Reliable Testing: Detecting State-Polluting Tests to Prevent Test Dependency

Alex Gyori, August Shi, Farah Hariri, and Darko Marinov
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{gyori,awshi2,hariri2,marinov}@illinois.edu

ABSTRACT

Writing reliable test suites for large object-oriented systems is complex and time consuming. One common cause of unreliable test suites are test dependencies that can cause tests to fail unexpectedly, not exposing bugs in the code under test but in the test code itself. Prior research has shown that the main reason for test dependencies is the "pollution" of state shared across tests.

We propose a technique, called PolDet, for finding tests that pollute the shared state. In a nutshell, PolDet finds tests that modify some location on the heap shared across tests or on the file system; a subsequent test could fail if it assumes the shared location to have the initial value before the state was modified. To aid in inspecting the pollutions, PolDet provides an access path through the heap that leads to the polluted value or the name of the file that was modified. We implemented a prototype PolDet tool for Java and evaluated it on 26 projects, with a total of 6105 tests. PolDet reported 324 polluting tests, and our inspection found that 194 are relevant pollutions that can easily affect other tests.

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

General Terms: Reliability

Keywords: Flaky Tests, Test Dependency, State Pollution

1. INTRODUCTION

Regression testing is a crucial activity in software development. Developers rely on regression testing to determine whether the newly made code changes break software functionality. If a regression test-suite run produces a failure, developers need to debug it. For a reliable test suite, failures should indicate a problem introduced by the code change. If the problem is indeed in the code under test, then it is highly beneficial that the test suite failed. However, if the problem is in the test code itself, then the test code should be changed.

ISSTA'15 , July 12–17, 2015, Baltimore, MD, USA

One common problem [1, 7, 13, 18, 23, 28] in regression test suites is dependency between tests. These dependencies arise when the tests read and write some shared resource, e.g., the heap state in the main memory, file system, database, etc. Prior research showed that these dependencies occur in various projects (ranging from small projects such as Maven to medium projects such as Apache Aries and to large projects such as Hadoop) [13], and that most dependencies are on the heap state, reported to range from 53% [13] to 61% [28] of all test dependencies. These dependencies make the outcome of regression test-suite runs unreliable: even for the same version of the code under test, the tests could pass when executed in one order but fail when executed in another order [13, 16, 28].

Several research groups have started developing techniques that can combat test dependencies. We discuss related work in Section 7 but highlight two techniques here. Zhang et al. [28] present a technique that can *find existing* test dependencies by running a test suite in various, carefully selected, orders and checking if any order fails. However, their technique requires that the test dependency already be present in the test suite, i.e., it does not proactively find potential test dependencies even before they can manifest. Bell and Kaiser [1] present VMVM, a technique that can *tolerate* the presence of test dependencies by restoring shared heap state, which may have been modified, after each test run. However, their technique does not report whether there is a modification or not; it always restores the state under the assumption that it may have been modified.

The existing techniques do not directly provide the information about the root cause of the dependencies, i.e., do not report which test "pollutes" what part(s) of the shared state. For example, consider a test t that starts from a shared state s, modifies it to s' such that there could be another test t' that would pass when started from s but fail when started from s'. Two issues are important to highlight. First, when the test t' seemingly non-deterministically fails or passes for the same code, the culprit is not necessarily the test t' but the polluting test t, which makes debugging harder.[1]

Second, even if the current test suite does not have any test t' that can be affected by the polluting test t, it is still valuable to know that t is a polluting test, so it could be fixed even before t' is added and the test order is changed. For example, the change in test order significantly affected a number of Java projects when they upgraded to Java 7 [8].

[1]While this paper does not consider fixing of test pollution, a typical fix is either for t to clean the state after it finishes its logic, or for t' to clean the state before it starts its logic.

The reason was that Java 7 changed the Reflection API implementation. Because JUnit uses reflection to find the tests to run, the tests started running in different orders than in previous versions of Java, exposing test dependencies as failing test suites. Some of those test suites were years old, and debugging such old test suites is rather hard as reported by several blog posts [9, 14, 15]. Ideally a polluting test should be caught *right when the developer is about to add it to the test suite* because that is when the developer is in the best position to fix the polluting test, or at least label it as a polluting test that could cause problems in the future.

We introduce PolDet, a technique that detects polluting tests. PolDet *proactively* finds tests that pollute the state, enabling the developers to fix the tests right away, rather than later when the pollution manifests in a test failure. Conceptually, PolDet is rather simple and finds polluting tests "by definition": for each test in a test suite, PolDet captures the shared state (on the heap and the file system) before and after the test, and then compares these two states to determine if there were any *relevant* differences.

To help developers find polluting tests, PolDet has to overcome several challenges. One challenge is to capture and compare the states at the appropriate abstraction level and appropriate program points such that the reported differences are likely to be relevant pollutions. Some state differences are irrelevant, e.g., if states s and s' differ only in the private content of some caches that the test code cannot observe via the public API, then the difference is irrelevant. An additional challenge is to offer information that helps developers in fixing the pollution. The final challenge is to make the technique efficient enough, but it is not the most important: the technique could be run only occasionally for the entire suite, or it could be run only for the newly added tests rather than for all the tests in the test suite. Indeed, our prior study [13] shows that 78% of the polluting tests pollute the shared state right when they are added (i.e., only 22% start polluting due to later changes in the test code or the code under test).

We make the following contributions:

- We formalize the problem of test pollution
- We present the PolDet technique that detects pollutions on shared heap or file system
- We implement a prototype PolDet tool for Java
- We evaluate the tool on 26 projects from GitHub

The experimental results show that PolDet effectively finds polluting tests. In the default configuration, PolDet reported 324 tests (out of 6105 tests) as potential polluting tests, and our inspection found that 194 of those are relevant polluting tests. The runtime overhead of our PolDet prototype is a geometric mean of 4.50x, on a machine representative of a build-farm server. We believe this overhead is acceptable for running PolDet occasionally on the entire test suites and running always on the newly added tests.

2. MOTIVATING EXAMPLE

We next discuss a real example of a polluting test that was added to the Apache Hadoop project [6] at one revision and then created problems in the test suite much later. Figure 1 shows a simplified code snippet from the `TestPathData` class. This snippet includes two tests of interest with

```
1 public class TestPathData {
2   static Path testDir;
3   ...
4   @BeforeClass
5   public static void initialize() {
6     ...
7     testDir = new Path(System.getProperty("test.build.data",
8                          "build/test/data") + "/testPD");
9   }
10  @Test // FT
11  public void testAbsoluteGlob() {
12    PathData[] items = PathData.expandAsGlob(testDir +
13                         "/d1/f1*", conf);
14    assertEquals(
15      sortedString(testDir + "/d1/f1", testDir + "/d1/f1.1"),
16      sortedString(items));
17  }
18  ...
19  @Test // PT
20  public void testWithStringAndConfForBuggyPath() {
21    dirString = "file:///tmp";
22    testDir = new Path(dirString);
23    assertEquals("file:/tmp", testDir.toString());
24    ...
25  }
26 }
```

Figure 1: Hadoop example of a polluting test that leads to the failure of other tests

their full names—`testAbsoluteGlob` and `testWithStringAndConfForBuggyPath`; for brevity, we will refer to them as FT and PT, respectively. The bug issue HADOOP-8695 [5] reported that the test FT occasionally fails. Debugging showed the cause was the pollution of the field `testDir`.

