PROACTIVELY DETECTING UNRELIABLE TESTS

BY

ALEX GYORI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Associate Professor Darko Marinov, Chair
Associate Professor Madhusudan Parthasarathy
Associate Professor Tao Xie
Dr. Shuvendu K. Lahiri, Microsoft Research

Abstract

Regression testing is the most wide-spread method to ensure the quality of software systems. Whenever a change is made to the software, tests are run to ensure bugs are not introduced: if all tests pass, the change is merged into the codebase; otherwise, the developer needs to identify the bug that was introduced by the change. Developers assume that the outcomes of the tests in the regression testing process are reliable, i.e., that the failure indicates a bug introduced by the change. Unfortunately, unreliable tests manage to get into the test suite, slowing down the developers' workflow and having developers debug not their software but rather the test code or infrastructure. This dissertation presents two techniques to enable developers to more easily and proactively detect and debug unreliable tests early, before they become a problem and slow down the development process.

Developers write unreliable tests, which may pass on their machine but may at a later point fail because the environment changes. This dissertation presents a technique to detect when code makes wrong assumptions on underdetermined APIs. While these assumptions may hold in the current environment, they may not hold in the future, causing tests to fail. The technique, NonDex, detects such wrong assumptions by exploring different behaviors that may not manifest in the current implementation, but are allowed by the specification. Furthermore, the dissertation presents PolDet, a technique to detect when tests pollute their environment; such polluting tests can cause other tests to fail or pass seemingly nondeterministically because of the environment pollution rather than changes in the code. The two techniques enable developers to identify when they are introducing unreliability in their test suite and help identify the root causes of the unreliable tests. The results of the evaluation on open-source projects show the techniques are effective at identifying issues in open-source code and also that developers are eager to fix the issues and adopt the tools.

To my parents, for their love and endless support...

Acknowledgments

During the last four years I benefited from a lot of help and support from several people towards finishing this dissertation. Inevitably, I will forget to mention some of you; I apologize in advance to those who I will not mention here due to my bad memory. You all have my deepest gratitude for all the help and support that you provided and that enabled me to finish this dissertation.

First, I would like to thank my advisor, Darko Marinov. Without him this work would not have been possible. I am thankful that Darko accepted me as his student back in 2013; I hope I have met his expectations and I hope to continue to do so in the future. I am thankful to Darko for the many hours he spent talking with me and advising me on research and life. I lost track of the many nights we spent discussing research in person or over the phone; Darko's energy is almost indescribable—his passion and excitement for research was so contagious that I am not sure I would have finished without it. I appreciate the freedom Darko gave me to explore different topics, sometimes not even related to Computer Science.

I would like to thank Shuvendu Lahiri, Madhusudan Parthasarathy, and Tao Xie for all their advice during my PhD and for agreeing to serve on my thesis committee. Shuvendu provided great mentoring during my internship and made our collaboration extremely enjoyable. Madhu was an endless source of encouragement during my PhD and I appreciated the discussions that lead me to refine my taste for research. Tao constantly pushed me to improve my research by forcing me to better motivate it. I am thankful to Shuvendu, Madhu, and Tao for their help throughout my PhD and for their comments that helped improve this dissertation.

My research would not have been as enjoyable and successful as it was without my collaborators: Danny Dig, Lyle Franklin, Pranav Garg, Milos Gligoric, Farah Hariri, Sarfraz Khurshid, Shuvendu Lahiri, Jan Lahoda, Ben

Lambeth, Owolabi Legunsen, Suleman Mahmood, Madhusudan Parthasarathy, Nimrod Partush, Edgar Pek, August Shi, Tifany Yung, and Andrey Zaytsev. I look forward to collaborate with them in the future. I particularly enjoyed my collaboration with August. The two of us started our PhD at the same time, and we published at least one paper together every year since we started our PhD. Many of the ideas in this dissertation emerged through discussions we had together.

Several research-related friends from the University of Illinois helped me bounce and develop my ideas, exposed me to their research, discussed philosophy, and overall provided feedback and helped shape my taste for research. I would like to thank Angello Astroga, Blake Bassett, Jon Bell, Earlence Fernandes, Ritwika Gosh, Mengqi Gu, Wing Lam, Yun-Young Lee, Sihan Li, Radu Mereuta, Sasa Misailovic, Karl Palmskog, Daejun Park, Amarin Phaosawasdi, Stas Negara, Cosmin Radoi, Andrei Stefnaescu, Mohsen Vakilian, Wei Yang, Xusheng Xiao, and Yi Zhang.

