DeFlaker: Automatically Detecting Flaky Tests

Jonathan Bell[1], Owolabi Legunsen[2], Michael Hilton[3],
Lamyaa Eloussi[2], Tifany Yung[2], and Darko Marinov[2]
[1]George Mason University, Fairfax, VA, USA
[2]University of Illinois at Urbana-Champaign, Urbana, IL, USA
[3]Carnegie Mellon University, Pittsburgh, PA, USA
bellj@gmu.edu,{legunse2,eloussi2,yung4,marinov}@illinois.edu,mhilton@cmu.edu

ABSTRACT

Developers often run tests to check that their latest changes to a code repository did not break any previously working functionality. Ideally, any new test failures would indicate regressions caused by the latest changes. However, some test failures may not be due to the latest changes but due to non-determinism in the tests, popularly called *flaky* tests. The typical way to detect flaky tests is to rerun failing tests repeatedly. Unfortunately, rerunning failing tests can be costly and can slow down the development cycle.

We present the first extensive evaluation of rerunning failing tests and propose a new technique, called DeFlaker, that detects if a test failure is due to a flaky test without rerunning and with very low runtime overhead. DeFlaker monitors the coverage of latest code changes and marks as flaky any newly failing test that did not execute any of the changes. We deployed DeFlaker live, in the build process of 96 Java projects on TravisCI, and found 87 previously unknown flaky tests in the 10 of these projects. We also ran experiments on project histories, where DeFlaker detected 1,874 flaky tests from 4,846 failures, with a low false alarm rate (1.5%). DeFlaker had a higher recall (95.5% vs. 23%) of confirmed flaky tests than Maven's default flaky test detector.

CCS CONCEPTS

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

KEYWORDS

Software testing, flaky tests, code coverage

ACM Reference Format:
Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 12 pages.
https://doi.org/10.1145/3180155.3180164

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180164

1 INTRODUCTION

Automated regression testing is widely used in modern software development. Whenever a developer pushes some changes to a repository, tests are run to check whether the changes broke some functionality. Ideally, every new test failure would be due to the latest changes that the developer made and the developer could focus on debugging these failures. Unfortunately, some failures are not due to the latest changes but due to *flaky tests*. As in previous work, we define a flaky test as a test that can *non-deterministically pass or fail when run on the same version of the code.*

Flaky tests are frequent in most large software, and create problems in development, as described by many researchers and practitioners [1, 25, 26, 36, 37, 43, 44, 48, 51, 54, 56–58, 61, 62, 65, 76, 80, 82, 85]. For example, according to Herzig and Nagappan [48], the Microsoft's Windows and Dynamics product teams estimate their proportion of flaky test failures to be approximately 5%. Similarly, Pivotal developers estimate that half of their build failures involve flaky tests [49], Labuschagne et al. [56] reported that 13% of builds in a TravisCI dataset failed because of flaky tests, and Luo et al. [61] reported that flaky tests accounted for 73K of the 1.6M (4.56%) *daily* test failures in the Google TAP system for regression testing.

When a test fails, developers need automated techniques that can help determine whether the failure is due to a flaky test or to a recently introduced regression [48, 54]. The most widely-used technique to identify flaky test failures, *Rerun*, is to rerun each failing test multiple times after witnessing the failure: if some rerun passes, the test is definitely flaky; but if all reruns fail, the status is unknown. Rerun is supported by several testing frameworks, e.g., Android [21], Jenkins [52], Maven [77], Spring [75], and the Google TAP system [42, 63]. Developers do *not* proactively search for flaky tests as a maintenance activity, instead simply using Rerun to identify that a given test failure is flaky.

There is little empirical guidance describing how to rerun failing tests in order to maximize the likelihood of witnessing the test pass. Reruns might need to be delayed to allow the cause of the failure (e.g., a network outage) to be resolved. Flaky tests are non-deterministic by definition, so there is no guarantee that rerunning a flaky test will change its outcome. The performance overhead of Rerun scales with the number of failing tests — for each failed test, Rerun will rerun it a variable number of times, potentially also injecting a delay between each rerun. Rerunning *every* failed test is extremely costly when organizations see hundreds to millions of test failures per day. Even Google, with its vast compute resources, does not rerun all (failing) tests on every commit [64, 87] but reruns only those suspected to be flaky, and only outside of peak test execution times [64].

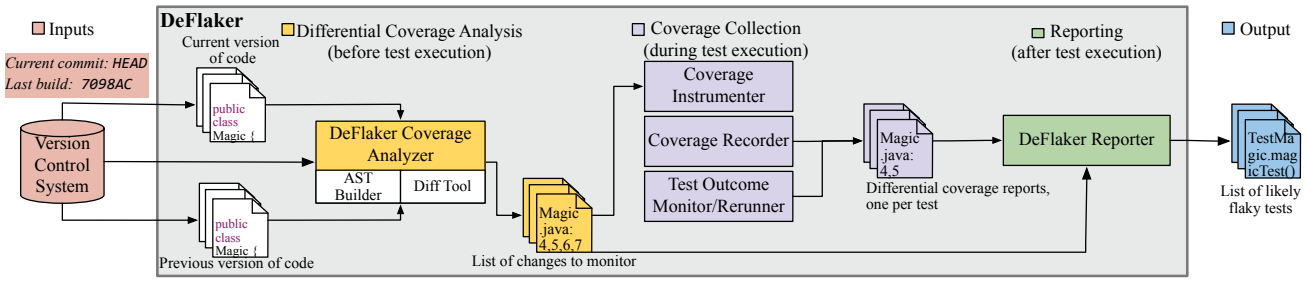

Figure 1: High-level architecture of DeFlaker, with three phases: before, during and after test execution.

We performed an extensive evaluation of Rerun (§3.1) on 5,328 test failures in 5,966 historical builds of 26 open-source Maven-based Java projects. The flaky test detector in Maven, which reruns each test shortly after it failed and in the *same JVM* in which it failed, marked only 23% of the 5,328 test failures as flaky. By isolating each rerun in its own JVM, and further rebooting the build system to clean the state between reruns, we confirmed that in fact, at least 95% of those failing tests were flaky. Maven likely does not isolate test reruns because of the high cost of creating a process, loading classes, and preparing clean state for each test; our prior study found that isolating tests can add a 618% overhead [25]. Hence, to effectively find flaky tests, Rerun introduces substantial performance overhead. Developers should ideally be able to know *immediately* after a test fails that it is flaky. Even if a developer suspects a test to be flaky, our goal is to provide evidence for that suspicion from the outcome of a single test execution: if the test fails, is it due to a regression or flakiness?

We propose a new and efficient technique, *DeFlaker*, that is complementary to Rerun and often marks failing tests as flaky immediately after their execution, without any reruns. Recall that a test is flaky if it both passes and fails when *the code that is executed by the test did not change*; moreover, a test failure is *new* if the test passed on the previous version of code but fails in the current version. A straw-man technique to detect flaky tests is to collect *complete* statement coverage for each test (of both the test code and the code under test), intersect coverage with the changes, and report as flaky new test failures that did not execute any changed code. However, collecting full statement coverage can be expensive.

Our key insight in DeFlaker is that one need not collect coverage of the *entire codebase*. Instead, one can collect only the coverage *of the changed code*, which we call *differential coverage*. Differential coverage first queries a version-control system (VCS) to detect code changes since the last version. It then analyzes the code and constructs an abstract-syntax tree for each changed file to determine where instrumentation needs to be inserted to track execution of each change. Finally, when tests run, it monitors change execution, generating an individual change-coverage report for each test.