The static field `testDir` (line 2) is of type `org.apache.hadoop.fs.Path`. This class represents the name of a file or a directory, and it performs operations on that name, e.g., extracting the components of the path.[2] The field is initialized in the `initialize` method, which is annotated with `@BeforeClass` so that JUnit executes it once before *all* the tests in the test class (and *not* once before *each* test in the test class).

In revision 1099612 in the Hadoop SVN repository [4], the developers added the test PT (while FT did not exist yet). PT sets the field `testDir` (line 22) and leaves it polluted. In that revision, no other test read the value of `testDir`, so no existing technique (e.g., Zhang et al.'s technique [28] based on test reordering or Huo and Clause's technique [7] based on taint analysis) would report PT as a polluting test.

Later on, in revision 1186529, the developers added the test FT. This test reads the value of `testDir` and expects to get its initial value set by the `initialize` method. In Java 6, JUnit indeed ran tests in the order they were listed in the source of the test class; since FT was listed before PT, FT was run first, causing no problems. However, in Java 7, the order in which JUnit runs the tests became non-deterministic, which led to FT failing seemingly non-deterministically. This failure is reported in HADOOP-8695, and debugging showed that FT fails whenever it is run after PT. In this example, it is easy to establish that the cause is the pollution of the field `testDir`.[3]

This example shows how test pollution can create problems, sometimes much later after the polluting test is added,

[2]While the objects in this example represent directory and file names, all objects are still *in memory*, and the example does *not* pollute the file system but only the heap.

[3]The fix in revision 1374447 moves the initialization of `testDir` to a new method annotated with `@Before` such that JUnit sets up the state before each test.

making it potentially hard to debug and fix. In this example, the polluted shared state is directly the static field in the test class. However, in general, the polluted shared state can be an object much deeper in the heap (not directly pointed to by the static field), and the polluted shared state can be reachable starting from a static field in the code under test (not in the test code). Debugging such cases is much harder, especially long after the code is written. Most importantly, the developers may not be aware that their tests pollute the shared state until such pollution results in failures.

PolDet helps developers find polluting tests early. If it were run on the class `TestPathData` when the test PT was added (although FT did not exist yet), PolDet would report that PT polluted the shared state. Moreover, PolDet would also report where the states differ. Given such a report, the developer can then choose to either fix the test right away or to provide a configuration option for PolDet to avoid reporting this pollution in the future. Had the Hadoop developers used a PolDet(-like) tool in revision 1099612, they could have avoided the problems that started after revision 1186529 and lasted until revision 1374447.

3. TECHNIQUE

We next describe our PolDet technique for finding polluting tests. PolDet takes as input a set of tests (and configuration options that specify how to compare states). PolDet produces as output a subset of tests that modify the state, and for each such test produces some difference, identified by an access path through the heap or a file name.

Test executions operate on the system state that consists of parts shared across tests (program heap, local file system, and network-accessible persistent state, e.g., services, databases, etc.) and parts not accessible across tests (e.g., the stack of each test invocation). We are interested in the parts that are shared and can be polluted from one test run to another. We refer to these parts as the *cross-test-shared state*. In general, pollutions could occur via network or databases, but in this paper, we focus on pollutions via heap state and file system; prior studies show these two to be the most prevalent causes of test dependency [13, 28].

We first discuss points at which to compare states. We then formalize the concept of heap-shared state, describe the state abstraction that PolDet uses, define heap-shared state differences, and describe what differences PolDet reports. We finally discuss the comparison of file-system states.

3.1 Program Points

To find state pollutions, PolDet captures the state before the test starts executing and after the test finishes executing. So far we have intuitively referred to the program points before and after the test execution. To precisely define these points, we need to consider how a test framework invokes the tests. Most test frameworks allow the developer to provide some `setUp` and `tearDown` code to execute before each test (to set up the state) and after the test (to clean the state at the end), respectively. Ideally, the states should be captured before the `setUp` code and after the `tearDown` code. We elaborate more on the choice of capture points in Section 4.2. (Interestingly, our experiments show that the `setUp` and `tearDown` code fragments do not themselves pollute the state; if a test pollutes the state, then almost always the test body itself pollutes the state.)

3.2 Heap-State Representation

Formally, we model the heap-shared state of an object-oriented program as a graph with labeled edges. Nodes represent the heap-allocated objects, classes, and primitive values including `null`. Edges represent object fields: if the graph models a concrete heap, then there exists an edge with a label f from node o_1 to node o_2 iff the field f of the object represented by node o_1 points to the object represented by node o_2. Each object has a field representing its class. Arrays are modeled as objects whose outgoing edges are labeled with array indexes and point to array elements. We also allow for abstract heaps whose labels need not be fields, as discussed later in this subsection.

Definition 1. *A* heap-shared state *is a multi-rooted graph* $\mathrm{G} = \langle \mathrm{V}, \mathrm{E}, \mathrm{R} \rangle$ *with* $\mathrm{V} \subseteq \mathcal{O} \cup \mathcal{C} \cup \mathcal{P}$, $\mathrm{E} \in 2^{(\mathcal{O} \cup \mathcal{C}) \times \mathcal{F} \times (\mathcal{O} \cup \mathcal{P})}$, *and* $\mathrm{R} \subseteq \mathrm{V}$, *where* $\mathcal{O}$ *is the set of objects in the heap,* $\mathcal{C}$ *is the set of classes in the program,* $\mathcal{P}$ *is the set of primitive values (including* `null`*), and* $\mathcal{F}$ *is the set of object fields in the program, integers (for array indexes), and additional labels introduced by abstraction. If the graph models a concrete heap, then* $\langle o_1, f, o_2 \rangle \in \mathrm{E}$ *iff* $o_1.f = o_2$ *on the heap.*

The heap-shared state represents the parts of the program state reachable from the roots R. The roots correspond to the variables in the global scope that are accessible across test executions. For example, in the Java language, the roots are the static fields of all classes loaded in the current execution, while in the C language, the roots are the global variables. The general definition of roots needs to be instantiated for each language and even for each test framework for the same language. For example, JUnit and TestNG are the two most popular frameworks for Java, and they share different parts of the heap across tests: JUnit shares only the state reachable from the static fields, while TestNG also shares the state reachable from the test-class instance.

State abstraction can ignore some parts of the state that are overly complex or contain regions irrelevant for the tests. For example, consider a state with an object representing a set. The concrete set implementation uses some data structure, e.g., an array, a tree, or a hashtable. For most tests (unless they focus on testing the set library itself), the particular set implementation is irrelevant, and only the elements that the set contains are relevant. A concrete heap-shared state that captures all objects, including all the data-structure implementation details, is usually not the best choice. To compare the states and present the differences, it is preferable to consider two sets with the same elements to be the same regardless of their implementation details. This is similar to how Java serialization ignores some fields when writing object graphs to the disk. Abstraction can also reduce the size of the captured state and runtime overhead.