I was fortunate throughout my PhD to benefit from great mentors during my internships: Shuvendu Lahiri (Microsoft Research) and Chang-Seo Park (Google) made my internships great and productive experiences.

I would like to thank my office mates for putting up with me on a daily basis—it is not an easy task. Lamyaa Eloussi, Owolabi Legunsen, and Deniz Arsan had to endure my bad jokes and thinking aloud throughout my PhD. My days at Illinois would not have been as enjoyable and productive without them.

I took many classes, some of which were not always directly related to my research, but I deeply enjoyed them all. I particularly enjoyed Madhu's class on Logic—it was one of the most enjoyable and intellectually stimulating class I have ever taken. I also enjoyed Vikram Amar's class on Constitutional Law which introduced me to a plethora of legal topics and involved me in many stimulating discussions.

The University of Illinois has a great staff that made my life easy through the administrative process. I would like to especially thank Elaine Wilson whose dedication in handling all of my travel logistics made the process seamless.

My mentors and friends from my undergraduate studies provided me with early research experience and other fun activities; I thank Danny Dig, Cristi Iuga, Radu Marinescu, Petru Mihancea, Marius Minea, Radu Stoita, and

Cosmin Tiru. Without their help I would not have started, let alone finish this PhD.

My friends outside of work made my life away from research enjoyable and helped me step away from research; without this my research life would not have been as productive. I would like to thank Joe and Carlota for introducing me to the joy of Argentine Tango. I would also like to thank Aarti and Linjia for being my best ballroom dancing partners and enduring my two left feet. My friends Abigail, Aradhana, Jason, Michael, Nikita, Nitasha, Ronak, Severus, and Sidharth helped me get out of the office and have some fun. During my internships I enjoyed my free time with Bogdan, Earlence, Gosia, Lopa, and Yige. My Romanian Student Club friends from the University of Illinois made my stay here feel closer to home; I thank Adina, Andrei, Codruta, Cosmin, Cristi, Gabi, and Ileana. Last but not least, I would like to thank Maria for her love and support.

Finally, and most importantly, I would like to thank my parents Cornelia and Francisc; I owe them all that I am today. They gave me the freedom and encouragement to always pursue my dreams and they did not doubt me even when I did. They constantly supported me, encouraged me, and are my biggest fans; they had to endure me being away from home and only occasionally seeing me—Thank you!

This research has been supported in part by NSF grant nos. CNS-0958199, CCF-1012759, CCF-1421503, CCF-1439957, and CNS-1646305. I was also supported by the Saburo Muroga Endowed Fellowship during my second year.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADIUS	Code Assuming a Deterministic Implementation of an Underdetermined Specification
API	Application Programming Interface
CI	Continuous Integration
LOC	Lines of Code
JSL	Java Standard Library
JSON	JavaScript Object Notation
JPF	Java PathFinder
JVM	Java Virtual Machine
SUT	System Under Test

Chapter 1

Introduction

Software is ubiquitous in today's society: from the now-classic computers, phones, and tablets, to the more recent wearables, IoT devices in our homes, smart cities, and critical infrastructure. While software is playing an increasingly more critical role in our lives, it also changes faster than ever to add new features, adapt to new requirements, eliminate bugs, improve performance, etc. The fast pace in changing software makes it imperative for developers to apply rigorous techniques to ensure the quality of the software as they change it.

Regression testing is the most widely used approach to ensure the quality of software systems while developers make changes to their code. When a developer submits a change to a software repository, the regression-testing system will run the tests and report the outcomes to the developers; this process is used to protect the software from regressions, i.e., changes that would break existing functionality. The software industry has developed large-scale regression-testing systems for in-house use, e.g., Facebook Buck [12], Google TAP [39, 89, 126, 128], Microsoft CloudBuild [27], and even released some of the tools as publicly available services, e.g., AWS CodeBuild [6]. Many more companies adopt a continuous deployment strategy which leverages regression testing as a gatekeeper for the deployment phase [101,109,115]. The open-source world in turn has also adopted a plethora of systems that perform regression testing through continuous-integration services like Travis [56,130] (overall the most used system on GitHub), AppVeyor [4] for Windows, and CircleCI [18] for Android.

