An Empirical Study of Bugs in Test Code

Arash Vahabzadeh Amin Milani Fard Ali Mesbah
University of British Columbia
Vancouver, BC, Canada
{arashvhb, aminmf, amesbah}@ece.ubc.ca

***Abstract*—Testing aims at detecting (regression) bugs in production code. However, testing code is just as likely to contain bugs as the code it tests. Buggy test cases can silently miss bugs in the production code or loudly ring false alarms when the production code is correct. We present the first empirical study of bugs in test code to characterize their prevalence and root cause categories. We mine the bug repositories and version control systems of 211 Apache Software Foundation (ASF) projects and find 5,556 test-related bug reports. We (1) compare properties of test bugs with production bugs, such as active time and fixing effort needed, and (2) qualitatively study 443 randomly sampled test bug reports in detail and categorize them based on their impact and root causes. Our results show that (1) around half of all the projects had bugs in their test code; (2) the majority of test bugs are false alarms, i.e., test fails while the production code is correct, while a minority of these bugs result in silent horrors, i.e., test passes while the production code is incorrect; (3) incorrect and missing assertions are the dominant root cause of silent horror bugs; (4) semantic (25%), flaky (21%), environment-related (18%) bugs are the dominant root cause categories of false alarms; (5) the majority of false alarm bugs happen in the exercise portion of the tests, and (6) developers contribute more actively to fixing test bugs and test bugs are fixed sooner compared to production bugs. In addition, we evaluate whether existing bug detection tools can detect bugs in test code.**

***Index Terms*—Bugs, test code, empirical study**

I. Introduction

Testing has become a wide-spread practice among practitioners. Test cases are written to verify that production code functions as expected. Test cases are also used as regression tests to make sure previously working functionality still works, when the software evolves. Since test cases are code written by developers, they may contain bugs themselves. In fact, it is stated [22] and believed by many software practitioners [11], [18], [30] that "*test cases are often as likely or more likely to contain errors than the code being tested*".

Buggy tests can be divided into two broad categories [11]. First, a fault in test code may cause the test to miss a bug in the production code (*silent horrors*). These bugs in the test code can cost at least as much as bugs in the production code, since a buggy test case may miss (regression) bugs in the production code. These test bugs are difficult to detect and may remain unnoticed for a long period of time. Second, a test may fail while the production code is correct (*false alarms*). While this type of test bugs is easily noticed, it can still take a considerable amount of time and effort for developers to figure out that the bug resides in their test code rather than their production code. Figure 1 illustrates different scenarios of fixing these test bugs.

Although the reliability of test code is as important as production code, unlike production bugs [26], test bugs have not received much attention from the research community thus far. This work presents an extensive study on test bugs that characterizes their prevalence, impact, and main cause categories. To the best of our knowledge, this work is the first to study general bugs in test code.

We mine the bug report repository and version control systems of the Apache Software Foundation (ASF), containing over 110 top-level and 448 sub open-source projects with different sizes and programming languages. We manually inspect and categorize randomly sampled test bugs to find the common cause categories of test bugs.

Our work makes the following main contributions:

- We mine 5,556 unique fixed bug reports reporting test bugs by searching through the bug repository and version control systems of the Apache projects;
- We systematically categorize a total of 443 test bugs into multiple bug categories;
- We compare test bugs with production bugs in terms of the amount of attention received and fix time.
- We assess whether existing bug detection tools such as FindBugs can detect test bugs.

Our results show that (1) around half of the Apache Software Foundation projects have had bugs in their test code; (2) the majority (97%) of test bugs result in false alarms, and their dominant root causes are "Semantic Bugs" (25%), "Flaky Tests" (21%), "Environmental Bugs" (18%), "Inappropriate Handling of Resources" (14%), and "Obsolete Tests" (14%); (3) a minority (3%) of test bugs reported and fixed pertain to silent horror bugs with "Assertion Related Bugs" (67%) being the dominant root cause; (4) developers contribute more actively to fixing test bugs and test bugs require less time to be fixed.

The results of our study indicate that test bugs do exist in practice and their bug patterns, though similar to that of production bugs, differ noticeably, which makes current bug detection tools ineffective in detecting them. Although current bug detection tools such as FindBugs and PMD do have a few simple rules for detecting test bugs, we believe that this is not sufficient and there is a need for extending these rules or devising new bug detection tools specifically geared toward test bugs.

978-1-4673-7532-0/15/$31.00

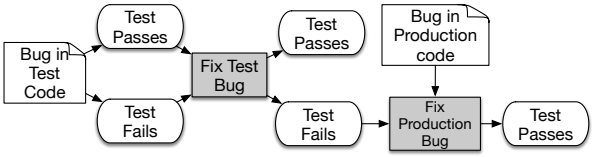


Fig. 1: Different scenarios for fixing test and production bugs.

II. Methodology

The goal of our work is to gain an understanding of the prevalence and categories of bugs in test code. We conduct quantitative and qualitative analyses to address the following research questions:

RQ1: How prevalent are test bugs in practice?
RQ2: What are common categories of test bugs?
RQ3: Are test bugs treated differently by developers compared to production bugs?
RQ4: Are current bug detection tools able to detect test bugs?

All our empirical data is available for download [3].

A. Data Collection

Figure 2 depicts an overview of our data collection, which is conducted in two main different steps, namely, mining bug repositories for test-related bug reports (A), and analyzing commits in version control systems (B and C).

1) Mining Bug Repositories: One of the challenges in collecting test bug reports is distinguishing between bug reports for test code and production code. In fact, most search and filtering tools in current bug repository systems do not support this distinction. In order to identify bug reports reporting a test bug, we selected the JIRA bug repository of the Apache Software Foundation (ASF) since its search/filter tool allows us to specify the type and component of reported issues. We mine the ASF JIRA bug repository [2], which contains over 110 top-level and 448 sub open-source projects, with various sizes and programming languages.