Our technique allows for abstraction that omits some concrete edges from the heap-shared state or introduces new edges to it. In a concrete heap-graph, every edge label corresponds to some concrete field or array index in the heap, but an abstract heap-graph can have additional edge labels. More importantly, in an abstract heap-graph, some objects may have multiple outgoing edges with the same label. In general, PolDet users can define abstractions specific to their program; by default, our implementation uses some generic abstractions from the XStream library [26] as described in Section 4.5.

3.3 Finding Heap-Shared State Differences

PolDet compares heap-graphs using graph isomorphism based on node bijection [24]. In other words, the actual identity of the objects in the two states does not matter, but only the shape that connects these objects and the primitive values stored in the objects do matter. The rationale for this is twofold. First, the two captured states come from the same program execution, so two nodes that bijectively correspond in the two heap-graphs most likely represent only one object in the actual program state. Second, most tests do not depend on the object identity, so even if two nodes that bijectively correspond do not represent the same object but represent two different objects that have equivalent field values, the test execution is unlikely to observe the difference. (In Java, code can observe the identity of an object `o` with `System.identityHashCode(o)`.)

We first define isomorphism for two heap-graphs that have exactly the same set of roots.

Definition 2. *Two multi-rooted graphs* $G = \langle \mathrm{V}, \mathrm{E}, \mathrm{R} \rangle$ *and* $G' = \langle \mathrm{V}', \mathrm{E}', \mathrm{R} \rangle$ *are* isomorphic*, in notation* $G \approx G'$*, iff there exists a bijection* $\rho : \mathrm{V} \to \mathrm{V}'$ *that is identity for all classes and primitive values (*$\rho(x) = x$ *for all* $x \in \mathcal{C} \cup \mathcal{P}$*) and* $\mathrm{E}' = \{\langle \rho(o), f, \rho(o') \rangle \mid \langle o, f, o' \rangle \in \mathrm{E}\}$.

Because this definition requires the two graphs to have the same set of roots, it is too strict for comparing heap-graphs in most popular languages, because the set of roots can change during program execution. For example, languages that run on the JVM [12] or CLR [10] have lazy class loading that can add static fields, increasing the number of roots, and programs can also dynamically unload classes, decreasing the number of roots. To accommodate different sets of roots, we define a restriction of a heap-graph with respect to a set of roots, intuitively capturing only the subgraph that is reachable from the given set of roots.

Definition 3. *A* root-restriction *of a graph* $\mathrm{G} = \langle \mathrm{V}, \mathrm{E}, \mathrm{R} \rangle$ *for a set of roots* $\mathrm{R}' \subseteq \mathrm{R}$*, in notation* $\mathrm{G}_{|R'}$*, is the graph* $\mathrm{G}' = \langle \mathrm{V}', \mathrm{E}', \mathrm{R}' \rangle$ *with* $\mathrm{V}' = \{v \in \mathrm{V} \mid \exists r \in \mathrm{R}'. \langle r, v \rangle \in \mathrm{E}^*\}$ *(where* E^* *is the reflexive transitive closure of* E*) and* $\mathrm{E}' = \mathrm{E} \cap (V' \times \mathcal{F} \times V')$.

We next define *common-roots isomorphism* that requires two restrictions to be isomorphic for the common roots.

Definition 4. *Let* $\mathrm{G} = \langle \mathrm{V}, \mathrm{E}, \mathrm{R} \rangle$ *and* $\mathrm{G}' = \langle \mathrm{V}', \mathrm{E}', \mathrm{R}' \rangle$ *be two heap-graphs. We say* G *is* common-roots isomorphic *with* G′*, in notation* $\mathrm{G} \approx_{\cap} \mathrm{G}'$*, iff* $\mathrm{G}_{|R \cap R'} \approx \mathrm{G}'_{|R \cap R'}$.

Finally, we precisely specify that PolDet checks common-roots isomorphism of heap-graphs to find tests that pollute the heap-shared state. If two heap-graphs are not common-roots isomorphic, PolDet reports a difference. More specifically, PolDet finds the difference by traversing the two graphs simultaneously from each root and then reports some path, called *access path*, that leads to two nodes that cannot bijectively correspond. For abstract heap-graphs, where some nodes may have multiple outgoing edges with the same label, there could be many differences even for the same node; we require the tool to report any one difference, rather than all differences.

Definition 5. *Two graph nodes* $v \in \mathrm{G}$ *and* $v' \in \mathrm{G}'$ *are not bijective if the subgraphs rooted in* v *and* v' *are not isomorphic, i.e.,* $\mathrm{G}_{|\{v\}} \not\approx_{\cap} \mathrm{G}'_{|\{v'\}}$ *when* $\rho(v) = v'$.

3.4 Class Loading

The use of common-roots isomorphism can lead to false negatives, i.e., not finding a difference between the graphs of two states even when a test does pollute the state. Common-roots isomorphism would not detect a test that polluted a part of the state only reachable from the roots (static fields) of classes that were lazily loaded after the test has begun. For example, consider a test whose execution loads a new class and initializes its static fields with default values, but the test modifies those values (or the state reachable from the static fields of the newly loaded class) before PolDet captures the state. If another test relies on the state reachable from this newly loaded class, this subsequent test could fail when the values are not the default from the class initialization. Because common-roots isomorphism ignores the roots of the new class, it misses this state pollution.

One solution we propose for lazy class loading is to eagerly load all classes needed by a test before starting the test. Such eager loading keeps the roots of the graphs the same at all capture points, reducing common-roots isomorphism (Def. 4) to simple isomorphism (Def. 2). Determining what classes a test needs can be done by running the test twice: first run just to collect the set of loaded classes, and second run, after eagerly loading all the classes, to actually compare the states. The granularity of the collection offers a trade-off between the performance of collection and comparison: collection at the test-suite level may load classes that are not needed for some tests (resulting in bigger states being collected for each test, incurring a high comparison overhead), while collection at the test-class or test-method level incurs a higher overhead for the collection itself. Moreover, eager class loading is challenging, e.g., when code dynamically generates and loads/unloads classes, uses specialized class loaders, or otherwise may change the behavior based on the order in which classes are loaded. Another solution to handle lazy class loading would be to capture and compare states also right after the static class initializer finishes; however, that requires more instrumentation and runtime overhead.