We present our DeFlaker tool that detects flaky tests through lightweight differential coverage tracking. If a test fails but does not cover any changed code, DeFlaker reports the test as flaky *without* requiring any reruns. Our evaluation of DeFlaker uses a traditional experimental methodology on historical builds on our own servers, and we also propose a novel methodology for evaluating testing and analysis tools on open-source projects in real time and in the exact same build environments that the projects' developers use. This new methodology allowed us to evaluate DeFlaker on complex projects that we could not easily get to compile and execute in our own local environments (the traditional methodology). Replicating software builds in a lab environment can be tricky, when complex projects may include a handful of manual configuration steps before they can compile. Even then, subtle differences in environment (e.g., the exact version of Java, and the distinction between an Oracle JVM and an OpenJDK JVM) can lead to incorrect results. The marginal human cost of adding a new project to an evaluation can be very high. In contrast, when projects are currently designed to be automatically compiled and tested in a standard environment (e.g., TravisCI), it can be much easier to study more projects.

Our experiments in the live environments involved 93 projects and 614 commits, and we found 87 previously unknown flaky tests in 10 of these projects. We reported 19 of these newly-detected flaky tests, and developers have already fixed 7 of them. In order to perform a larger evaluation (without abusing TravisCI's resources), we also performed a traditional evaluation on 26 projects and 5,966 commits running in our own environment, in which DeFlaker detected 4,846 flaky test failures among 5,328 confirmed flaky tests (95.5%), with a low false positive rate (1.5%). In comparison, the current Rerun in Maven found only 23% of these same test failures to be flaky. For projects with very few test failures, DeFlaker can impose almost no overhead: only collecting coverage in a single rerun of failed tests. For projects with too many failures to rerun, we found that DeFlaker imposes a low enough overhead (4.6%) to be used with every test run, eliminating the need for reruns.

The primary contributions of this paper are:

- **New Idea:** A general purpose, lightweight technique for detecting flaky tests by tracking differential coverage.
- **Robust Implementation:** A description of our open-source flaky test detector, DeFlaker [28].
- **Extensive Evaluation:** Experiments with various Rerun approaches and DeFlaker on 5,966 commits of 26 projects, taking 5 CPU-years in aggregate; to our knowledge, this is the first empirical comparison of different Rerun approaches.
- **Novel Methodology:** A new research methodology for evaluating testing tools by shadowing the live development of open-source projects in their original build environment.

2 DEFLAKER

DeFlaker detects flaky tests in a three-phase process illustrated in Figure 1. In the first phase, *differential coverage analysis*, DeFlaker uses a syntactic diff from VCS and an AST builder to identify a list of changes to track for each program source file. In the second phase, *coverage collection*, DeFlaker injects itself into the test-execution process, monitoring the coverage of each change identified in the

```
public class SuperOld {
  public void magic() {
  }
}
public class SuperNew extends SuperOld {
  public void magic() {
    assert(false); // causes test to fail
  }
}
public class App extends SuperOld/*SuperNew*/ {
}
public class TestApp {
  @Test public void testApp() {
    new App().magic();//unchanged line changes behavior
  }
}
```

Figure 2: Sample change that challenges a syntactic diff

prior phase. Finally, once tests have finished executing, DeFlaker analyzes the coverage information and test outcomes to determine the set of test failures that are likely flaky. In principle, these reports could also be printed immediately, as tests fail, but we report them at the end of the test run to conform with existing testing APIs.

2.1 Differential Coverage Analysis

DeFlaker analyzes program code and version history to determine how to track the impact of changed code, combining syntactic change information from a VCS (in our case, Git), with structural information from each program source file. DeFlaker tracks the coverage of changes to *all* program source files (including both test code and program code). The output of this phase are locations in the program code in which to add coverage probes that are used at runtime to determine if a test executes changed code.

As other researchers pointed out, e.g., in the context of regression-test selection [39, 66] and change-impact analysis [23, 69, 84], using solely syntactic change information is often insufficient in object-oriented languages. In other words, it is necessary to monitor even some syntactically unchanged lines to determine that a change gets executed. For instance, changes that modify the inheritance structure of a class or method overloading may cause dynamic dispatch to occur differently at a call site that itself is unchanged.

Figure 2 shows an example test prone to some of these challenges: changing the super type of `App` from `SuperOld` to `SuperNew` would cause the (unchanged) `magic` method call to refer to a different implementation, causing the test to fail instead of pass. Similarly, adding an empty implementation of the `magic` method to the `App` class would change the test behavior as well. To handle these sorts of changes, prior work either (1) performs static analysis to model changes at a fine granularity or (2) instead, simply tracks the changes at a coarse granularity. Traditional approaches—e.g., JDiff [23], Chianti [69], DejaVOO [66], and FaultTracer [84]—model the dynamic dispatch behavior of the JVM to infer the exact semantics of each change. These approaches can precisely identify the impact of adding or removing a method to or from a class, or changing a type hierarchy, allowing downstream tools to take into account any potential changes to method invocation.

More recent work has shown that in some contexts it can be more cost-effective to model these changes more coarsely, greatly reducing the cost of the analysis, at the cost of some loss in precision [39]. These tools track class file coverage rather than statement coverage: if a test references a class in any way, then that class is considered covered. This approach is safe because it will identify when a test might be impacted by some change, but may flag some changes as impacting a test, even if they do not. Referring again to Figure 2, a class coverage tool would declare classes `App`, `SuperOld`, `SuperNew`, and `TestApp` as covered by `testApp()`, which references all these classes when executed. This coarse granularity introduces imprecision: a change to a statement not covered by `testApp()` in any of those classes would be considered covered. However working at a coarser granularity is fast and does not require the type resolution of heavyweight static analysis. In our context, working only at coarser granularity could lead to false negatives: flaky tests may be falsely considered to have executed changed code, preventing DeFlaker from detecting them.

DeFlaker takes a *hybrid* approach (similar to recent work in hybrid regression test selection [83]), benefiting from the low upfront analysis cost of class-level coverage, while often still achieving the high precision of statement-level coverage. DeFlaker does not need to find that *every* change was covered by a test, but only that *some* change was covered by a test. Therefore, DeFlaker can ignore tracking some changes, but infer their coverage from other facts. For each statement that is identified as added, removed or changed (based on the syntactic diff), DeFlaker classifies it as: (1) safe to track using statement-level coverage (e.g., changing a statement in a method body); (2) not safe to track using statement-level coverage, so instead should be tracked using class-level coverage; or (3) does not need to be tracked. §3.3 evaluates performance improvements of hybrid coverage over class-level coverage.

DeFlaker identifies new, changed, and removed files directly from the VCS. DeFlaker assumes that removed classes, if referenced by other classes, will result in changes in those other classes, which it will track. For each new or changed file, DeFlaker builds an abstract-syntax tree (AST) representation of the file. For new files, DeFlaker marks every type (i.e., class, interface, enum, or annotation) defined in that file to be tracked with class coverage. If the change is to a statement in a method body or to a variable initializer, DeFlaker tracks that statement directly. If the change adds/removes an exception handler, DeFlaker marks the change as covered if the first statement enclosed by the try block is executed. DeFlaker defers to class-level tracking of the enclosing type if a change alters the parent type or list of interfaces implemented by that type, or if the change removes (or reduces the visibility of) a method declared in that type — changes that may impact dynamic dispatch in non-trivial ways. Adding a new method to a type might also change dynamic dispatch, but in a predictable way: the new method would be called instead of another. Therefore, DeFlaker simply tracks the coverage of the first statement in newly added methods. DeFlaker does not track changes that add or remove field definitions, assuming that for such changes to impact some test, other code must change (e.g., code referencing that field). Changes to field initializer code are tracked using statement-level coverage.

Limitations. The kinds of changes listed above are the only ones that DeFlaker tracks. Hence, DeFlaker's differential coverage is not complete, because it may not report that some changed code was covered even if it is. For instance, DeFlaker ignores the impact of various changes on code that uses reflection: adding a field, for example, will change the array of fields returned by the `Class.getDeclaredFields()` method, which some test might call, but

DeFlaker would not track. Similarly, DeFlaker does not consider the impact of changes to annotations of types, fields, or methods. These limitations of DeFlaker's hybrid coverage are an intentional trade-off, because the goal of DeFlaker is to detect, as efficiently as possible, whether a test executed changed code.