We search the bug repository by selecting the *type* as "Bug", *component* as "test", and *resolution* as "Fixed".

Type. The ASF JIRA bug report types can be either "Bug", "Improvement", "New Feature", "Test", or "Task". However, we observed that most of the reported test-related bugs have "Bug" as their type. The "Test" label is mainly used when someone is contributing extra tests for increasing coverage and testing new features.

Component. The ASF bug repository defines components for adding structure to issues of a project, classifying them into features, modules, and sub-projects [23]. We observed that many projects in ASF JIRA use this field to distinguish different modules of the project. Specifically, they use "test" for the component field to refer to issues related to test code.

Resolution. We only consider bug reports with resolution "Fixed" because if a reported bug is not fixed, it is difficult to verify that it is a real bug and analyze its root causes.

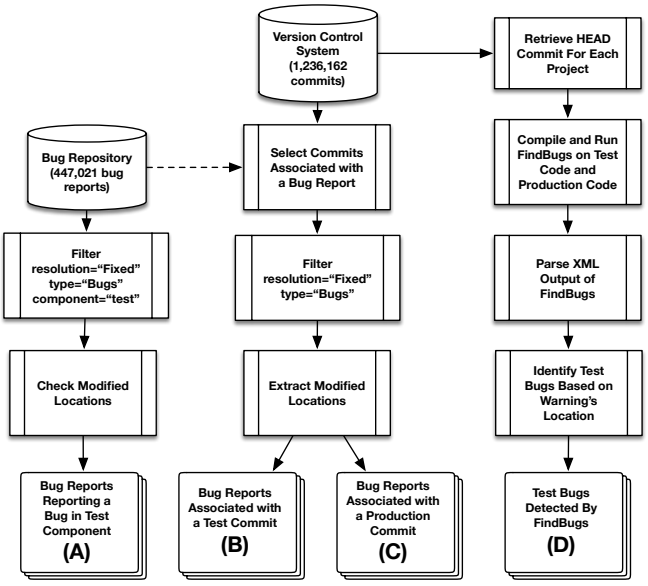


Fig. 2: Overview of the data collection phase.

2) Analyzing Version Control Commits: Since our search query used on the bug repository is restrictive, we might miss some test bugs. Therefore, we augment our data by looking into commits of the version control systems of the ASF projects, Similar to [21]. We use the read-only Git mirrors of the ASF codebases [1], which "contain full version histories (including branches and tags) from the respective source trees in the official Subversion repository at Apache"; thus using these mirrors does not threaten the validity of our study. We observed that most commits associated with a bug report mention the bug report ID in the commit message. Therefore, we leverage this information to distinguish between bug reports reporting test bugs and production bugs. We extract test bugs through the following steps:

Finding Commits with Bug IDs. We clone the Git repository of each Apache project and use JGIT [6] to traverse the commits. In the ASF bug repository, every bug report is identified using an ID composed of `{PROJECTKEY}-#BUGNUM` where `PROJECTKEY` is a project name abbreviation. Using this pattern, we search in the commit messages to find if a commit is associated with a bug report in JIRA. Once we have the ID, we can seamlessly retrieve the data regarding the bug report from JIRA.

Identifying Test Commits. For each commit associated with a bug report, we compute the `diff` between that commit and its parents. This enables us to identify files that are changed by the commit, which in turn allows us to identify *test commits*, i.e., commits that only change files located in the test directory of a project. We refer to commits that change at least one file outside test directories[1] as *production commits*. If a project is using Apache Maven, we automatically extract information about its test directory from the `pom.xml` file. Otherwise, we

[1]We ignored auxiliary files such as `.gitignore` and `*.txt`.

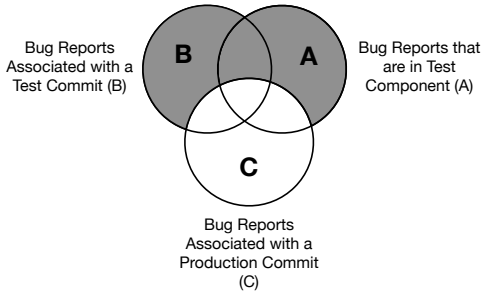


Fig. 3: Bug reports collected from bug repositories and version control systems. $|(A \cup B) - C| = 5,556$ test bug reports in total ($|B - A - C| = 3,849$, $|A - C| = 1,707$).

consider any directory with "test" in its name as a test directory; we also manually verify that these are test directories.

This phase resulted in two sets of bug reports, namely (1) those associated with a test commit (block B in Figure 2), and (2) those associated with a production commit (block C in Figure 2). Since a bug report can be associated with both test and production commits, in our analysis we only consider bug reports that are associated with test commits but not with any production commit (set $B - C$ in the venn diagram of Figure 3).

B. Test Bug Categorization

To find common categories of test bugs (RQ2), we manually inspect the test bug reports. Manual inspection is a time consuming task; on average, it took us around 12 minutes per bug report to study the comments, patches, and source code of any changed files. Therefore, we decided to sample the mined test-related bug reports from our data collection phase.

Sampling. We computed the union of the bug reports obtained from mining the bug reports (subsubsection II-A1) and the version control systems (subsubsection II-A2). This union is depicted as a grey set of $(A \cup B) - C$ in the venn diagram of Figure 3. We randomly sampled $\approx 9.0\%$ of the *unique* bug reports from this set.

Categorization. For the categorization phase, we leverage information from each sampled bug report's description, discussions, proposed patches, fixing commit messages, and changed source code files.