3.5 Finding File-System State Differences

A test can pollute not only heap-shared state but also file-system state. For example, a test can create a new file or modify an existing file, without deleting the new file or resetting the content of the existing file after it finishes, resulting in a polluted file system that could affect the behavior of some subsequent test. PolDet tracks file-system state by tracking which files are present, hashing their contents, and checking the file/directory last-modified timestamps provided by the operating system. Before a test starts, PolDet iterates through each file in a given portion of the file system, computes a hash of the content for each file, and stores a map from the file name to the file hash. PolDet also saves the time before the test starts. After the test finishes, PolDet uses the last-modified timestamp of the files in the portion of the file system to check if any file or directory was modified. If an existing file was written to, PolDet hashes the new content of the file to compare with the saved hash in order to check if the content indeed changed (or if the write just rewrote the old value). If a file PolDet hashed before no longer exists, then the file was deleted. If any existing file is changed or deleted, or if some new file is created, PolDet reports that the test polluted the file-system state.

```
1 class T {
2     @Before void setUp() {
3         ...
4     }
5     @Test void t1() {
6         ...
7     }
8     @Test void t2() {
9         ...
10    }
11    @After void tearDown() {
12        ...
13    }
14 }
```

```
15 // before constructor
16 T t = new T();
17 // before setup
18 t.setUp();
19 // after setup
20 t.t1();
21 // before teardown
22 t.tearDown();
23 // after teardown
24
25 t = new T();
26 t.setUp();
27 t.t2();
28 t.tearDown();
```

Figure 2: JUnit workflow for running tests and illustration of capture points

4. IMPLEMENTATION

We have implemented a prototype of our PolDet technique in a tool, also called PolDet, that finds polluting tests written in the JUnit testing framework. We built PolDet on top of JUnit, so it can be run on any project that uses JUnit. We first introduce the relevant background about JUnit, then describe where and how PolDet captures and compares heap-shared states, and finally describe how PolDet compares file-system states.

4.1 JUnit Background

We briefly summarize some details of JUnit 4. JUnit is the most popular unit testing framework for Java, e.g., out of 666 most active Maven-based Java projects from GitHub, 520 use JUnit. JUnit test suites are organized in test classes, with each test being an instance method annotated with `@Test`. Test classes can also have methods that set up the state before the test and clean it after the test; these methods are annotated with `@Before` and `@After`, respectively. Figure 2 shows an example test class with two tests and illustrates how JUnit invokes the constructor and methods of this class.

First, JUnit creates a new instance of the test class. Next, it invokes on the instance the setup methods annotated with `@Before`. Then, it invokes the test method itself on the instance, running the test. Finally, it invokes the cleanup methods annotated with `@After`. JUnit uses each test-class instance to run only one test; hence, it creates a new instance and repeats the same process for each test method defined in the test class. Any instance fields defined in the test class cannot be accessed across test-method runs because they belong to their own separate instances. Therefore, the heap-shared state consists of all objects reachable from static fields.

4.2 Capture Points

PolDet extends the JUnit's test running mechanism to capture the state before and after the test executes. Figure 2 shows various execution points in the JUnit's workflow where the state could be captured. For example, the state *before* the test is run can be captured at the point before or after the `setUp` is run, and the state *after* the test is run can be captured at the point before or after the `tearDown` method. In general, all these points could have different states because `setUp` and `tearDown` methods can mutate the state either to set it up or clean it for the test execution. Moreover, some software projects may enforce a discipline where tests only use `@Before` methods to set up the entire state the test depends on, so one could compare the states right after `@Before` methods across *consecutive* tests rather than at various points for the same test. Our tool can be configured to these various scenarios.

4.3 Capturing Heap-Shared State

To capture states, we (1) modified the JUnit runner to call our state-capturing logic whenever a test execution reaches one of the capture points and (2) wrote a Java agent that keeps track of all classes loaded (and unloaded) by the JVM. Running our PolDet tool requires providing the agent to the JVM and using our modified JUnit. The modified JUnit runner invokes our state-capturing logic that first queries the agent to obtain all the classes loaded at the point of capture. For each loaded class, PolDet uses reflection to obtain all the static fields for that class. PolDet ignores final static fields that point to immutable objects because the heap values reachable from these fields cannot be changed. All other static fields that are not final or point to mutable objects become the roots of the heap-graph. The state reachable from these roots can change, so PolDet needs to capture the objects reachable from these fields. Note that PolDet *does* consider static fields that are *not* `public` because the values referred to by these fields can still be observed and modified through various getter or setter methods.

More specifically, PolDet first creates a map whose keys are fully qualified names of static fields and values are the pointers to the actual heap objects pointed by these fields. PolDet then invokes XStream [26], a Java library for XML serialization, to traverse the entire heap reachable from this map and to serialize it into an XML format. The produced XML string encodes the captured state of the program.

4.4 Comparing Heap-Shared States

After obtaining the serialized XML strings of the captured states, PolDet diffs them using XMLUnit [25], an XML diffing library. XMLUnit compares (XML) parse trees rather than graphs. However, if XMLUnit reports no differences, the two heap-graphs encoded in XML are common-roots isomorphic (Def. 4). If XMLUnit does report some difference, it also provides a path to some differing entry in the trees; in other words, it provides an access path that leads to the difference (Def. 5). Each access path starts from one of the roots (static fields), traverses fields through the heap, and ends up with a differing value pointed to by the last field on the path. Such access paths can aid the developer in debugging the state modification.

4.5 Abstracting Heap State for Java

As discussed in Section 3.2, not all heap objects are relevant for state pollution. Some regions of the state are expected to change between test runs and are not observable by any *natural* code that developers would likely write in a test. While one could always observe all the state changes via reflection—indeed, that is how XStream traverses the state to produce XML—most natural code does not do that.

For example, common data structures found in the standard `java.util` package, such as `ArrayList` or `HashMap`, have a field `modCount`, which is an integer that counts how many times a data structure is modified in order to detect concurrent iteration and modification of collections. As this counter is private, the test code cannot easily access this

field, and the developer is unlikely to desire to observe this state. XStream abstracts away many such implementation details when performing serialization. For example, by default it serializes data structures from the `java.util` package at an abstract level, e.g., serializes sets as unordered collections without storing the concrete implementation details.

While some fields should be ignored when considering state pollution for all projects, other fields may be ignored only for some projects. The developer can decide whether or not some modified field could affect other tests, and PolDet provides three options for the user to specify what fields to ignore when comparing states.

First, PolDet has an `include_roots` option. Typically, the developer is only concerned with problems in her *own* code. Any pollution accessible only from some third-party library static field is less likely to be something the developer can easily fix or even reason about. The `include_roots` option allows the developer to define a set of packages in which PolDet should search for roots. For example, PolDet can include the static fields only from classes that belong to the packages in the current code under test.

Second, PolDet allows the user to ignore certain roots by specifying regular expressions for names of *static* fields. For example, many tests use mocking frameworks, such as Mockito, that keep internal counters or other static variables that do not affect the execution of the test. (Many static fields that originate from Mockito are not filtered out by the `include_roots` option as the generated mocks are in some package from the code under test.) The developer can opt to ignore such static fields with the `exclude_roots` option.

Third, PolDet allows the user to apply a finer-grain control and ignore certain *instance* fields of classes with the `exclude_fields` option. Our inspection found fields that may refer to values such as caches, which are easily affected by the execution of tests, yet will not affect their execution. As PolDet uses XStream for state traversal, it can easily specify fields to ignore by passing the class and field names to XStream, so it does not serialize the field.