It is important to highlight what problems can arise if a tool under-approximates code coverage. When DeFlaker misses that a test executes some changed code, it only results in a *false positive*, as DeFlaker reports the test as flaky when it may not be. In other words, DeFlaker does not miss a flaky test if it misses some coverage. In contrast, a regression test-selection tool [34, 39, 45, 66, 70] that under-approximates coverage could miss to run a test and thus could miss a bug, resulting in a *false negative*. Hence, regression test-selection tools aim to over-approximate code coverage. Additionally, because DeFlaker only tracks changes to Java files, DeFlaker may completely miss some non-Java changes that impact the execution of a test, e.g., changing a configuration file. This limitation is not fundamental, and indeed, DeFlaker could track more detailed dependencies between tests and external resources using approaches from regression test-selection work [26, 32]. Our experiments showed that considering *any* change to a non-Java files as impacting every test (a safe, but imprecise approach) did not substantially change DeFlaker's false-alarm rate.

2.2 Coverage Collection

DeFlaker uses bytecode injection to insert statement and class coverage-tracking probes as classes are loaded into the JVM during test execution. Tracking statements is simple: for each line L containing a statement that should be tracked, DeFlaker inserts a probe just before the bytecode instructions for line L to record its execution. To track class coverage of type T, DeFlaker inserts probes in the static initializer of T, in each constructor of T, and in each non-private static method of T. This approach is safe: it always detects the use of type T in the absence of test-order dependencies [25, 26, 43, 85], similar to the regression-test selection tool Ekstazi [39]. DeFlaker generates a coverage report that lists each changed line/class covered by each test. DeFlaker also monitors the outcome of each test, ensuring that this information can be tracked for historical comparisons.

2.3 Reporting

Once the test run finishes, DeFlaker collects outcomes and coverage of each test and generates a report. DeFlaker marks a test as flaky if: (1) it previously passed, now fails, and did not cover any code that changed; or (2) it failed, was rerun on the same code, and subsequently passed. If any non-code files changed since the last run, DeFlaker warns that the failing test *might* be flaky and that the non-code changes may have caused the failure. For each failing test that covered changed code DeFlaker prints a message containing each part of the changes that the test covered. This can help developers debug tests (flaky or not) that failed due to code changes. Lastly, for changes that were not covered by any test, DeFlaker prints a warning that changes are not being tested.

2.4 Implementation

While the DeFlaker approach is generic and can apply to nearly any language or testing framework, we implemented our tool for Java, the Maven build system, and two testing frameworks (JUnit and TestNG). DeFlaker is thread-safe and fully supports multi-threaded code. DeFlaker is available under an MIT license on GitHub [28] with binaries published on Maven Central. More information on DeFlaker is available at http://www.deflaker.org.

```
<extensions>
 <extension>
  <groupId>org.deflaker</groupId>
  <artifactId>deflaker-maven-extension</artifactId>
  <version>1.4</version>
 </extension>
</extensions>
```

Figure 3: Adding DeFlaker to a Maven build is trivial, consisting of just these 7 lines to be inserted to the build file.

DeFlaker consists of six core components: a Maven build extension, two Maven plugins, a Maven Surefire test execution provider, a Java instrumentation agent, and a JUnit/TestNG test-outcome listener. DeFlaker relies only on public, documented, and supported APIs to interface with these systems, allowing it to be widely compatible with many versions of each tool. DeFlaker supports arbitrarily complex Maven configurations (with multiple modules, multiple executions of the `test` or `verify` phases, and arbitrary configuration arguments passed to each phase). We tested DeFlaker on the most recent versions (at time of writing) of Maven (3.2.5, 3.39, 3.5.0), the Surefire test execution Maven Plugin (2.18, 2.19, 2.20), JUnit (4.6–4.12) and TestNG (5.5-5.14.9 and 6.8-6.11). Despite having several components, configuring a project to use DeFlaker is trivial: a developer need only add the Maven extension to the build file, as shown in Figure 3. We briefly describe the implementation and functionality of each DeFlaker component.

DeFlaker Maven Extension injects our coverage analysis plugin and reporting plugin into each module being built. Maven's Build Extension interface [22] allows third-party tools to customize how Maven processes build files, including the ability to rewrite build files as they are loaded. Our extension also modifies the execution of the Surefire and Failsafe plugins to include DeFlaker instrumentation agent when tests are run, and the appropriate DeFlaker test listener (JUnit or TestNG, based on the project).

DeFlaker Coverage Analyzer Maven Plugin performs the change analysis described in Section 2.1. We use the JGit library [53] to perform the syntactic diff, and the Eclipse JDT library [78] to construct the ASTs. The coverage analyzer runs just once even for a multi-module build, detecting what changes need to be tracked for every file in the repository, regardless of what module that file is in. Out of the box, Surefire supports rerunning failed tests within the *same* JVM, just after they fail. While evaluating DeFlaker, we found that Maven's Rerun technique did not even come close to flagging as many test failures as flaky as DeFlaker (§3.1). Hence, to help with debugging and verifying that a flaky report represents an actual flaky test, DeFlaker includes a **Surefire Provider** to re-run each failed test in a *fresh* JVM, which we found can help show more tests as flaky than Maven's Rerun. This optional feature is exposed through the system property `deflaker.rerunFlakiesCount`. Another property, `deflaker.delayedCoverage`, instructs DeFlaker to collect coverage only during reruns and not while running tests normally. The latter configuration has effectively no overhead for coverage tracking (unless some test fails).

DeFlaker Java Instrumentation Agent rewrites all class-files as they are loaded into the JVM during test execution. It uses cached coverage analysis results to determine what coverage probes to insert in which classes. **DeFlaker Test Listener** interfaces with the testing framework (JUnit or TestNG) and records when tests finish, collecting coverage results and resetting internal counters. **DeFlaker Reporting Maven Plugin** executes after the unit test and integration test phases, collecting coverage and test outcomes, and handling all functionality outlined in Section 2.3.

3 EVALUATION

We performed an extensive evaluation of DeFlaker and Rerun with the goal of answering several key research questions:

RQ1: How many test failures do different Rerun approaches mark as flaky?

RQ2: How many test failures does DeFlaker mark as flaky?

RQ3: Given that DeFlaker only detects flaky tests that do not cover changed code, can it detect real flaky tests? Does DeFlaker's hybrid coverage find more flakes than class-level coverage?

RQ4: What is the overhead of running DeFlaker for *all* tests?

Experimental Environments To answer these questions, we created two evaluation environments. The *historical* environment simulates how DeFlaker could have worked had developers used it in the past, and allows us to run experiments on thousands of commits. The historical environment was constructed on our own servers, and required us to manually identify projects that we could successfully build and test. This evaluation methodology is common in recent studies on regression testing [26, 32, 39, 41, 59, 60, 68, 73]. In contrast, the *live* environment requires *no configuration or tuning* of individual projects to get them to successfully compile and run their tests. Instead, the live environment evaluates DeFlaker on a number of open-source projects in the exact same environment that the developers use and while development is happening on those projects, i.e., exactly as a developer would use DeFlaker. The latter environment allowed us to evaluate DeFlaker on more projects than would have been possible had we manually configured each project to build and test. This environment is a *new* evaluation methodology, to our knowledge, not used before this paper. Previous studies *have* looked at builds on Travis [29, 56] but *have not* deployed tools to instrument those builds.

Our historical evaluation environment consisted of 250 Amazon EC2 "c3.large" instances, each with 2 Intel Xeon E5-2680 v2 vCPUs, 3.75GB of RAM. Each instance ran Ubuntu 14.04 "trusty", and used either Java 1.7.0_45 or 1.8.0_131 for builds (depending on the age of the code being compiled and executed), with Maven 3.3.9. To answer our RQs, we completed a total of 47,748 Maven builds in this historical environment, taking *over 5 CPU-years* to run. The live environment ran on the TravisCI platform. Each project included in the live evaluation was mirrored and configured to use DeFlaker; when developers pushed commits to their own repositories and triggered builds, our forks synced and triggered a build of our repository as well. Because we selected projects that used TravisCI for their own builds, our build environment was identical to theirs. The live environment completed 614 builds between May 12, 2017 and Aug 10, 2017. Note that we conducted far fewer builds in the live environment due to resource constraints: we did not want to abuse the free TravisCI service. All of our projects were configured on TravisCI under the same organization, allowing TravisCI to easily throttle and restrict our builds if needed.