First, we categorize each test bug in one of the two main impact classes, of *false alarms*, i.e., test fails while the production code is correct, or *silent horrors*, i.e, test passes while the production code is or could be incorrect. We adopt the terms false alarms and silent horrors coined by Cunningham [11].

Second, we infer common cause categories while inspecting each bug report. When three or more test bugs exhibited a common pattern, we added a new category. Subcategories also emerged to further subdivide the main categories.

Finally, we also categorize test bugs with respect to the location (in the test case) or unit testing phase in which they occur as follows:

1) **Setup.** Setting up the test fixture, e.g., creating required files, entries in databases, or mock objects.
2) **Exercise.** Exercising the software under test, e.g., by instantiating appropriate object instances, calling their methods, or passing method arguments.
3) **Verify.** Verifying the output or changes made to the states, files, or databases of the software under test, typically through test assertions.
4) **Teardown.** Tearing down the test, e.g., closing files, database connections, or freeing allocated memories for objects.

The categorization step was a very time consuming task and was carried out through several iterations to refine categories and subcategories; the manual effort for these iterations was more than 400 hours, requiring more than 100 hours for each iteration.

C. Test Bug Treatment Analysis

To answer RQ3, we measure the following metrics for each bug report:

Priority: In JIRA, the priority of a bug report indicates its importance in relation to other bug reports. For the Apache projects we analyzed, this field had one of the following values: *Blocker*, *Critical*, *Major*, *Minor* or *Trivial*. For statistical comparisons, we assign a ranking number from 5 to 1 to each, respectively.

Resolution time: The amount of time taken to resolve a bug report starting from its creation time.

Number of unique authors: Number of developers involved in resolving the issue (based on their user IDs).

Number of comments: Number of comments posted for the bug report. It captures the amount of discussions between developers.

Number of watchers: Number of people who receive notifications; an indication of the number of people interested in the fate of the bug report.

We collected these metrics for all the test bug reports and all the production bug reports, separately. For the comparison analysis, we only included projects that had at least one test bug report. To obtain comparable pools of data points, the number of production bug reports that we sampled, were the same as the number of test bug reports mined from each project.

D. FindBugs Study

To answer RQ4, we use FindBugs [17], a popular static bytecode analyzer in practice for detecting common patterns of bugs in Java code. We investigate its effectiveness in detecting bugs in test code.

1) Detecting Bugs in Tests: We run FindBugs (v3.0.0) [4] on the test code as well as the production code of latest version of Java ASF projects that use Apache Maven (see Figure 2 (D)). Compiling projects that do not use Maven requires much manual effort, for instance in resolving dependencies on third party libraries. Also we noticed that FindBugs crashes while running on some of the projects. In total, we were able to successfully run FindBugs on 129 of the 448 ASF sub-projects.

2) Analysis of Bug Patterns Found by FindBugs: We parse the XML output of FindBugs and extract patterns from the reported bugs. FindBugs statically analyzes byte code of Java programs to detect simple patterns of bugs in the byte code. This is done by applying static analysis techniques such as control and data flow analyses. Among patterns of bugs that FindBugs detects, we only considered reported **Correctness** and **Multithreaded Correctness** as others, such as *internationalization*, *bad practice*, *security* or *performance*, are more related to non-functional bugs.

3) Effectiveness in Detecting Test Bugs: To evaluate FindBugs' effectiveness in detecting test bugs, we choose a similar approach used by Couto et al. [10]. We sample 50 bug reports from projects that we can compile the version containing the bug, just before the fix. By comparing the versions before and after a fix, we are able to identify the set of methods that are changed as part of the fix. We run FindBugs on the version before and after the fix to see if FindBugs is able to detect the test bug and could have potentially prevented it. If FindBugs reports any warning in any of the methods changed by the fix and these warnings disappear after the fix, we assume that FindBugs is able to detect the associated test bug.

The next four sections present the results of our study for each research question, subsequently.

III. Prevalence of Test Bugs

Overall, our analysis reveals that 47% of the ASF subprojects (211 out of 448) have had bugs in their tests. Our search query on the JIRA bug repository retrieved 2,040 bug reports. After filtering non-test related reports, we obtained 1,707 test bug reports, shown as $A - C$ in the venn diagram of Figure 3. The search in version control systems resulted in 4,982 bug reports associated only with test commits, depicted as the set $B - C$ in Figure 3. In total, we found 5,556 unique test bug reports ($(A \cup B) - C$). Table I presents descriptive statistics for the number of test bug reports and Table II shows the top 10 ASF projects in terms of the number of test bug reports we found in their bug repository[2].

TABLE I: Descriptive statistics of test bug reports.

Min	Mean	Median	σ	Max	Total
0	12.4	0	48.3	614	5,556

Finding 1*: Around half of all the projects analyzed had bugs in their test code that were reported and fixed. On average, there were 12.4 fixed test bugs per project.*

IV. Categories of Test Bugs

We manually examined the 499 sampled bug reports; 56 of these turned out to be difficult to categorize due to a lack of sufficient information in the bug report. We categorized the remaining 443 bug reports. Table III shows the main categories and their subcategories that emerged from our manual analysis.

[2]Source lines of code is for all programming languages used in project, measured with CLOC: http://cloc.sourceforge.net

TABLE II: Top 10 ASF projects sorted by the number of reported test bugs.