4.6 Eager Class Loading

We implement eager class loading by (1) reusing the agent from PolDet to keep track of all loaded classes, (2) adding a shutdown hook, which is a thread that JVM runs right before it exits, and (3) adding code that uses reflection to load a set of classes whose names are in a given file. We run PolDet twice on all tests. The first run is with the agent but without state capturing, and the hook queries the agent to obtain all classes loaded by the tests and saves the class names to a file. The second run is with state capturing, but before capturing any state, our code loads all the classes from the first run.

4.7 Comparing File-System State

To detect file-system state pollutions, PolDet hashes file contents and uses the file last-modified timestamps provided by the operating system. To avoid the high overhead of exploring the entire file system, PolDet allows the user to specify the portions of the file system to consider. By default, we consider the current directory where the tests are run and the temporary directory (`/tmp` on Linux systems), because these are most likely places where tests would modify files. Before the test suite starts running, PolDet finds all files recursively reachable from these starting directories, hashes each file's content, and maps the file name to this hash. Before each individual test run, PolDet creates a new file marked with the current timestamp, executing `touch f` to create a fresh file `f`. When the test finishes, PolDet runs `find d -newer f`, where `f` is the file created before the test started, and `d` is either the current directory or `/tmp`. This command finds all files (and directories) reachable from `d` whose last-modified timestamp is newer than `f`. For each such file, if it was mapped to some hash (i.e., it existed before the test), PolDet hashes the file content again and compares it with the hash from the map. If a file that was hashed before no longer exists, then the test deleted the file. If any file is new, the hash of some old file differs, or a file is deleted, the test polluted the file system, and PolDet reports the polluting test and the file name. The map of file name to hash is updated with any changed hash and any deleted files are removed from the mapping in preparation for the next test run.

Project	LOC	Classes w/ Static Fields All	Classes w/ Static Fields CUT	Number of Static Fields All	Number of Static Fields CUT
android-rss	1733	80	5	244	9
Athou Commafeed	11095	62	1	220	5
FizzBuzzEE	1353	29	0	72	0
Maven-Plugins	2061	216	0	1093	0
JSoup	14925	52	23	242	170
Mozilla Metrics	4180	255	10	981	19
Spring JDBC	3170	47	1	106	8
Jopt Simple	9655	88	5	241	13
slf4j	14085	42	13	129	57
Spring MVC	3675	364	1	1397	4
Spring Petclinic	2970	219	0	1161	0
Spring Test MVC	8240	446	17	1575	22
Apache Httpclient	78497	437	106	4593	355
Bukkit	32984	166	90	1393	1108
Caelum Vraptor	33898	449	62	5837	94
cuke4duke	8104	429	5	2230	5
Dropwizard	25838	1910	44	15886	105
Fakemongo Fongo	13755	458	76	2904	1616
Scribe Java	6049	60	21	151	46
Kuujo Vertigo	27708	165	12	484	43
Java APNS	5462	264	17	1006	62
Spark	6075	277	23	1096	58
Square Retrofit	9729	388	40	1482	104
Square Wire	13998	109	51	499	299
twitter Ambrose	5927	248	10	866	37
twitter hbc	6025	215	13	1595	54
Total	351191	7475	646	47483	4293

Figure 3: Project statistics

5. EVALUATION

To evaluate PolDet, we ask the following questions:

RQ1. What percentage of tests pollute heap-shared state?
RQ2. How accurate is PolDet (true vs. false positives)?
RQ3. What is the time overhead of running PolDet?
RQ4. How does eager loading compare with lazy loading?
RQ5. What percentage of tests pollute file-system state?

5.1 Experimental Setup

To scale our experiments to a wide variety of projects, we automated the integration of PolDet into Maven. Maven is a popular build system for Java projects, widely used on the GitHub repository for open-source projects. Because PolDet builds on top of JUnit, we integrated PolDet

Project	Test Methods					Test Classes							
	#Tot	AR #Pol	ER #Pol	ER #TP	FS #Pol	#Tot	AR #Pol	ER #Pol	ER #TP	FS #Pol	Roots	Fields	Overhead
android-rss	24	0	0	n/a	0	4	0	0	n/a	0	0	0	2.37
Athou Commafeed	8	0	0	n/a	0	2	0	0	n/a	0	0	0	1.13
FizzBuzzEE	1	0	0	n/a	0	1	0	0	n/a	0	0	0	1.07
Maven-Plugins	28	0	0	n/a	1	5	0	0	n/a	1	0	0	1.18
JSoup	410	0	0	n/a	0	24	0	0	n/a	0	0	0	23.56
Mozilla Metrics	33	0	0	n/a	0	14	0	0	n/a	0	0	0	1.95
Spring JDBC	12	0	0	n/a	0	1	0	0	n/a	0	0	0	1.34
Jopt Simple	701	0	0	n/a	1	115	0	0	n/a	1	0	0	1.76
slf4j	13	0	0	n/a	0	2	0	0	n/a	0	0	0	1.21
Spring MVC	36	0	0	n/a	0	9	0	0	n/a	0	0	0	1.22
Spring Petclinic	2	0	0	n/a	0	2	0	0	n/a	0	0	0	1.17
Spring Test MVC	288	3	3	0	0	44	1	1	0	0	1	14	4.15
Apache Httpclient	1634	129	94	78	0	138	35	20	14	0	6	7	1.72
Bukkit	285	11	9	1	0	38	2	2	1	0	3	4	24.07
Caelum Vraptor	1132	172	36	1	5	165	66	4	1	4	8	5	56.01
cuke4duke	51	25	25	0	0	10	3	3	0	0	1	4	1029.57
Dropwizard	419	37	3	1	1	108	22	3	1	1	3	5	27.54
Fakemongo Fongo	359	68	64	50	0	15	14	13	2	0	2	4	4.17
Scribe Java	99	3	3	0	0	18	1	1	0	0	1	3	2.14
Kuujo Vertigo	63	13	13	13	0	4	2	2	2	0	1	5	1.55
Java APNS	89	18	15	0	0	15	10	9	0	0	10	6	1.82
Spark	54	42	42	39	0	6	4	4	3	0	5	18	622.74
Square Retrofit	197	9	1	0	0	17	4	1	0	0	1	3	2.14
Square Wire	61	5	5	0	0	8	3	3	0	0	1	3	2.70
twitter Ambrose	13	8	8	8	0	7	3	3	3	0	2	2	3.09
twitter hbc	93	32	3	3	0	14	4	1	1	0	1	1	2.00
Total	6105	575	324	194	8	786	174	70	28	7	46	84	4.50

Figure 4: State pollution results; the columns are described in Section 5.2

into Maven by replacing the `junit.jar` file in the Maven dependency repository with our version that invokes PolDet instead of the original JUnit. Moreover, we automatically modify the Maven `pom.xml` configuration file for each project to add our Java agent to run alongside our modified JUnit. With this setup, any Maven project using JUnit 4 can be run with PolDet to find polluting tests.