Project Selection. To identify projects to include in the live environment, we selected projects from GitHub that use TravisCI, have Java as the primary language, and use Maven, yielding a total of 96 projects. A complete list of all of these projects appears in the following tables, and additional details (including the revisions used) are available on our supplemental website [27].

For our historical environment, we selected projects that we knew had at least one flaky test (and a range of commits that had the flaky behavior), which served as a ground truth, allowing us to calculate how often DeFlaker would find bonafide flaky tests. We selected 96 known flaky tests from 26 projects, consisting of: (1) 4 projects (achilles, checkstyle, jackrabbit-oak, and togglz) from our live experiment in which we identified 5 flaky tests along with the starting and ending commits exhibiting that flaky behavior; and (2) 22 other open-source projects in which the project developers previously found and fixed a total of 91 flaky tests.

The 22 projects with previously fixed flaky tests contain 17 projects that we obtained from querying GitHub for terms related to flaky tests ("intermit" or "flak") and 5 projects that we selected from a prior study on flaky tests [61]. From the results of querying GitHub, we selected 81 tests (from 17 projects) where we could confirm by manual inspection that the commit message actually fixed a flaky test, and for which we could still run these old tests. We also consulted a prior study of flaky tests [61], selecting from it 10 flaky tests (from 5 projects) that we could build and run. These flaky tests and projects come from various categories domains (networking, databases, etc.), offering a relatively diverse sample of flaky tests. For each of these 96 flaky tests, we identified a precise set of commits where the test was flaky by manually investigating the cause of flakiness. To limit the time for our experiments, we run DeFlaker only on 500 randomly selected commits from those ranges. The test could have failed for any build of those commits.

3.1 RQ1: Finding Flaky Tests through Rerun

While we knew of 96 flaky tests in the projects for the historical environment, we expected that there were more, too, and wanted to study each approach's ability to find these. The methodology for this study was to run each of the 5,966 builds, rerunning tests as they failed. At first, we considered only the Rerun approach implemented by Maven's Surefire test runner [77], which reruns each failed test in the *same* JVM in which it originally failed. We were surprised to find that this approach only resulted in 23% of test failures eventually passing (hence, marked as flaky) even if we allowed for up to five reruns! The *strategy* by which test is rerun matters greatly: make a poor choice, and the test will continue to fail for the same reason that it failed the first time. Hence, to achieve a better oracle of test flakiness, and to understand how to best use reruns to detect flaky tests, we experimented with the following strategies, rerunning failed tests: (1) **Surefire**: up to five times in the same JVM in which the test ran (Maven's Rerun); then, if it still did not pass (2) **Fork**: up to five times, with each execution in a clean, new JVM; then, if it still did not pass (3) **Reboot**: up to

Table 1: Number of flaky tests found by re-running 5,966 builds of 26 open-source projects. We consider only new test failures, where a test passed on the previous commit, and report flakes reported by each phase of our Rerun strategies. DeFlaker found more flaky tests than the Surefire or Fork rerun strategies: only the very costly Reboot strategy found more flaky tests than DeFlaker.

		Test Methods in Project		Total New	Confirmed flaky by Rerun strategy			DeFlaker labeled as: Flaky		Not Flaky	
Project	**#SHAs**	**Total**	**Failing**	**Failures**	**Surefire**	**+Fork**	**++Reboot**	**Confirmed**	**Unconf.**	**Confirmed**	**Unconf.**
achilles	227	337	77	242	13	14	230	225	4	5	8
ambari	500	896	7	75	52	71	74	74	0	0	1
assertj-core	29	6,261	2	3	2	2	2	2	0	0	1
checkstyle	500	1,787	1	1	0	0	0	0	0	0	1
cloudera.oryx	332	275	23	29	5	5	5	5	20	0	4
commons-exec	70	89	2	22	22	22	22	21	0	1	0
dropwizard	298	428	1	60	60	60	60	55	0	5	0
hadoop	298	2,361	365	1,081	284	865	1,054	1,028	25	26	2
handlebars	27	712	7	9	3	7	7	6	2	1	0
hbase	127	431	106	406	62	242	390	383	12	7	4
hector	159	142	12	87	0	74	79	72	4	7	4
httpcore	34	712	2	2	2	2	2	1	0	1	0
jackrabbit-oak	500	4,035	26	34	10	33	34	32	0	2	0
jimfs	164	628	7	21	21	21	21	15	0	6	0
logback	50	964	11	18	18	18	18	18	0	0	0
ninja	317	307	37	122	37	77	110	94	2	16	10
okhttp	500	1,778	129	333	296	305	310	231	0	79	23
oozie	113	1,025	1,065	2,246	42	2,032	2,244	2,234	0	10	2
orbit	227	86	9	86	84	85	85	73	0	12	1
oryx	212	200	38	46	14	14	46	14	0	32	0
spring-boot	111	2,002	67	140	73	107	135	135	3	0	2
tachyon	500	470	4	5	3	5	5	5	0	0	0
togglz	140	227	21	28	5	14	28	28	0	0	0
undertow	7	340	0	0	0	0	0	0	0	0	0
wro4j	306	1,160	114	217	39	96	99	80	8	19	110
zxing	218	415	2	15	15	15	15	15	0	0	0
26 Total	5,966	28,068	2,135	5,328	1,162	4,186	5,075	4,846	80	229	173

five times, running a `mvn clean` between tests and rebooting the machine between runs.

Table 1 shows the results of this study, including the number of test failures confirmed as flaky by each Rerun strategy. Overall, we observed 2,135 tests that exhibited some potentially flaky behavior, having *new* failures (passing on one commit, then failing on the following commit). Collectively, these tests had a total of 5,328 new failures, with 1,162 detected by the Surefire (same JVM) reruns, 4,186 detected by the Surefire strategy *or* the Fork strategy, and 5,075 detected by the Surefire, Fork, *or* Reboot strategy. This result is striking: the existing flaky test detector in Maven only identified 23% of the flaky failures identified by all three strategies (including the heavyweight Reboot strategy)!

It would be difficult to fairly state the *cost* of these various reruns, as the cost of rerunning a test varies with many factors (how long the test took to run the first time, how much shared state it might need to setup, etc.). If all tests fail, then the cost of rerunning them all once would be at least the cost of running the test suite the first time. Even when (re)running fewer tests, any Rerun strategy aside from Maven's Rerun will incur the high computational cost of isolating tests in their own JVM as documented by prior work [25], or more if employing stronger isolation similar to our Reboot strategy [65].

Table 2 summarizes Rerun results by strategy, including the number of reruns needed to witness the flake. From these results, we may conclude that only one rerun is needed for *each kind* of rerun: first run a failing test in the same JVM once, and if it fails, run in a new JVM once, and if still fails, run after a reboot. Performing more runs of the same kind increases the cost but does not substantially increase the chance to obtain a pass. In other words, changing the kind of rerun is more likely to help than just increasing the number of reruns, and various testing frameworks [21, 42, 52, 63, 75, 77] that support reruns and offer defaults such as 3, 5, or 10 reruns of the same kind should rather offer new kinds of reruns. DeFlaker allows Maven users to automatically have tests rerun in a new JVM.

3.2 RQ2: Finding Flaky Tests with DeFlaker

We evaluated DeFlaker's efficacy in marking test failures as flaky on the same test executions as in the previous section. That is,

Table 2: Number of reruns required to confirm the flakes from Table 1, and the percent of flakes confirmed by reruns at each tier also confirmed by DeFlaker without any reruns required. If a flake was confirmed, we stopped rerunning it; we executed the three rerun strategies in the order listed.