Project	Production Code KLOC	Test Code KLOC	# Test Bug Reports
Derby	386	370	614
HBase	587	195	440
Hive	836	124	295
Hadoop HDFS	101	57	286
Hadoop Common	1249	380	279
Hadoop Map/Reduce	60	24	231
Accumulo	405	78	187
Qpid	553	93	152
Jackrabbit Content Repository	247	107	145
CloudStack	1361	228	111

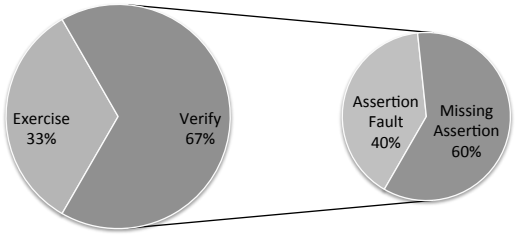


Fig. 4: Distribution of silent horror bug categories.

Our results show that a large number of reported test bugs result in a test failure (97%), and a small fraction pertains to silent test bugs that pass (3%).

A. Silent Horror Test Bugs

Silent test bugs that pass are much more difficult to detect and report compared to buggy tests that fail. Hence, it is not surprising that only about 3% of the sampled bug reports (15 out of 443) belong to this category.

Figure 4 depicts the distribution of silent horror bug categories in terms of the location of the bug. In five instances, the fault was located in the exercise step of the test case, i.e., the fault caused the test not to execute the SUT for the intended testing scenario, which made the test useless. For instance, as reported in bug report `JCR-3472`, due to a fault in the test code of the Apache Jackrabbit project, queries in `LargeResultSetTest` run against a session where the test content is not visible and thus the resulting set is empty and the whole test is pointless. In another example, due to the test dependency between two test cases, one of test cases "is actually testing the GZip compression rather than the DefaultCodec due to the setting hanging around from a previous test" (`FLUME-571`). Such issues could explain why these bugs remain unnoticed and are difficult to detect.

The other 10 instances were located in the verification step, i.e., they all involved test assertions. From these, six pertained to a missing assertion and four were related to faults in the assertions, which checked a wrong condition or variable.

Interestingly, two of the silent test bugs resulted in a failure when they were fixed, indicating a bug in the production code that was silently ignored. For example, in `ACCUMULO-1878`, `1927`, `1988` and `1892`, since the test did not check the

TABLE III: Test bug categories for false alarms.

Category	Subcategory	Description
Semantic Bugs	**S1.** Assertion Fault	Fault in the assertion expression or arguments of a test case.
	S2. Wrong Control Flow	Fault in a conditional statement of a test case.
	S3. Incorrect Variable	Usage of the wrong variable.
	S4. Deviation from Test Requirement and Missing Cases	A missing step in the exercise phase, missing a possible scenario, or when test case deviates from actual requirements.
	S5. Exception Handling	Wrong exception handling.
	S6. Configuration	Configuration file used for testing is incorrect or test does not consider these configurations.
	S7. Test Statement Fault or Missing Statements	A statement in a test case is faulty or missing.
Environment	**E1.** Differences in Operating System	Tests in this category pass on one OS but fail on another one.
	E2. Differences in third party libraries or JDK versions and vendors	Failure is due to incompatibilities that exist between different versions of JDK or different implementations of JDK by different vendors, or different versions of third party libraries.
Resource Handling	**I1.** Test Dependency	Running one test affects the outcome of other tests.
	I2. Resource Leak	A test does not release its acquired resources properly.
Flaky Tests	**F1.** Asynchronous Wait	Test failure is due to an asynchronous call and not waiting properly for the result of the call.
	F2. Race Condition	Test failure is due to non-deterministic interactions of different threads.
	F3. Concurrency Bugs	Concurrency issues such as deadlocks and atomicity violations.
Obsolete tests	**O1.** Obsolete Statements	Statements in a test case are not evolved when production code has evolved.
	O2. Obsolete Assertions	Assertion statements are not evolved as production code evolves.

```
1  -for (int j = 0; i < cr.getFiles().size(); j++) {
2  +for (int j = 0; j < cr.getFiles().size(); j++) {
3   assertTrue(cr.getFiles().get(j)
4   .getReader().getMaxTimestamp() < (System.↩
        currentTimeMillis() - this.store.getScanInfo()↩
        .getTtl()));
```

Fig. 5: An example of a silent horror test bug due to a fault in `for` loop.

```
1  try {
2      nsp.unregisterNamespace("NotCurrentlyRegistered↩
           ");
3  +   fail("Trying to unregister an unused prefix ↩
        must fail");
4  } catch (NamespaceException e) {
5      // expected behaviour
6  }
```

Fig. 6: An example of a silent horror test bug due to a missing assertion.

return value of the executed M/R jobs, these jobs were failing silently (`ACCUMULO-1927`), when this was fixed, the test failed. Figure 5 shows the fixing commit for `HBASE-7901`, a bug in the `for` loop condition that caused the test not to execute the assertion.

Although JUnit 4 permits to assert a particular exception through the `expected` annotation and `ExpectedException` rule, many testers are used to or prefer [13] using the traditional combination of `try/catch` and `fail()` assertion type to achieve this goal. However, this pattern tends to be error-prone. In our sampled list, four out of 15 silent bugs involved incorrect usage of the `try/catch` and in combination with the `fail()` primitive. For example, Figure 6 shows the fixing commit for the bug report `JCR-500`; the test needs to assert that unregistering a namespace that is not registered should throw an exception. However, a `fail()` assertion is missing from the code making the whole test case ineffective. Another pattern of this type of bug is when the SUT in the `try` block can throw multiple exceptions and the tester does not assert on the type of the thrown exception. It is worth mentioning that two of these 15 bugs could have potentially been detected statically; in one case (`ACCUMULO-828`), the whole test case did not have any assertions, and in another (`SLIDER-41`) a number of test cases were not executed because they did not comply with the test class name conventions of Maven, i.e., their name did not start with "Test".