For our evaluation, we randomly chose 26 diverse Maven-based Java projects from GitHub, varying in size (from 1,353 to 78,497 LOC), number of tests, number of static fields, and application domains (including web frameworks, gaming servers, or networking libraries). Figure 3 shows some statistics about these projects.

PolDet has four main configuration options:

`-capture_points` determines where to capture the states to be compared. Figure 2 illustrates several points at which PolDet can capture the state, and the user can configure PolDet to use any pair of capture points. Our default uses the point before `setUp` paired with the point after `tearDown`. We have also evaluated several other pairs and obtained almost identical results.

`-include_roots` determines whether the graph roots should include static fields from all loaded classes or only from the classes whose name matches given regular expressions. In our experiments, we set the expressions to match the packages from the project under test such that PolDet ignores fields from library classes. Figure 3 shows some statistics about classes and static fields. It shows the number of classes that are loaded during the execution of the project's test suite and have at least one static field; it shows this number both "All" from all packages (i.e., as if running PolDet with no specified `include_roots`, considered *disabled*) and "CUT" only from the packages whose source belongs to the project under test (i.e., as if running PolDet with the `include_roots` option matching package names, considered *enabled*). Likewise, it shows the number of static fields as if `include_roots` was both disabled and enabled. However, disabling `include_roots` results in many more roots and much larger heap-graphs. (In fact, our PolDet prototype would often run out of memory if comparing states for `include_roots` disabled.) All our subsequent experiments run with `include_roots` enabled. We automatically find the packages in the project under test by exploring the project's source code.

`-exclude_roots` specifies the set of roots to ignore when serializing the states; while this set can be arbitrary, our experiments evaluate two settings: (1) not ignoring any roots and (2) ignoring roots from classes that are known to lead to irrelevant state, in particular `mock` classes, certain fields of standard libraries, and automatically generated classes that have `$$` in their name (but not the inner classes that have only `$` in their name).

`-exclude_fields` specifies the set of instance fields to ignore when serializing the states; while this set can be arbitrary, our experiments evaluate two settings: (1) not ignoring any fields and (2) ignoring the minimum number of fields that makes PolDet report no pollution (which is used just in the experiments to measure the size of pollutions and is not a recommended option as it makes PolDet miss both all true positives and all false positives).

5.2 Results

Figure 4 shows the results of running PolDet. For both test methods and test classes, it tabulates the total number, the number that PolDet reports as polluters when run without `exclude_roots` (AR #Pol), the number that

PolDet reports as polluters when run with `exclude_roots` (ER #Pol), the number of true positives among the latter reports (ER #TP), and the number that PolDet reports as polluting the file-system state (FS #Pol).

5.3 Heap-State Pollution

Inspection Procedure: We manually inspect each report to determine if it is a true positive or a false positive. We label a report as a true positive if one can write a reasonable test that would pass or fail depending on whether it was run before or after the reported polluting test. Otherwise, if one cannot write a test that would observe the state difference using the available API but would need to resort to reflection, we label the report as a false positive. We inspect the access path from a static root to the polluted field reported by PolDet to find how to access the polluted state. For each field on the path, we check how it can be accessed starting from the static root. If we find a reasonable way to read the polluted field, we consider the case a true positive. When the path is short, it is relatively easy to determine whether a report is a true positive or a false positive. In contrast, if the polluted field is in some third-party library code or the access path to it is long, then the code under test very likely cannot directly observe the value of the field, suggesting it to be a false positive. Indeed, we used the location of the field and the length of the path to prioritize our inspection of the reported polluting tests; we examine first the reports where the polluted field is in the code under test and has a relatively short access path. We recommend such simple prioritization for developers to inspect the reports. We discuss one example of each true positive and false positive later in this section. When PolDet reports no pollution, the true positive count does not apply, so we mark the cell `n/a`; we still show the other statistics about PolDet, e.g., runtime overhead.

Inspection Results: Our brief, initial inspection of the reports without `exclude_roots` (i.e., with all roots – *AR*) found many cases of false positives due to a small number of common issues across projects. As one example, several projects use the Mockito library that internally keeps various counters, e.g., `SequenceNumber.sequenceNumber` that tracks the number of times a mock instance is created. A developer using Mockito would not care that such an internal counter changed as it is effectively inaccessible. As another example, several states include `java.lang.ref.SoftReference` objects that have a field updated by the JVM to track the timestamp of when an object was garbage collected. We want to avoid such fields. Finally, we found several projects with automatically generated classes whose name includes double `$$`.

Our default configuration for the PolDet tool is thus to run with `exclude_roots` (*ER*) to exclude `mockito`, standard library fields for timestamps, and classes with `$$`. In this configuration, we provide the answers to our first two questions. **RQ1:** PolDet reported 5.30% (324 out of 6105) tests as polluting tests. **RQ2:** Of those, 59.87% (194 out of 324) tests are true positive polluters.

While PolDet reports test methods, we also present the results for test classes: a test class is considered a polluter if it has at least one method that is a polluter, and a test class is considered a true positive if it has at least one method that is a true positive. The ratios for classes are similar as for methods: PolDet reported 8.90% (70 out of 786) classes, and of those, 40.00% (28 out of 70) are true positive polluters. An interesting finding is that a class often has both true positive and false positive test methods.

We have even more interesting findings for roots that lead to the heap-shared state differences for the tests in our projects. Given the overall small number of such roots (46), we wonder if we can classify the reported polluting tests based on these roots. Intuitively, a developer determines if a report is a true positive by examining some portion of the polluted state, and the developer can begin examining the state from the static root. We clustered all the reports by the 46 static roots that lead to the pollution. We found that the number of polluting tests associated with a reported static root ranges from 1 to 76, with an average of 10.02 tests per root. We also found that for almost all of the roots (43 out of 46), the tests associated with the root are either all true positives or all false positives. Only three of the roots are associated with tests that are a mix of both. Two roots are in Apache Httpclient: `NTLMEngineImpl.RND_GEN` has 3 associated tests, and `LocalTestServer.TEST_SERVER_ADDR` has 15 associated tests. One root is in Spark: `Spark.server` has 33 associated tests. In all these cases, tests associated only with this static root are false positives, while the other tests associated also with another, different static root are true positives. Moreover, all the tests associated with that other static root are true positives. Overall, for all tests reported by PolDet, a developer could just examine the static root(s) that lead(s) to the state difference and with high confidence determine if the report is a true positive or a false positive.

While we expect a developer would inspect the PolDet reports starting from the roots of the access paths that lead to differences, the developer could also inspect starting from the differences themselves. The column "Fields" in Figure 4 shows the minimum number of fields that should be set in `exclude_fields` to obtain zero reports from PolDet, and it is a measure of how much the states differ. Note that these fields are *instance fields*, close to the difference, rather than *static fields* that are roots from which the differences are reachable. Overall we find that the user would need to ignore a larger number of fields than roots to cover all the differences. As a result, we recommend the users to inspect PolDet reports starting from the roots.