	# Reruns to Find Flaky						% Also Found
Strategy	**1**	**2**	**3**	**4**	**5**	**Total**	**by DeFlaker**
Same JVM	994	90	38	24	16	1,162(22.9%)	87.6%
New JVM	2,913	32	32	19	28	3,024(59.6%)	98.4%
Reboot	889	0	0	0	0	889(17.5%)	95.8%
All						5,075(100.0%)	95.5%

when running tests in our historical environment, we also ran DeFlaker. We used the reruns as an oracle for whether a test failure was truly flaky, which allowed us to identify DeFlaker-reported flaky failures that were confirmed as failures versus those that remain unconfirmed. Note that we may *over-estimate* the number of false alarms for DeFlaker because the test could still be flaky if investigated further. Table 1 reports these results. In summary, DeFlaker reported 4,846 failures as flaky (95.5% of confirmed flakes) with a very low false alarm rate, just 1.5%. These reports represent a total of 1,874 flaky tests. DeFlaker finds significantly more flaky test failures than the Surefire strategy and slightly more than those found using the Fork JVM strategy.

Given that most of the flakes (77%) couldn't be confirmed through a simple rerun in the same JVM, we believe that DeFlaker is even more valuable to developers, as it can provide immediate, trusted feedback with significantly less delay. Table 2 shows what percent of flakes detected by each rerun technique were also detected by DeFlaker. Of those 3,024 flaky tests detected only after rerunning tests in a new JVM, DeFlaker accurately marked 98.4% of them as flaky, suggesting that the developers could have detected these flaky tests without paying the cost to rerun these tests. Most compelling is that for the 889 failures that required the most expensive reruns to confirm as flaky tests – running a `mvn clean` between tests and rebooting the machine between runs – DeFlaker detected 95.8% of these flaky test runs *without any expensive rerun*.

Overall, based on these results, we find DeFlaker to be a cost-beneficial approach to run *before or together with* Rerun, and we also suggest a potentially optimal way to run Rerun. For projects that have lots of failing tests, DeFlaker can be run on all the tests in the entire test suite, because DeFlaker immediately detects many flaky tests without needing *any* rerun. For projects that have a few failing tests, DeFlaker can be run only when rerunning failed tests in a new JVM; if the tests still fail but do not execute any changed code, then reruns can stop without costly reboots.

To evaluate DeFlaker on a wider set of projects, and to find previously unknown flaky tests, we performed experiments in the live environment, the results of which are summarized in Table 3. Of the 96 open-source Java projects that we shadowed, relatively few were actively developed (Labuschagne et al. reported a similar finding [56]), having more than a handful of builds over the three-month time period. Of particular note are the projects where DeFlaker was only run on a handful of builds (i.e., achilles, quickml), yet still identified flaky tests. In total, only 10 projects had at least one test that DeFlaker detected as a candidate flake (that had passed in the previous commit, then failed in the current commit). We found a total of 91 failures that were potential flakes, and confirmed that they were all flakes by repeatedly rerunning them on our local machines: if they eventually passed given the same code, and no other changes, we declared them true flakes. Although we performed far fewer builds in the live environment (constrained by the resources provided by TravisCI), DeFlaker actually identified more flaky test failures per-build in the live environment (546) than in the historical environment (91/614). We found no false positives in this study (but DeFlaker can have false positives, as discussed previously). Unfortunately, we cannot comment on the efficacy of individual rerun strategies here, as we began collecting this data in the field before automating the three-strategy rerun approach described in the previous section.

Table 3: Results from live environment, showing only projects that had tests fail after previously passing. Showing the total failures, and for DeFlaker flake reports: **C**onfirmed flakes, **R**eports of flakes sent to developers, **A**ddressed flakes by developer.

			New	Flake Reports			
Project	**Tests**	**# SHAs**	**Fails**	C	R	A	Issue Links
achilles	573	5	2	2	2	2	[2, 3]
checkstyle	26,935	96	1	1	1	1	[4]
geoserver	4,919	60	39	39	1	0	[5]
jackrabbit-oak	9,788	99	5	5	2	1	[6, 7]
jmeter-plugins	1,571	19	1	1	1	0	[8]
killbill	14,827	31	26	26	1	0	[9]
nutz	1,117	87	1	1	1	1	[10]
presto	4,554	203	11	11	7	0	[11–16]
quickml	98	2	2	2	2	0	[17, 18]
togglz	748	12	3	3	2	2	[19, 20]
10 Total	65,130	614	91	91	19	7	

Out of the 91 previously unknown flaky tests that DeFlaker detected, we reported 19 to developers (one test in togglz and three tests in presto had been previously detected as flaky by the developers). Of the 19 reports, 7 have been addressed, most by removing or reducing flakiness, but one by removing/ignoring the test (which is why we use the term "addressed" rather than "fixed," because removing a test is not a fix for that test, but it was a fix for the test suite as the developers found more value in removing this test than keeping it and having to deal with its flakiness). In several cases, we found that flaky tests previously believed to have been fixed by developers were still flaky, due to the same or different causes [16]. We received several very positive responses, such as "Thank you very much for your help with this. I just committed a fix. Looking forward to see more green builds now." [19] and "@flakycov, thanks a lot!" [4]. All remaining reports are still open at the time of writing, with the exception of one geoserver issue: those developers could not reproduce the failure and were not interested in debugging it [5]. In cases where a project had multiple flaky tests, we began by opening an issue on just one of the flaky tests, and did not open more issues if the developers didn't respond. We reported two flaky tests in presto with similar root causes in one issue.

3.3 RQ3: Coverage tracking for flake detection

We evaluated the efficacy of using DeFlaker's hybrid statement- and class-level coverage in comparison to simple class-level coverage used by a state-of-the-art regression test selection tool [39]. Because DeFlaker uses differential coverage to judge when a failing test is flaky, it can only identify test failures as flaky when the test does not cover any recently changed code. If the test does cover changed code, DeFlaker cannot determine if the failure is flaky, or if the test will always fail. To determine the limitations of this coverage-based approach, we evaluate how often flaky tests cover changed code, for two kinds of coverage.

To answer this question, we need to know precisely the versions of code in which each flaky test was present and could have flaked on. To perform this analysis on *all* 4,846 flaky failures would not be possible, as debugging flaky tests is a very time consuming task. Hence, to answer this question, we focused on the 96 known flaky tests that we had manually identified in the 5,966 commits of code that we built and tested. We calculated the percentage of test runs on which DeFlaker would have marked the test as flaky *if* it failed, regardless of whether the test actually failed or not in our experiments on that given commit. Because the test was flaky for each of these commits, it could have failed in any of those runs. Also, we assume that, on average, a test run that covers no changed code when the test passes would also cover no changed code when the test fails (although the exact coverage between a test failure and a test pass does differ, at least for one branch or exception that determines the test outcome). Table 4 shows the results per project, for flaky detection using both DeFlaker's hybrid statement/class coverage and only class coverage [39]. DeFlaker identified that the flaky test did not cover any code change 90% of the time. The improvements over class coverage are roughly 11 percentage points, showing the importance of DeFlaker's precise analysis, compared with a regression test selection tool that uses class coverage [39].

3.4 RQ4: Performance

Recall that DeFlaker can be run on only failing tests (potentially when rerun) or on all tests. In some cases, it might be preferred that DeFlaker runs for every test, regardless of the outcome, to potentially eliminate the need for reruns. In this case, it is important to understand what slowdown DeFlaker might add to the testing process. We measured the relative performance of running tests without any coverage tracking tool, with DeFlaker, and with the most recent versions of three most popular code-coverage tools for Java: JaCoCo (0.7.9), Cobertura (2.1 with Maven plugin 2.7), and Clover (OpenClover 4.2.0) [24]. We also used a recent research tool that tracks class coverage: Ekstazi (version 4.6.1) [39]. We configured each of these tools following the instructions provided on their respective installation pages. For our own tool, DeFlaker, we used version 1.4 (available on Maven Central and our website [27]). By default, JaCoCo and Cobertura only create a single coverage report containing the overall results for the entire test suite, which is not useful for identifying if a particular test executed any changed code. Hence, we measured also the performance of using these two tools to generate coverage reports for each test by connecting our test listener to each tool and triggering a coverage dump and reset after each test.

