBaseCluster();
  MiniHBaseClusterRegionServer firstServer =
    (MiniHBaseClusterRegionServer)cluster.getRegionServer(0);
  HServerInfo hsi = firstServer.getServerInfo();
  firstServer.setHServerInfo(...);
  // Sleep while the region server pings back
  cluster.waitOnRegionServer(0);
  assertTrue(firstServer.isOnline());
  ... // similarly for secondServer
}
```

# Async Wait - Fixing

- Effectiveness:
  - `waitFor`: Most effective; explicitly states the necessary condition before proceeding
    More efficient when the result is available earlier
  - `sleep`: Inefficient, counterintuitive (hard to infer desired ordering)
    No guarantees (overestimation is inefficient and underestimation leads to failure)
  - Reordering code: similar to `sleep` except a bit more efficient
- Implication:
  - For developers: Explicitly express dependencies using `waitFor` to synchronize code
  - For researchers: Techniques could automatically insert order enforcing methods such as `waitFor` to fix the code

# Concurrency

- The second largest category – 20%
- Def: Test non-determinism is due to undesirable interactions between different threads (not asynchronous calls)
  - E.g., data races, atomicity violations, deadlocks
  - Note: Non-determinism can be in the code under test or in the test code itself
- Example:

```
if (conf != newConf) {
  for (Map.Entry<String, String> entry : conf) {
    if ((entry.getKey().matches("hcat.*")) &&
        (newConf.get(entry.getKey()) == null)) {
      newConf.set(entry.getKey(), entry.getValue());
    }
  }
  conf = newConf;
}
```

# Concurrency - Manifestation

- Manifestation Strategies:
  - Almost all flaky tests involve only two threads or their failures can be simplified to only two threads
  - 97% of their failures due to concurrent accesses only on in-memory objects
- Implication: Existing techniques of increasing context switch portability could help in the manifestation of most concurrency flaky tests

# Concurrency - Fixing

- Common Fixes:
  - Adding locks (31%)
  - Making code determinisitic (25%)
  - Changing concurrency guard conditions (9%)
  - Changing assertions (9%)
  - Other (26%)
- Example:

```
if (conf != newConf) {
  for (Map.Entry<String, String> entry : conf) {
    synchronized (this) {
      if ((entry.getKey().matches("hcat.*")) &&
          (newConf.get(entry.getKey()) == null)) {
        newConf.set(entry.getKey(), entry.getValue());
      }
    }
  }
  conf = newConf;
}
```

# Concurrency

- Effectiveness: As long as the root cause is correctly identified and addressed, all the fixes completely remove the flakiness
- Implication: No common strategy – developers have to carefully inspect the root cause and address it

# Test-Order Dependency

- The third largest category – 12%
- Def: Test outcome depends on the order in which the tests are run
  - In principle, all tests should be independent of one another; in practice however, it is not the case
- Example:

```
@BeforeClass
public static void beforeClass() throws Exception {
  bench = new TestDFSIO();
  ...
  cluster = new MiniDFSCluster.Builder(...).build();
  FileSystem fs = cluster.getFileSystem();
  bench.createControlFile(fs, ...);
}
public void testWrite() {...} /* Writes data needed for other tests*/
```

# Test-Order Dependency - Manifestation

- Manifestation Strategies:
  - 16% due to static field in test
  - 32% due to static field in code under test
  - 52% due to external dependency
  - Almost half of test order dependency cases depend on external resources, while the other half depends on the internal state
- Implications:
  - We need techniques that record and compare internal memory states
  - We need sophisticated techniques that model the external environment and explicitly rerun in different orders for some cases

# Test-Order Dependency - Fixing

- Common Fixes:
  - Setting up/cleaning up states (74%)
  - Removing dependency (16%)
  - Making local copies of shared variables merging tests (10%)
- Example:

```
@BeforeClass
public static void beforeClass() throws Exception {
  bench = new TestDFSIO();
  ...
  cluster = new MiniDFSCluster.Builder(...).build();
  FileSystem fs = cluster.getFileSystem();
  bench.createControlFile(fs, ...);
  /* Check write here, as it is required for other tests */
  testWrite();
}
```

# Test-Order Dependency - Fixing

- Effectiveness:
  - All of them completely remove the flakiness
    - Third one (copying code) is rather a workaround and not a fix
- Implications:
  - For developers: Identify and clean shared state before and after test runs
  - For researchers: Automated techniques can record the shared state before and after the execution to automatically generate setup/clean up methods and fix flaky tests due to test-order dependency

# Other Cases

- Resource leak: Application does not manage resources properly
  - E.g., memory allocations, database connections
  - Fixes: Resource pool
- Network: Test fails due to network uncertainties
  - E.g., remote connection failure, bad local socket management
  - Fixes: Use mocks, `waitFor`
- Time: Test depends on system time
  - E.g., fail when midnight passes, differences in time reporting across platforms
  - Fixes: Don’t rely on time…
- I/O: I/O operation failures lead to test failures
  - E.g., not closing files, file not set up
  - Fixes: Close resources, synchronization

# Other Cases Cont'd

- Randomness: Depending on random numbers
  - E.g., using random number generator
  - Fixes: Control seed, handle boundary conditions
- Floating-point operations: Nondeterminism due to computations
  - E.g., overflows/underflows, non-associative addition, platform differences
  - Fixes: make test assertions independent from floating-point results
- Unordered collections: Assuming deterministic ordering of collection
  - E.g., iterating over HashSet, using JSON
  - Fixes: Program to specification, not implementation

# Fixing: Why not remove flaky tests?

- While most of the cases (74%) of flaky tests are fixed by modifying the test, 24% of the fixes modify the code, out of which 94% are fixes to bugs in the code
- Therefore, flaky tests should not be simply removed or disabled because they can help uncover bugs in the code under test

# Conclusions

- Regression testing is important but can be greatly undermined by the presence of flaky tests
- Conducted study of a large number of fixes to flaky tests to understand the common root causes, and describe common strategies that developers use to fix and reproduce 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

# Discussion

- Should flaky tests even be run when considering changes to merge into a repository?
- Why do developers make these mistakes and what measures can we implement to prevent writing flaky tests?
- Could we get a more precise data-set if we use NLP techniques to analyze the commit messages or bug reports to determine if a bug is actually a flaky tests?
- Is it always wrong to use `sleep` calls instead of `waitFor` calls? In which cases would `sleep` be a better option than `waitFor`?
- When a test fails, how should we categorize it? Is it flaky? What kind of flaky test?