Example True Positive: One example true positive found by PolDet is the `PotionTest.setExtended` test from the Bukkit project [2]. Bukkit implements a server for the popular Minecraft game. The root `PotionEffectType.byName` (declared in the code under test) has type `java.util.Map` and tracks the added potion effects (which are one of the game features to modify game entities).

Figure 5 shows the relevant code snippet. The body of the polluting test `setExtended` calls the method `registerPotionEffectType`, which leads to adding the `PotionEffectType` passed as the argument. In this case, the argument passed is 19, representing the type of potion effect to be created and registered. The problem is that the potion type still resides inside the static map `byName` even after the test finishes execution, and other tests could depend on that map. To confirm this is a true positive, we generate the test `flakyTest`, which adds the `PotionEffectType` 18 (which increases damage to an entity over time), and assert that the `PotionEffectType` 19 (which decreases damage to an entity over time) does not exist. This added test passes if run before `setExtended` and fails if run after `setExtended`.

```
1 public class PotionTest {
2     ...
3     @Test
4     public void setExtended() {
5         PotionEffectType.registerPotionEffectType(new
6             PotionEffectType(19) { ... }
7         });
8     }
9     ...
10    @Test
11    public void flakyTest() { // we added this test
12        PotionEffectType.registerPotionEffectType(new
13            PotionEffectType(18) { ... }
14        });
15        assertNull(PotionEffectType.getByName(new
16            PotionEffectType(19) { ... }
17        });
18    }
19 }
```

Figure 5: The Bukkit true positive example with a test written to confirm the pollution

We chose this, relatively simple example for the ease of presentation. In many other cases, the difference would be hard to understand without the access paths from PolDet.

Example False Positive: Some of the pollutions reported by PolDet are false positives, i.e., no reasonable test may fail because of the polluted state. Figure 6 shows an example from the Java APNS project [17]. PolDet reports that the test `ApnsConnectionTest.sendOneQueued` pollutes the field `marshall` reachable from the static root `msg1` declared in the test class. We omit details of the test body not relevant to the pollution; the key is the assert statement that calls the method `marshall()` on `msg1`. The code of `marshall()` inside the class `SimpleApnsNotification` lazily initializes the field `marshall`, so the state modification is a false positive. In fact, we find lazy initialization to be a common cause of false positives in PolDet, and we plan in the future to devise a heuristic to automatically remove such reports.

5.4 Efficiency

We evaluate the overhead of PolDet by measuring the ratio of the runtimes of executing the test suites with and without PolDet. We ran our timing experiments on a 64-core Scientific Linux machine with 64 GB of RAM. While such a machine is not a common developer's desktop/laptop, it is representative of a build-farm server on which many projects run their continuous integration systems. All time measurements are wall-clock time. Note that our PolDet prototype is not optimized for real deployment but aimed for experimental purposes (e.g., it collects states at several points in test execution).

The last column of Figure 4 shows the PolDet overhead. It ranges from 1.07x to 1029.57x. The two outliers, Spark and cuke4duke, have large overhead due to heavy use of highly complex objects. For example, cuke4duke is a specification framework that embeds JRuby, a Ruby JVM interpreter, and hence the state that PolDet traverses is highly complex, including all JRuby data structures. In such cases, using good state abstractions or filtering out static roots and fields in PolDet can be useful not only to stop the traversal of irrelevant state and reduce the high overhead but also to control false positives. The last row ("Total") reports the *geometric mean* of overheads: 4.50x. **RQ3:** PolDet has a reasonable overhead on a build-farm server even when run on the entire test suite, but we expect that most developers would run PolDet only on their newly added tests.

```
1 public class ApnsConnectionTest {
2     ...
3     static SimpleApnsNotification msg1 =
4         new SimpleApnsNotification(...);
5     @Test(timeout = 2000)
6     public void sendOneQueued() {
7         ...
8         assertArrayEquals(msg1.marshall(), ...);
9     }
10 }
11
12 public class SimpleApnsNotification {
13     ...
14     private byte[] marshall = null;
15     public byte[] marshall() {
16         if (marshall == null)
17             marshall = Utilities.marshall(COMMAND,
18                 deviceToken, payload);
19         return marshall;
20     }
21 }
```

Figure 6: The Java APNS false positive example

5.5 Eager Class Loading

We also apply the eager loading of PolDet on all 26 projects except Jopt Simple, where eager loading causes the tests to deadlock. PolDet reports 468 polluting tests (144 more than with the default lazy loading) and has a geometric mean overhead of 12.29x (as test suites are run twice, and bigger heap-graphs are created and compared). The new reports stem from common-roots isomorphism ignoring static field roots of classes not loaded before the test. Many new reports are tests that are the first to run in their test class, often with some other tests from the same test class previously reported by the default lazy loading, and the true or false positive status of the new reports being the same as the other reports in the test class. However, with eager loading, PolDet reports more false positives than with lazy loading. The majority of the new false positives (120 out of 144) are from Bukkit and largely due to eager loading including a static field to an instance of a server whose fields indirectly point to thread-related services from the JVM; the only heap-shared state modifications are to these thread services, which are rather non-deterministic and not controlled by the code under test. In total, of 468 reports, 203 are true positives, i.e., eager loading detected 9 true positives not detected by lazy loading. **RQ4:** With eager loading, PolDet can detect more true positives, but at the cost of many more false positives and higher overhead.

5.6 File-System Pollution

Figure 4 also shows the results for file-system state pollutions (FS #Pol). PolDet found only eight file-system state polluting tests, much fewer than heap-shared state polluting tests, with a geometric mean overhead of 2.73x. Interestingly, two projects that had no heap-shared state polluting tests had file-system state polluting tests.

We examined all eight reports and found that each pollutes the `/tmp` directory. More precisely, each test adds some new file, using Java's `File.createTempFile` method, which creates a temporary file guaranteed to have a fresh name. PolDet reports these tests because they do not delete the new files. Although the pollution is mostly benign as the name is guaranteed to be fresh every time the test is run, one can still consider this pollution unnecessary as the file system has extra files added, potentially resulting in filling up the disk space or reaching the limit on inodes. Hashing files removes some false positives, e.g., a Caelum Vraptor

test writes to an existing file in `/tmp`, but writes the same content. **RQ5:** PolDet reports that very few tests pollute the file system and just create fresh temporary files in `/tmp`.

6. THREATS TO VALIDITY

There are several threats to the validity of our evaluation. First, our results may not generalize beyond the projects used in our evaluation. To mitigate this threat, we randomly selected a diverse set of *actively developed and popular* open-source projects that vary in size, number of developers, and number of tests, and that span domains such as web frameworks, gaming servers, or networking libraries.

Second, we implemented our PolDet tool only for JUnit 4 and for heap-shared state and file-system state pollutions. Our results may be affected by the way JUnit runs tests, but JUnit is the most popular testing framework for Java. PolDet does not report pollutions in the database state or network-connected storage systems. While those were not found as widespread in the past [13, 28], they are becoming more important, and future work is needed to address the other persistent cross-test-shared state.