***Finding 2**: Silent horror test bugs form a small portion (3%) of reported test bugs. Assertion-related faults are the dominant root cause of silent horror bugs.*

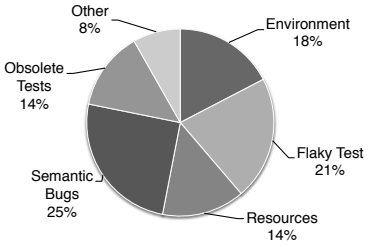
B. False Alarm Test Bugs

We categorized the 428 bug reports that were false alarms based on their root cause. We identified five major causes for false alarms. Figure 7 shows the distribution for each main category and also testing phase in which false alarm bug occurred.

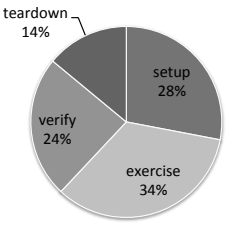
***Finding 3**: Semantic bugs (25%) and Flaky tests (21%) are the dominant root causes of false alarms, followed by Environment (18%) and Resource handling (14%) related causes. The majority of false alarm bugs occur in the exercise phase of testing.*

1) Semantic Bugs: This category consists of 25% of the sampled test bugs. Semantic bugs reflect inconsistencies between specifications and production code, and test code. Based on our observations of common patterns of these bugs, we categorized them into seven subcategories as shown in Table III. Figure 8a presents percentages of each subcategory, and Figure 9a shows the fault location distribution in the testing phase.

The majority of test bugs in this category (33%) belongs to tests that miss a case or deviate from test requirements (S4). Examples include tests that miss setting some required properties of the SUT (e.g., `CLOUDSTACK-2542` and `MYFACES-1625`), or tests that miss a required step to exercise the SUT correctly (e.g., `HDFS-824`). Test statement faults or missing statements account for 19% of bugs in this category. For example in `CLOUDSTACK-3796`, a statement fault resulted in ignoring to set the attributes needed for setting up the test



(a) Distribution based on bug categories.



(b) Distribution based on testing phases.

Fig. 7: Distribution of false alarms.

correctly, thus resulting in a failure. The use of an incorrect variable, which may result in asserting the wrong variable (e.g., `DERBY-6716`) or a wrong test behaviour was observed in 9% of the semantic bugs. 7% of semantic bugs in our sampled bugs were due to improper exception handling in test code, which resulted in false test failures (e.g., `JCR-505`). Some tests require reading properties from an external configuration file to run with different parameters without changing the test code itself; however, some tests did not use these configurations properly or in some other cases these configurations were buggy themselves. 7% of the false alarm bugs had this issue. We categorized a bug in the *wrong control flow* category if the test failed due to a fault in a conditional statement (e.g., `if`, `for` or `while` conditional). 5% of semantic bugs belong to this category. Another 5% of semantic bugs were due to faulty assertions (e.g., `JCR-503`).

Finding 4*: Deviations from test requirements or missing cases in exercising the SUT (33%) and faulty or missing test statements (19%) are the most prevalent semantic bugs in test code.*

2) Environment: Around 18% of bug reports pertained to a failing test due to environmental issues, such as differences in path separators in Windows and Unix systems. In this case, tests pass under the environment they are written in, but fail when executed in a different environment. Since open source software developers typically work in diverse development environments, this category accounts for a large portion of the test bug reports filed.

Figure 8b and Figure 9b show the distribution of environmental bugs and their fault locations. About 61% of the bug reports in this category were due to operating system differences (E1), and particularly differences between the Windows and Unix operating systems. Testers make platform-specific assumptions that may not hold true in other platforms — e.g., assumptions about file path and classpath conventions, order of files in a directory listing, and environment variables (`MAPREDUCE-4983`). Some of the common causes we observed that result in failing tests in this category include: (1) Differences in path conventions — e.g., Windows paths are not necessarily valid URIs while Unix paths are, or Windows uses quotation for dealing with spaces in file names but in Unix spaces should be escaped (`HADOOP-8409`). (2) File system differences — e.g., in Unix one can rename, delete, or move an opened file while its file descriptor remains pointing to a proper data; however, in Windows opened files are locked by default and cannot be deleted or renamed (`FLUME-349`). (3) File permission differences — e.g., default file permission is different on different platforms. (4) Platform-specific use of environmental variables — e.g., Windows uses the `%ENVVAR%` and Unix uses the `$ENVVAR` notations to retrieve environmental variable values (`MAPREDUCE-4869`). Also `classpath` entries are separated by `';'` in Windows and by `':'` in Unix.

Differences in JDK versions and vendors (E2) were responsible for 26% of environment related test bugs. For example, with IBM JDK developers should use `SSLContext.getInstance(''SSL_TLS")` instead of "`SSL`" in Oracle JDK, to ensure the same behaviour (`FLUME-2441`). There is also compatibility issues between different versions of JDKs, e.g., testers depended on the order of iterating a `HashMap`, which was changed in IBM JDK 7 (`FLUME-1793`).

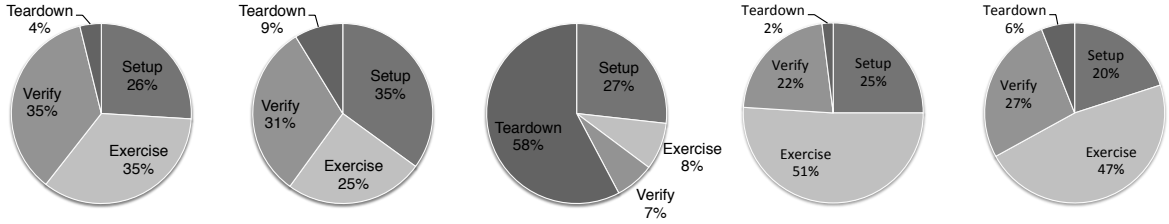
Finding 5*: 61% of environmental false alarms are platform-specific failures, caused by operating system differences.*

3) Inappropriate Handling of Resources: Ideally, test cases should be independent of each other, however, in practice this is not always true, as reported in a recent empirical study [33]. Around 14% of bug reports (61 out of 428) point to inappropriate handling of resources, which may not cause failures on their own, but cause other dependent tests to fail when those resources are used. Figure 8c shows the percentage for sub-categories of resource handling bugs and Figure 9c shows the distribution of testing phases in which the fault occurs. About 61% of these bugs were due to test dependencies.