Table 4: DeFlaker's efficacy finding 96 known flaky tests across 5,966 different commits of 26 open-source projects, comparing its novel hybrid statement/class coverage with only class coverage.

Project	Known Flaky Tests	# SHAs	% of Runs Flaky by: Hybrid Cov	Class Cov
achilles	1	227	77%	71%
ambari	1	500	100%	100%
assertj-core	1	29	97%	83%
checkstyle	1	500	100%	98%
cloudera.oryx	1	332	95%	94%
commons-exec	1	70	94%	69%
dropwizard	1	298	83%	79%
hadoop	3	298	92%	85%
handlebars	7	27	89%	78%
hbase	3	127	86%	72%
hector	1	159	100%	96%
httpcore	1	34	82%	82%
jackrabbit-oak	1	500	75%	66%
jimfs	1	164	54%	24%
logback	11	50	91%	83%
ninja	2	317	90%	80%
okhttp	33	500	91%	81%
oozie	1	113	91%	75%
orbit	1	227	81%	74%
oryx	3	212	100%	100%
spring-boot	5	111	98%	98%
tachyon	7	500	77%	58%
togglz	3	140	97%	94%
undertow	3	7	75%	17%
wro4j	1	306	68%	56%
zxing	2	218	98%	98%
26 Total	96	5,966	88%	77%

We considered running this experiment on all 26 projects studied in **RQ1** and **RQ2**, but found that several were not well amenable to performance measurements, with a very high variance in test-execution times, even without any coverage tool (e.g., `cloudera.oryx`, which took on average 241 seconds with a standard deviation of 110 seconds), and filtered out 9 such projects. For each of the remaining 17 projects, we executed its build (using `mvn verify`) on the 10 consecutive versions (most recent as of August 10, 2017) with each tool, and repeated this process 15 times to reduce noise. Considering all of the executions, we performed a total of 21,600 builds over a total of approximately 445 CPU-days (counting the total time needed to checkout and build each project).

We collected the outcomes of all tests from each execution and marked the tool as 'n/a' if the tool caused errors. Table 5 shows the results, including the baseline time spent running each test suite without collecting any coverage (plus or minus a single standard deviation), and the relative overhead of each tool. To provide a fair comparison, we timed only the tool's instrumentation phase (if applicable), and the tests running phase — we exclude time to fetch dependencies, compile code, generate reports, etc. Cobertura runs the entire test suite *twice* (once without, and once with coverage

Table 5: Measurements comparing the runtime performance overhead of DeFlaker and four other coverage tools vs. the baseline test execution. Cobertura and Clover did not work for many projects (marked as 'n/a'), Ekstazi and Cobertura each timed out (after a four hour limit) on one project (marked as 't/o').

Project	Test Methods	Baseline Time (sec)	DeFlaker	Ekstazi	JaCoCo Per-test	JaCoCo Default	Cobertura Per-test	Cobertura Default	Clover
achilles	563	184.52 ± 6.72	6.2%	62.3%	34.9%	45.2%	n/a	n/a	n/a
ambari	5,186	3,726.11 ± 573.55	6.2%	29.1%	52.9%	52.3%	n/a	n/a	n/a
assertj-core	10,334	24.56 ± 5.09	12.2%	100.6%	30.9%	141.9%	180.1%	194.3%	129.2%
checkstyle	2,452	75.49 ± 10.61	3.7%	27.0%	25.3%	39.6%	18.5%	45.9%	n/a
commons-exec	96	64.85 ± .79	3.7%	2.9%	1.5%	1.4%	2.8%	4.7%	4.4%
dropwizard	1,455	219.62 ± 7.06	5.6%	94.4%	59.6%	68.9%	33.5%	50.5%	n/a
hector	349	481.96 ± 141.28	4.8%	29.6%	16.7%	14.0%	3.0%	10.6%	9.5%
httpcore	1,059	80.77 ± 10.12	6.5%	18.8%	14.9%	18.6%	18.8%	41.2%	30.2%
jackrabbit-oak	9,694	1,290.23 ± 336.05	0.2%	34.4%	31.1%	34.6%	n/a	n/a	n/a
killbill	801	354.66 ± 63.47	3.8%	21.3%	24.8%	19.8%	n/a	n/a	4.7%
ninja	797	55.53 ± 4.83	8.4%	87.6%	101.8%	385.0%	41.2%	68.3%	20.2%
spring-boot	6,180	1,161.90 ± 48.57	0.0%	t/o	18.7%	24.7%	n/a	n/a	n/a
tachyon	2,126	2,299.54 ± 194.84	3.0%	17.8%	35.1%	36.7%	40.1%	t/o	n/a
togglz	400	174.70 ± 5.98	4.8%	36.6%	34.0%	35.2%	7.4%	10.2%	n/a
undertow	725	180.29 ± 3.94	4.6%	18.2%	11.2%	0.1%	n/a	n/a	n/a
wro4j	1,280	153.18 ± 6.19	1.5%	26.9%	29.6%	33.6%	6.9%	24.3%	8.1%
zxing	415	114.60 ± 2.21	2.8%	16.2%	36.6%	37.3%	n/a	n/a	425.8%
Average		626.03 ± 83.61	4.6%	39.0%	32.9%	58.2%	35.2%	50.0%	79.0%

tracking); we report only the time spent running the test suite with coverage tracking (otherwise, Cobertura would always have over 100% overhead). Cobertura and Clover did not work correctly on many of the projects, typically due to incompatibilities between the tool and various parts of the Java 8 syntax. (We found that these issues were reported before[1].) In one case (spring-boot), we encountered a deadlock while running Ekstazi.

DeFlaker was very fast in nearly every case, with an average slowdown of only 4.6% across all of these projects. The worst performance from DeFlaker (12.2%) occurred in the `assertj-core` project, which had relatively fast test execution (25 seconds): DeFlaker's impact on the actual test-execution time was insignificant, but opening the Git repository and scanning it for changes still required several seconds, contributing to the overhead. DeFlaker performed far better than the other coverage tools, both in `assertj-core` and other projects. Clover showed highly variable performance among the projects, and is unique among the coverage tools compared: it adds coverage instrumentation to program source code and not to bytecode.

In summary, the results show two important points. First, DeFlaker has a relatively low overhead, 4.5% on average, compared to test runs without any coverage tool, low enough that we believe it can be "always on" in many projects. Even if DeFlaker were only run on failing tests, its 4.5% overhead is still low compared with the cost of rerunning failing tests in isolation. Second, the overhead of DeFlaker is substantially lower than the overhead of the other coverage tools. We expected DeFlaker to have a lower overhead because it collects less information, but we did not expect it to be that much lower than the very mature JaCoCo (originally released in 2009) or Cobertura (originally released in 2005). In particular, Clover was a commercial product, originally released in 2006 and sold and maintained as one until 2017 when it was released as open-source software. Finally, we did not yet optimize the performance of DeFlaker and believe we can improve it further, e.g., using techniques such as smart insertion of coverage tracking based on control-flow graph analysis [79].

[1] https://github.com/cobertura/cobertura/issues/166, https://jira.atlassian.com/browse/CLOV-1839

4 THREATS TO VALIDITY

While our experiments show that DeFlaker can detect flaky tests with low overhead, there are threats to generalizing our result.

External: The projects used in our evaluation may not be representative. To alleviate this threat, we consider a large number of projects from different application domains, with different code sizes and number of test classes. We also considered flaky tests with varying characteristics. In experiments in our historical environment, we considered only commits of each project in which there was at least one flaky test. Our results could differ for a different range of commits. We chose this range because it allows us to precisely measure DeFlaker's ability to find known flaky tests. Further, we believe that the number of commits that we tested DeFlaker on (5,966) is reasonable, especially considering the large amount of time needed to run these experiments (over 5 CPU-years).