Third but most important, we manually examined the polluting tests reported by PolDet to label false positives and true positives. Because we are not developers on the projects and lack domain knowledge, our labeling can be wrong. We discussed the inspection results with one another to minimize the risk of mislabeling. However, a further study with real developers is required to establish that PolDet reports are useful and prompt changes of polluting tests. Note that reodering the existing test suite [28] to find a failure due to test-order dependency may not work in many cases as the test suite may have no test that can fail due to the pollution. Indeed, the goal of PolDet is to help proactively find pollutions even before they can manifest in test failures.

7. RELATED WORK

Finding problems in test suites is an active area of research. PolDet focuses on a particular type of problems, namely test pollution. We discuss prior research on test dependence, proactive detection of potential software problems, and handling shared state or multiple states.

Test Dependence: Zhang et al. [28] empirically studied test-order dependence and proposed a technique to find dependent tests in the existing test suites. Their study of issue-tracking systems for five projects found 96 dependent tests, of which 61% are due to heap-shared state. Their technique explores selected permutations of test suites to manifest dependent tests. While their technique can *actively* detect dependent tests among the tests in the existing test suite, PolDet can *proactively* detect polluting tests even before a dependency can manifest.

Huo and Clause [7] use taint analysis to find brittle assertions, i.e., cases when a test reads from state regions not explicitly written by the test. These reads can find *potential* test dependencies on heap-shared state. Our common goal is to find potential dependencies, but PolDet finds *writes* to the shared state rather than *reads* from the shared state. Combining the two techniques could give more accurate reports by pairing the tests that pollute certain state regions with the tests that read from those state regions.

Bell and Kaiser [1] present VMVM, a tool that runs multiple tests in the same JVM but selectively resets state regions that may have been written by tests such that each test runs from the initial state as if run in a separate JVM. VMVM instruments all classes and re-initializes the static fields that can be shared across tests. The goal is to speed up testing compared to running each test in a separate JVM. VMVM can tolerate test pollution by providing support for automatically resetting state, but it does not determine if a pollution occurred or not. Muslu et al. [16] also proposed to handle test dependence by running each test in an isolated environment. In contrast, PolDet uses a less intrusive instrumentation than VMVM, can also detect not only avoid/tolerate test dependence, and proactively encourages developers to fix polluting tests.

Proactive Tools: Various research projects proactively detect software problems. For example, Shacham et al. [19] propose a technique that finds atomicity violations that can lead to potential bugs after software changes; Lin et al. [11] propose a technique for retrofitting parallelism into existing applications to prevent performance problems; and Yabandeh et al. [27] propose a technique for distributed systems where nodes predict distributed consequences of their actions and can avoid errors. We share the common philosophy of proactively detecting problems but focus on test suites.

Handling State: Researchers have developed techniques that compare states. For example, Cleve and Zeller [3] and Sumner and Zhang [20] use the state differences between a passing run and a failing run to isolate the cause of a failure. In contrast, PolDet uses state comparison to determine whether or not a test pollutes state.

Researchers have also proposed techniques to refactor shared state into private state. For example, Wloka et al. [21] propose a program transformation for re-entrant programs to refactor shared state to thread-local state, and Wrigstad et al. [22] propose a simple type system to annotate thread-local data for Java. Similar research could be applied to refactor data to be test local to remove pollution. We plan in the future to consider automatic fixing of polluting tests.

8. CONCLUSIONS

When a test fails without exposing a bug in the code under test, the testing process becomes less reliable. Polluting tests introduce dependencies, leading developers to waste time and resources. We formalize the test pollution problem and present PolDet, a technique to find polluting tests by capturing and comparing heap-graphs and file-system states during test execution. Our PolDet prototype runs relatively fast on build machines, incurring on average 4.50x overhead. Our manual inspection of PolDet reports found 194 polluting tests that could easily cause other tests to fail. We envision PolDet to be used during testing to *prevent the introduction of polluting tests* in the test suite. We believe the philosophy of proactively maintaining a reliable test suite can help software teams to develop and test software faster and better. We foresee future research in automatically finding and fixing more causes of flaky tests [13].

Acknowledgments: We thank Xinyue Xu for help with PolDet code; Milos Gligoric for feedback on our ideas; and Owolabi Legunsen, Tifany Yung, and the anonymous reviewers for feedback on a paper draft. This research was partially supported by the NSF Grant Nos. CCF-1012759, CCF-1421503, and CCF-1434590. Alex Gyori was partially supported by the Saburo Muroga Endowed Fellowship.

9. REFERENCES

[1] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *ICSE*, pages 550–561, 2014.

[2] Bukkit. `http://bukkit.org/`.

[3] H. Cleve and A. Zeller. Locating causes of program failures. ICSE, pages 342–351, 2005.

[4] Hadoop Trunk. `http://svn.apache.org/repos/asf/hadoop/common/trunk`.

[5] TestPathData fails intermittently with JDK7. `https://issues.apache.org/jira/browse/HADOOP-8695`.

[6] Welcome to Apache Hadoop. `http://hadoop.apache.org/`.

[7] C. Huo and J. Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, pages 621–631, 2014.

[8] JUnit 4.11 - What's new? Test execution order. `http://randomallsorts.blogspot.com/2012/12/junit-411-whats-new-test-execution-order.html`.

[9] JUnit and Java 7. `http://intellijava.blogspot.com/2012/05/junit-and-java-7.html`.

[10] A. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *PLDI*, pages 1–12, 2001.

[11] Y. Lin, C. Radoi, and D. Dig. Retrofitting concurrency for Android applications through refactoring. In *ESEC/FSE*, pages 341–352, 2014.

[12] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[13] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *FSE*, pages 643–653, 2014.

[14] Maintaining the order of JUnit3 tests with JDK 1.7. `http://www.coderanch.com/t/600985/Testing/Maintaining-order-JUnit-tests-JDK`.

[15] all and sundry: JUnit test method ordering. `http://www.java-allandsundry.com/2013/01/junit-test-method-ordering.html`.

[16] K. Muşlu, B. Soran, and J. Wuttke. Finding bugs by isolating unit tests. ESEC/FSE, pages 496–499, 2011.

[17] Java APNS - Version 1.0.0. `https://github.com/notnoop/java-apns`.

[18] Pivotal Labs: Fighting test pollution. `http://pivotallabs.com/find-test-pollution-rspec/`.

[19] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, pages 51–64, 2011.

[20] W. N. Sumner and X. Zhang. Comparative causality: Explaining the differences between executions. In *ICSE*, pages 272–281, 2013.

[21] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *ESEC/FSE*, pages 173–182, 2009.

[22] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, and J. Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, pages 445–469, 2009.

[23] J. Wuttke, K. Muşlu, S. Zhang, and D. Notkin. Test dependence: Theory and manifestation. Technical report, University of Washington, 2013.

[24] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.

[25] XMLUnit. `http://www.xmlunit.org/`.

[26] XStream. `http://xstream.codehaus.org/`.

[27] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, pages 229–244, 2009.

[28] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *ISSTA*, pages 385–396, 2014.