A good practice in unit testing is to mitigate any side-effects a test execution might have; this includes releasing locally used resources and rolling back possible changes to external resources such as databases. Most of unit testing frameworks provide opportunities to clean up after a test run, such as the `tearDown` method in JUnit 3 or methods annotated with `@After` in JUnit 4. However, testers might forget or fail to perform this clean up step properly. One common mistake is when a test that changes some persistent data (or acquires some resources), conducts the clean up in the test method's body. In this case, if the test fails due to assertion failures, exceptions or time outs, the clean up operation will not take place causing other tests or even future runs of this test case to fail. Figure 10 illustrates this bug pattern and its fix. Another common problem we observed is that testers forgot to call the `super.tearDown()` or `super.setUp()` methods

(a) Semantic bug. (b) Environmental bugs. (c) Resource related. (d) Flaky tests. (e) Obsolete tests.

Fig. 8: Percentage of subcategories of test bugs.



(a) Semantic bug. (b) Environmental bugs. (c) Resource related. (d) Flaky tests. (e) Obsolete tests.

Fig. 9: Test bugs distribution based on testing phase in which bugs occurred.

```
1  @Test
2  public void test(){
3    acquireResources();
4    assertEquals(a,b);
5    releaseResources();
6  }
```

(a) Buggy test.

```
1  @Before
2  public void setUp(){
3    acquireResources();
4  }
5  @Test
6  public void test() {
7    assertEquals(a,b);
8  }
9  @After
10 public void tearDown(){
11   releaseResources();
12 }
```

(b) Fixed test.

Fig. 10: Resource handling bug pattern in test code.

and this prevents their superclass to free acquired resources (`DERBY-5726`). Bug detection tools such as FindBugs can detect these types of test bugs.

Finding 6*: 61% of inappropriate resource handling bugs are caused by dependent tests. More than half of all resource handling bugs occur in the teardown phase of test cases.*

4) Flaky Tests: These test bugs are caused by non-deterministic behaviour of test cases, which intermittently pass or fail. These tests, also known as 'flaky tests' by practitioners, are time consuming for developers to resolve, because they are hard to reproduce [12]. A recent empirical study on flaky tests [21] revealed that the main root causes for flaky tests are *Async Wait*, which happens when a test does not wait properly for a asynchronous call, and *Race Condition*, which is due to interactions of different threads, such as order violations. Our results are also inline with their findings; we found that not waiting properly for asynchronous calls (46%) is the main root cause of flaky tests, followed by race conditions between different threads (Figure 8d). As shown by Figure 9d, most of flaky tests (51%) are due to bugs in exercise phase of tests.

Finding 7*: The majority of flaky test bugs occur when the test does not wait properly for asynchronous calls during the exercise phase of testing.*

5) Obsolete Tests: Ideally, test and production code should evolve together, however, in practice this is not always the case [32]. An obsolete test [15] is a test case that is no longer valid due to the evolution of the specifications and production code of the program under test. Obsolete tests check features that have been modified, substituted, or removed. When an obsolete test fails, developers spend time examining recent changes made to production code as well as the test code itself to figure out that the failure is not a bug in production code.

As shown in Figure 9e, developers mostly need to update the exercise phase of obsolete tests. This is expected as adding new features to production code may change the steps required to execute the SUT, however, may not change the expected correct behaviour of the SUT, i.e., assertions. In fact, as depicted in Figure 8e, only 23% of obsolete tests required a change to assertions.

Finding 8*: The majority of obsolete tests require modifications in the exercise phase of test cases, and mainly in normal statements (77%) rather than assertions.*

TABLE IV: Test code warnings detected by FindBugs.

Bug Description	Bug Category	Percentage
Inconsistent synchronization	Flaky	29.8%
Possible null pointer dereference in method on exception path	Semantic	17.6%
Using pointer equality to compare different types	Semantic	8.8%
Possible null pointer dereference	Semantic	7.3%
Class defines field that masks a superclass field	Semantic	3.9%
Nullcheck of value previously dereferenced	Semantic	2.9%
An increment to a volatile field isn't atomic	Flaky	2.9%
Method call passes null for nonnull parameter	Semantic	2.4%
Incorrect lazy initialization and update of static field	Flaky	2.4%
Null value is guaranteed to be dereferenced	Semantic	2.0%

TABLE V: Comparison of test and production bug reports.

Metric	Type	Med	Mean	SD	Max	d	OR	p-value
Priority	PR	3.00	2.91	0.76	5.00	-0.13	0.78	4.9e-14
	TE	3.00	2.80	0.75	5.00			
Resolution Time(days)	PR	6.39	109.70	282.04	2843.56	-0.20	0.69	<2.2e-16
	TE	2.77	58.97	213.72	2666.60			
#Comments	PR	3.00	4.91	6.74	101.00	0.15	1.31	
	TE	4.00	5.88	6.26	99.00			
#Authors	PR	2.00	2.41	1.53	18.00	0.31	1.77	
	TE	2.00	2.89	1.53	12.00			
#Watchers	PR	0.00	1.32	2.04	24.00	0.25	1.58	
	TE	1.00	1.84	2.06	16.00			

V. Test Bugs vs Production Bugs

Table V shows the median, mean, standard deviation, and maximum of each metric defined in subsection II-C, for test bugs (TE) and production bugs (PR). We used the nonparametric Mann-Whitney U tests to compare the distribution of test bugs with that of production bugs and compute the p-values. The p-value is close to zero because of a large sample size effect; we computed effect size — standardized mean difference (d) and odds ratio (OR) — to compare the meaningfulness of differences. The results indicate that test bugs take less time to be fixed compared to production bugs. Although the priority assigned to test bugs and production bugs have a similar distribution, developers seem to contribute more to fixing test bugs as both median and mean for the number of unique authors, watchers and comments for test bugs are higher than production bugs.