Internal: The tools we use in our evaluation may have bugs. To increase confidence in our experiments, we use as many tools that are adopted by the open-source community as possible. To test DeFlaker, we compared its coverage reports with those from the stable and popular statement coverage tool JaCoCo.

Construct: To evaluate DeFlaker's ability to detect known flaky tests, we computed the percentage of commits of each project with a known flaky test in which the test did not cover any code

changed by that commit, without regard for the test outcome. The assumption here is that if a flaky test covers some change (which caused it to fail), it will cover that change regardless of whether it passes or fails. We expect that the likelihood of a flaky test covering changed code is independent of it failing, because the pass/fail outcome of the flaky test depends on some source of non-determinism unrelated to the code changes.

Reproducibility: To enable other researchers to reproduce our results and build upon DeFlaker we have taken several steps. We have released DeFlaker under the MIT license on GitHub [28] and have published binaries on Maven Central [74]. We have released a companion page on our DeFlaker website [27], which lists each commit of each project studied, their URLs, as well as each test identified as flaky in our runs.

5 RELATED WORK

Flaky Tests. There have been several recent studies of flaky tests. Luo et al. conducted the first extensive study of flaky tests, studying over 200 flaky tests found from commit logs and issue trackers of 51 Apache projects, categorizing their root causes, common fixes, and ways to manifest them [61]. Many of the flaky tests used in Section 3 came from Luo et al.'s dataset. Palomba and Zaidman studied 18 open-source Java projects, executing each test 10 times to detect flaky tests and then studying automated techniques to repair the flakiness in each test [82]. We used a similar rerun technique to detect whether a test failure was a true failure or due to flakiness. Whereas both of these papers specifically searched projects to seek out flaky tests, we present instead an evaluation of techniques to find flaky tests from regular test failures.

The most common approach to detecting flaky tests is to rerun failing tests, which we call Rerun. Several popular build and test systems provide support for this, such as Google TAP [42, 63], Maven Surefire [77], Android [21], Jenkins [52], and Spring [75]. To the best of our knowledge, TAP (or any other similar system) does not offer the isolated rerun (in a new JVM/process) that we found to be more effective than an in-process rerun. We have shown that the way that each rerun is performed can have a significant impact on the outcome of the test. Our previous work [33] considered using coverage information to identify flaky tests but did not consider differential coverage, which is key to DeFlaker's performance. Also related are a variety of machine-learning techniques for categorizing test failures as false alarms, flaky, or related to a specific change [48, 54]. Such approaches could be used simultaneously with Rerun or DeFlaker.

One approach may be that developers perform regular maintenance, running a tool to identify tests that *might* fail in the future due to flakiness (i.e., can become flaky tests), and then repair those tests proactively [26, 38, 43, 72]. However, these tools can be too expensive to run after every code change and may report too many warnings, hindering their adoption. For instance, specific techniques have been recently proposed for handling order-dependent tests. Zhang et al. propose several methods to detect such tests by rerunning them in different orders [85]. Huo and Clause propose another technique that can detect such tests [51], although their technique was originally proposed to detect brittle assertions (that may cause non-deterministic failures). Bell and Kaiser [25] propose an approach to tolerate the effects of order-dependent tests by isolating them in the same JVM. However, none of these techniques focuses on detecting whether a given test failure is due to a flaky test or not, and none of the techniques handle general case of arbitrary flaky tests.

Change-Impact Analysis. Change-impact analysis (CIA) techniques aim to determine the effects of source code changes, using static, dynamic, or combined analysis [31, 69, 71]. For example, Ren et al. [69] proposed Chianti, a CIA technique that uses static analysis to decompose the difference between two program versions into atomic changes and uses dynamic call graphs to determine the set of tests whose behavior might be affected by these changes. It also uses these call graphs to determine, for each affected test, which subset of changes can affect the test's behavior. Our differential coverage technique also collects coverage but fully dynamically, requires no expensive static analysis or call-graph generation, and has a much lower overhead.

Regression Test Selection. Regression test selection (RTS) techniques determine which tests can be affected by a code change and only run those to speed up regression testing. Many RTS techniques have been proposed [35, 40, 45–47, 70, 86] and are summarized in two literature reviews [30, 81]. Most RTS techniques collect coverage, first for all the tests, and then recollect coverage only for the tests that are run as potentially affected by the code changes. We compared the performance of DeFlaker to the publicly available Ekstazi RTS tool [39]. In some ways, DeFlaker is an extension of residual coverage, a high-level approach to reduce the overhead of program coverage tracking by only tracking code that has not-yet been covered [67]. In our terms, the code changed by a new commit is de-facto not-yet-covered, and hence, tracked.

6 CONCLUSIONS AND FUTURE WORK

Flaky tests can disrupt developers' workflows, because it is difficult to know immediately if a test failure is a true failure or a flake. We presented DeFlaker, an approach and a tool for evaluating whether a test failure is flaky *immediately* after it occurs, with very low overhead. Even if developers still want to rerun their test failures to determine if they are flaky or not, DeFlaker is still useful because it can provide its results immediately after the first failure, rather than requiring tests to be delayed and reran.

We implemented DeFlaker for Java, integrating it with popular build and test tools, and found 87 previously unknown flaky tests in 10 projects, plus 4,846 flakes in old versions of 26 projects. We are interested in exploring other applications of differential coverage. For instance, if a test fails and covers some changed code, reporting the covered changed code may help debugging [69]. Similarly, reporting when a change is not covered may help test augmentation [55]. Our results are very promising, and we plan to continue working with the open-source software community to find flaky tests and encourage the adoption of DeFlaker, which we have released under the MIT license [28].

ACKNOWLEDGMENTS

We thank Traian Şerbănuţă and Grigore Roşu for help in debugging flaky tests using RV-Predict [50], and Alex Gyori for help in debugging flaky tests using NonDex [72]. Darko Marinov's group is supported by NSF grants CCF-1409423, CCF-1421503, CNS-1646305, and CNS-1740916; and gifts from Google and Qualcomm.

REFERENCES

[1] 2008. TotT: Avoiding Flakey Tests. (2008). http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html.

[2] 2017. Achilles Issue Tracker Issue Number 309. (2017). https://github.com/doanduyhai/Achilles/issues/309.

[3] 2017. Achilles Issue Tracker Issue Number 310. (2017). https://github.com/doanduyhai/Achilles/issues/310.

[4] 2017. checkstyle Issue Tracker Issue Number 4664. (2017). https://github.com/checkstyle/checkstyle/issues/4664.

[5] 2017. geoserver Issue Tracker Issue Number 8213. (2017). https://osgeo-org.atlassian.net/browse/GEOS-8213.

[6] 2017. jackrabbit-oak Issue Tracker Issue Number 6512. (2017). https://issues.apache.org/jira/browse/OAK-6512.

[7] 2017. jackrabbit-oak Issue Tracker Issue Number 6524. (2017). https://issues.apache.org/jira/OAK-6524.

[8] 2017. JMeter ConcurrencyThreadGroupTest::testFlow flaky test failure. (2017). https://groups.google.com/forum/#!topic/jmeter-plugins/Fxg2ojVuxBs.

[9] 2017. killbill Issue Tracker Issue Number 769. (2017). https://github.com/killbill/killbill/issues/769.

[10] 2017. nutz Issue Tracker Issue Number 1283. (2017). https://github.com/nutzam/nutz/issues/1283.

[11] 2017. presto Issue Tracker Issue Number 8374. (2017). https://github.com/prestodb/presto/issues/8374.

[12] 2017. presto Issue Tracker Issue Number 8491. (2017). https://github.com/prestodb/presto/issues/8491.

[13] 2017. presto Issue Tracker Issue Number 8492. (2017). https://github.com/prestodb/presto/issues/8492.

[14] 2017. presto Issue Tracker Issue Number 8493. (2017). https://github.com/prestodb/presto/issues/8493.

[15] 2017. presto Issue Tracker Issue Number 8494. (2017). https://github.com/prestodb/presto/issues/8494.