***Finding 9**: On average, developers contribute more actively to fixing test bugs compared to production bugs and test bugs are resolved faster than production bugs.*

VI. FindBugs Study

A. Detected Bugs

FindBugs reported 205 *Correctness* and *Multithreaded Correctness* warnings in the test code of 20 out of 129 ASF projects that we were able to compile and run the tool on. Table VI summarizes descriptive statistics for the number of reported warnings. For additional fine-grained data per project we refer the reader to our online report [3], which we removed from this paper due to space limitations.

TABLE VI: Descriptive statistics of bugs reported by FindBugs.

Min	Mean	Median	σ	Max	Total
0	1.6	0	5.5	48	205

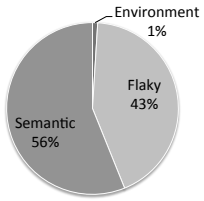


Fig. 11: Distribution of warnings reported by FindBugs.

B. Categories of Test Bugs Detected by FindBugs

Table IV shows the top 10 most frequent potential test bugs detected by FindBugs and their percentage. Figure 11 shows distribution of different categories of these warnings. We consider *Multithreaded Correctness* warnings reported by FindBugs as flaky test bugs and the *Correctness* warnings as semantic bugs, i.e., inconsistencies between actual and intended behaviour of the test. FindBugs has also one rule to detect bugs that cause different behaviours in Linux and Windows due to path separator differences, i.e., environment related bugs.

FindBugs has six test related rules that are part of the included correctness rules. However, FindBugs did not report any warnings related to these categories. As depicted in Figure 11, semantic related warnings and flaky ones are the major warnings reported by FindBugs.

C. FindBugs' Effectiveness in Detecting Test Bugs

Earlier studies [10], [24] report that static code analysis tools are able to detect 5-15 % of general bugs in software projects. We wanted to know how they perform on test bugs. We sampled 50 bug reports out of 623 bug reports that changed Java source files and we were able to compile their project when we checked out the version just before the fix. Among these 50 sampled bug reports, in 3 (6%) instances at least one Findbugs' warning disappeared after the fix. We analyzed these 3 instances manually and found out one of them was actually a false positive and the other two warnings were not directly related to the bug report. This means, FindBugs was not able to detect any of the 50 test bugs.

***Finding 10**: FindBugs could not detect any of the test bugs in our sampled 50 bug reports.*

VII. Discussion

Our study has implications for both testers and developers of bug detection tools. The results of this study imply that test code is susceptible to bugs just like production code (Finding 1). Test code is supposed to guard production code against potential (future) bugs and thus should be bug free itself. However, current bug detection tools are mainly geared towards production code. For instance, FindBugs has only six bug detection patterns dedicated for testing code among its 424 bug patterns [5]. Similarly, PMD, another popular static bug detection tool for Java, has only 12 bug pattern rules for bugs in JUnit [7]. Moreover patterns of environmental bugs, flaky tests, and resource handling bugs in test code differ from production code, making current bug detection tools unable

to detect them in test code. For example the latest version of FindBugs detects run-time error handling mistakes based on the method proposed by [31]. We identified a similar pattern of this bug in test code (Figure 10). However, because of a slight change of pattern in the test code, FindBugs was not able to detect this.

In our study, FindBugs detected an average of 1.6 bugs in the test code of 129 open source projects, most of which fall into semantic and flaky test categories, the most prevalent categories of bug reports (Findings 2 and 4). However, many of the reported bugs cannot be simply detected using current static bug detection tools. This is particularly true for the silent horror bugs, which are mostly due to assertion related faults (Finding 3). Finding automated ways of detecting silent horror test bugs could be of great value to developers, since such bugs are extremely difficult to detect.

Our results show that a large portion of bugs in test code belongs to semantic bugs, i.e., test code does not properly test production code (Finding 4). Any method that can enhance developers' understanding of the requirements, the software under test, and its valid usage scenarios, can help to reduce the number of semantic bugs in test code.

Compared to production bugs, we find that test bugs receive more attention from developers and are fixed sooner. This might be because the majority of the test bugs result in a test failure, which is difficult to ignore for developers. Another explanation could be that bugs in test code might be easier to fix than bugs in production code.

Threats to Validity. An internal validity threat is that the categorization of bug reports was made by two of the co-authors, which may introduce author-bias. To mitigate this, we conducted a review process in which each person reviewed the categorization done by the other person. Regarding the test bugs detected by FindBugs, we did not manually inspect each to see if is indeed a real bug. However, since we chose only the *Correctness* categories of FindBugs, we believe the reported bugs are issues in the test code that need to be fixed.

In terms of external threats, our results are based on bug reports from a number of experimental objects, which calls into question the representativeness. However, we believe that the chosen 448 ASF projects are representative of real-world applications as they vary in domains such as desktop applications, databases and distributed systems, and programming languages such as Java, C++ and Python. In addition, we focus exclusively on bug reports that were *fixed*. This decision was made since the root cause would be difficult to determine from open reports, which have no corresponding fix. Further, a fix indicates that there was indeed a bug in the test code.

VIII. Related Work

Test smells were first studied by van Deursen et al. [27] and later other works defined types of test smells, such as test fixture [14], eager test, and mystery guest, and proposed methods to detect these test smells [17], [9], [28], [29]. Test smells are, however, not bugs. In this study, we focus on bugs that change the intended behaviour of the test.

Zhang et al. [33] found that the test independence assumption does not always hold in practice. They observed that the majority of dependent tests result in false alarm and some of these dependencies result in missed alarms. In this case a test which should reveal a fault passes accidentally because of the environment generated by another dependent test case. Test dependency (I1) is one of the 16 cause subcategories for test bugs emerged in our empirical study.

Lu et al. [20] studied real world concurrency bugs, and found that most of concurrency bugs belong to order or atomicity violations. Luo et al. [21] categorized and investigated the root cause of failures in test cases manifested by non-determinism, known as flaky tests. Flaky tests are one of the main cause categories emerged in our categorization study. Our results are inline with the findings of Lou et al in terms of the root causes of such test bugs.

Li et al. [19] mined the software bug repositories to categorize types of bugs found in production code. Their work is similar to ours in case of categorization but we looked and categorized types of bugs in test code instead of production code. Athanasiou et al. [8] proposed a model to assess test quality based on source code metrics. They showed that there is a high correlation between the test quality as assessed by their model and issue handling performance.

Herzig and Nagappan [16] proposed an approach to identify false alarms. They use association rule learning to automatically identify these false alarms based on patterns learned from failing test steps in test cases that lead to a false alarm. The authors aim at identifying test alarms to prevent development process disruption, since a test failure halts the integration process on the code branch that test failure occurred. Our work, however, aims at providing insights into patterns of faults in test code to help detect them by static analysis tools.

Zaidman et al. [32] investigated how production code and test code co-evolve. They introduced three test co-evolution views, namely change history view, growth history view, and test quality evolution view. It would be interesting to see how test bugs would fit into these views, for instance, are test bugs introduced when they are first added or when they are modified later as test code co-evolves with production code?

IX. Conclusions and Future Work

This work presents the first extensive quantitative and qualitative study of test bugs. Test bugs may cause a test to fail while production code is correct (false alarms), or may cause a test to pass, while the production code is incorrect (silent horrors). Both are costly for developers. Our results show that test bugs are in fact prevalent in practice, the majority are false alarms, and semantic bugs and flaky tests are the dominant root causes of false alarms, followed by environment and resource handling related causes. Our evaluation reveals that FindBugs, a popular bug detection tool, is not effective in detecting test bugs. For future work, we plan to analyze correlations between test bugs and various software metrics, and use the results of this study to design a bug detection tool with test bug patterns capable of detecting bugs in test code.

REFERENCES

[1] Apache projects' Git repositories. http://git.apache.org.

[2] Apache projects supported by Jira. https://issues.apache.org/jira/secure/BrowseProjects.jspa.

[3] An empirical study of bugs in test code. Dataset, additional tables and figures. http://salt.ece.ubc.ca/software/testbugs-study/.

[4] FindBugs - find bugs in Java programs. http://findbugs.sourceforge.net.

[5] Findbugs' rules. http://findbugs.sourceforge.net/bugDescriptions.html.

[6] JGIT library. https://eclipse.org/jgit/.

[7] PMD rules. http://pmd.sourceforge.net/pmd-5.2.3/pmd-java/rules/java/junit.html.

[8] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman. Test code quality and its relation to issue handling performance. *Transactions on Software Engineering*, (11):1100–1125, 2014.

[9] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 56–65, 2012.

[10] C. Couto, J. a. E. Montandon, C. Silva, and M. T. Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Control*, pages 241–257, 2013.

[11] W. Cunningham. Bugs in the tests. http://c2.com/cgi/wiki?BugsInTheTests.

[12] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah. Works for me! Characterizing non-reproducible bug reports. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 62–71. ACM, 2014.

[13] J. Goulding. Be careful when using JUnit's expected exceptions. http://jakegoulding.com/blog/2012/09/26/be-careful-when-using-junit-expected-exceptions/.

[14] M. Greiler, A. van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 322–331, 2013.

[15] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang. Is this a bug or an obsolete test? In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 602–628. Springer-Verlag, 2013.

[16] K. Herzig and N. Nagappan. Empirically detecting false test alarms using association rules. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2015.

[17] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[18] C. Jones. *Programming Productivity.* McGraw-Hill., New York, NY, 1986.

[19] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: An empirical study of bug characteristics in modern open source software. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33. ACM, 2006.

[20] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339. ACM, 2008.

[21] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 643–653. ACM, 2014.

[22] S. McConnell. *Code Complete.* Microsoft Press., Redmond, WA, 1993.

[23] D. Meyer. Organize your JIRA issues with subcomponents. http://blogs.atlassian.com/2013/11/organize-jira-issues-subcomponents/, 2013.

[24] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 99–108. ACM, 2010.

[25] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *International Symposium on Fault-Tolerant Computing*, pages 475–484, 1992.

[26] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[27] A. van Deursen, L. Moonen, A. v. d. Bergh, and G. Kok. Refactoring test code. In *Extreme Programming Perspectives*, pages 141–152. Addison-Wesley, 2002.

[28] B. van Rompaey, B. D. Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

[29] B. van Rompaey, B. Du Bois, and S. Demeyer. Characterizing the relative significance of a test smell. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 391–400, 2006.

[30] R. J. Weiland. *The Programmer's Craft: Program Construction Computer Architecture, and Data Management.* Reston Publishing., Reston, VA, 1983.

[31] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 419–431. ACM, 2004.

[32] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 220–229, 2008.

[33] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 385–396. ACM, 2014.