[16] 2017. presto Issue Tracker Issue Number 8666. (2017). https://github.com/prestodb/presto/issues/8666.

[17] 2017. quickml Issue Tracker Issue Number 152. (2017). https://github.com/sanity/quickml/issues/152.

[18] 2017. quickml Issue Tracker Issue Number 154. (2017). https://github.com/sanity/quickml/issues/154.

[19] 2017. togglz Issue Tracker Issue Number 233. (2017). https://github.com/togglz/togglz/issues/233.

[20] 2017. togglz Issue Tracker Issue Number 240. (2017). https://github.com/togglz/togglz/issues/240.

[21] AndroidFlaky 2017. Android FlakyTest annotation. (2017). http://developer.android.com/reference/android/test/FlakyTest.html.

[22] Apache Software Foundation. 2017. Maven Extension API. (2017). http://maven.apache.org/ref/3.5.0/apidocs/org/apache/maven/AbstractMavenLifecycleParticipant.html.

[23] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A Differencing Algorithm for Object-Oriented Programs. In *ASE*.

[24] Atlassian. 2017. Comparison of code coverage tools. (2017). https://confluence.atlassian.com/clover/comparison-of-code-coverage-tools-681706101.html.

[25] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *ICSE*.

[26] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient Dependency Detection for Safe Java Test Acceleration. In *ESEC/FSE*.

[27] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2017. DeFlaker Companion Website. (2017). http://www.deflaker.org/icsecomp.

[28] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2017. DeFlaker source code. (2017). https://github.com/gmu-swe/deflaker.

[29] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *MSR*.

[30] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. *Informatica* (2011).

[31] Shawn A Bohner. 1996. Software change impact analysis. (1996).

[32] Ahmet Celik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression Test Selection Across JVM Boundaries. In *ESEC/FSE*.

[33] Lamyaa Eloussi. 2015. *Determining Flaky Tests from Test Failures.* Master's thesis. University of Illinois at Urbana-Champaign, Urbana, IL.

[34] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. *I&ST-J* (2010).

[35] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical evaluations of regression test selection techniques: a systematic review. In *ESEM*.

[36] FlakinessDashboardHOWTO 2017. Flakiness Dashboard HOWTO. (2017). http://www.chromium.org/developers/testing/flakiness-dashboard.

[37] Martin Fowler. 2011. Eradicating Non-Determinism in Tests. (2011). http://martinfowler.com/articles/nonDeterminism.html.

[38] Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical Test Dependency Detection. In *ICST*.

[39] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *ISSTA*.

[40] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 2001. An empirical study of regression test selection techniques. *TOSEM* (2001).

[41] Marco Guarnieri, Petar Tsankov, Tristan Buchs, Mohammad Torabi Dashti, and David Basin. 2017. Test Execution Checkpointing for Web Applications. In *ISSTA*.

[42] Pooja Gupta, Mark Ivey, and John Penix. 2011. Testing at the speed and scale of Google. (2011). http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[43] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable Testing: Detecting State-polluting Tests to Prevent Test Dependency. In *ISSTA*.

[44] Dan Hao, Tian Lan, Hongyu Zhang, Chao Guo, and Lu Zhang. 2013. Is This a Bug or an Obsolete Test?. In *ECOOP*.

[45] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression Test Selection for Java Software. In *OOPSLA*.

[46] Mary Jean Harrold, David S. Rosenblum, Gregg Rothermel, and Elaine J. Weyuker. 2001. Empirical Studies of a Prediction Model for Regression Test Selection. *TSE* (2001).

[47] Mary Jean Harrold and Mary Lou Soffa. 1988. An incremental approach to unit testing during maintenance. In *ICSM*.

[48] Kim Herzig and Nachiappan Nagappan. 2015. Empirically Detecting False Test Alarms Using Association Rules. In *ICSE SEIP*.

[49] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *ESEC/FSE*.

[50] Jeff Huang, Patrick Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *PLDI*.

[51] Chen Huo and James Clause. 2014. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *FSE*.

[52] JenkinsRandomFail 2016. Jenkins RandomFail annotation. (2016). https://github.com/jenkinsci/jenkins-test-harness/blob/master/src/main/java/org/jvnet/hudson/test/RandomlyFails.java.

[53] JGitWebPage 2017. JGit Home Page. (2017). http://www.eclipse.org/jgit/.

[54] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What Causes My Test Alarm?: Automatic Cause Analysis for Test Alarms in System and Integration Testing. In *ICSE*.

[55] Wei Jin, Alessandro Orso, and Tao Xie. 2010. Automated Behavioral Regression Testing. In *ICST*.

[56] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *ESEC/FSE*.

[57] F.J. Lacoste. 2009. Killing the Gatekeeper: Introducing a Continuous Integration System. In *Agile*.

[58] Tim Lavers and Lindsay Peters. 2008. *Swing Extreme Testing.*

[59] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*.

[60] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How Does Regression Test Prioritization Perform in Real-world Software Evolution?. In *ICSE*.

[61] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *FSE*.

[62] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: Models, tools, and controlling flakiness. In *ICSE*.

[63] John Micco. 2013. Continuous Integration at Google scale. (2013). http://eclipsecon.org/2013/sites/eclipsecon.org.2013/files/2013-03-24%20Continuous%20Integration%20at%20Google%20Scale.pdf.

[64] John Micco. 2017. The State of Continuous Integration Testing @Google. (2017). https://research.google.com/pubs/pub45880.html.

[65] Kıvanç Muşlu, Bilge Soran, and Jochen Wuttke. 2011. Finding bugs by isolating unit tests. In *ESEC/FSE*.

[66] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *FSE*.

[67] Christina Pavlopoulou and Michal Young. 1999. Residual Test Coverage Monitoring. In *ICSE*.

[68] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. 2012. Understanding Myths and Realities of Test-suite Evolution. In *FSE*.

[69] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *OOPSLA*.

[70] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *TOSEM* (1997).

[71] Barbara G Ryder and Frank Tip. 2001. Change impact analysis for object-oriented programs. In *PASTE*.
[72] August Shi, Alex Gyori, Owolabi Legunsen, and Darko Marinov. 2016. Detecting Assumptions on Deterministic Implementations of Non-deterministic Specifications. In *ICST*.
[73] August Shi, Suresh Thummalapenta, Shuvendu K. Lahiri, Nikolaj Bjorner, and Jacek Czerwonka. 2017. Optimizing Test Placement for Module-level Regression Testing. In *ICSE*.
[74] Sonatype. 2017. Maven Central Repository. (2017). https://search.maven.org.
[75] spring-junit-page 2017. Spring Repeat Annotation. (2017). https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/annotation/Repeat.html.
[76] Pavan Sudarshan. 2012. No more flaky tests on the Go team. (2012). http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team.
[77] SurefireRerun 2017. Surefire rerunFailingTestsCount Option. (2017). http://maven.apache.org/surefire/maven-surefire-plugin/examples/rerun-failing-tests.html.
[78] The Eclipse Foundation. 2017. Eclipse Java Development Tools (JDT). (2017). http://www.eclipse.org/jdt/.
[79] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2002. Efficient Instrumentation for Code Coverage Testing. In *ISSTA*.
[80] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. 2015. An empirical study of bugs in test code. In *ICSME*.
[81] Shin Yoo and Mark Harman. 2012. Regression Testing Minimization, Selection and Prioritization: A Survey. *STVR* (2012).
[82] Andy Zaidman and Fabio Palomba. 2017. Does Refactoring of Test Smells Induce Fixing Flaky Tests?. In *ICSME*.
[83] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *ICSE*.
[84] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing Failure-Inducing Program Edits based On Spectrum Information. In *ICSM*.
[85] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanc Muslu, Michael Ernst, and David Notkin. 2014. Empirically Revisiting the Test Independence Assumption. In *ISSTA*.
[86] Jiang Zheng, Brian Robinson, Laurie Williams, and Karen Smiley. 2006. Applying Regression Test Selection for Cots-Based Applications. In *ICSE*.
[87] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build?: Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *ICSE-SEIP*.