Test outcomes control whether a change can be merged; if tests fail, the system does not merge the change into the codebase (although sometimes developers override this gate-keeping functionality in situations like hot-fixes). In contrast, if all tests pass, the code change is merged into the codebase. This quality-assurance process assumes that the outcomes of the tests are

reliable, i.e., if any test fails, then the change is introducing a bug that the current test suite can detect, and it is therefore beneficial that the test failed early and did not let a bug slip in. Alternatively, if all tests pass, then developers assume the change does not introduce any bug that the current test suite could detect (although there may be other bugs that the test suite misses because it is non-exhaustive). Unfortunately, this assumption does not hold in practice because many times test outcomes are unreliable [7, 8, 24, 30, 31, 44, 75, 85, 89, 90, 124, 131, 134, 137, 148].

Unreliable tests fail without indicating a fault in the system under test (SUT) or pass while missing faults that they should otherwise detect [131]. Tests whose pass/fail outcome is unreliable, i.e., tests that can pass or fail without the developer changing the code, are traditionally called *flaky tests* [85]. A recent study at Google indicates that over 40% of modules have had at least one flaky test (out of all the modules that ever passed and failed at least once) [89]. Such unreliable tests can slow down the developers by making them waste time debugging failures that are not related to their change, debugging failing tests that were potentially written by other developers and are not directly related to what they are developing, and therefore increasing the debugging effort. Alternatively, unreliable tests may also let bugs slip in (when they should fail but they pass), with potentially extreme consequences, albeit this situation is more rare [131]. Unfortunately, the false alarms not only waste developers time, but also render techniques that use historical failures unusable [75, 89], e.g., test prioritization may prioritize tests that failed in the past to run earlier than tests that did not, based on the assumption that their past failures were reliable and indicate that they are effective at detecting bugs. Furthermore, while unreliable tests are an important problem in software *practice*, we also encountered them in *teaching* software development in general and software testing in particular [120].

There are several main causes for unreliable test outcomes [85] stemming from concurrency, test-order dependency, time, I/O, environment assumptions, specific JDK or library assumptions, etc. Rothermel and Harrold precisely identified over two decades ago in *the controlled regression testing assumption* the ideal environment in which regression tests should be run: "*the only factor that may change the outcome of a test is the code change*" [110]; unfortunately, this assumption does not materialize in practice, and there are no easy solutions to control all the (nondeterministic) factors in the test

execution to guarantee that when a test fails it is indeed due to only the code change.

This dissertation aims to enable developers to make their tests resilient to changes in the environment; our techniques help developers find common causes of unreliable tests and provide debugging information that enables developers to proactively fix their unreliable tests as soon as they write them, rather than wait for a later time when problems are harder to fix. We envision that developers, whenever they add new tests or periodically, would use our techniques to check that their tests are not unreliable. We argue that fixing the tests as soon as the problems are introduced lowers the future costs of false alarms and missing bugs. This approach contrasts the current laissez-faire approach that constitutes the state of the practice, where tests are rerun until they all happen to pass and the code can be merged [89, 122].

1.1 Thesis

Our thesis is the following:

> Proactively detecting causes of unreliable tests is an effective and efficient approach for developers to use in order to prevent future slowdowns due to unreliable tests that appear in the test suites.

When the research in this dissertation started, the state of the art in research and practice for remedying unreliable tests was centered around detecting unreliable tests once they appear, isolating the environment that unreliable tests run in to ensure that tests cannot manifest as unreliable, or making the testing process resilient to unreliable tests by rerunning failing tests to ensure the tests exhibit reliable failures. It was not even clear if preventing the introduction of unreliable tests in the test suite was feasible before a failure could be exhibited in the existing environment. Further, it was not clear that it could be done efficiently; there are so many potential sources of unreliability that it was not clear whether efficiently exploring or controlling the space was feasible. Moreover, detecting unreliability may yield false positives, therefore hindering the effectiveness of techniques based on proactively detecting causes of unreliable tests. Detecting unreliable tests

early is more economical than waiting for problems to appear later down the road. Developers are best equipped to fix problems when they introduce them because the overhead of context-switching is rather minimal—they are already familiar with the test and code they wrote so fixing the test to prevent it from being unreliable is the easiest at that point. In this dissertation, we not only show that it is feasible to proactively detect causes of unreliable tests, but we demonstrate that we can do so efficiently and effectively, and we present two techniques that achieve it. First, we describe NonDex, a technique to detect tests that make wrong assumptions about underdetermined APIs by exploring behaviors allowed by the API specification [42, 43, 120]. Second, we present PolDet, a technique that detects state-polluting tests in order to prevent test-order dependency [44]. Both techniques help developers proactively detect causes of unreliability and enable them to fix the issues in their test-suites.

1.2 Terminology

In this section we define unreliable tests and also discuss some background on terminology used in other research.

1.2.1 Flaky Tests

The term "*flaky test*" has been used in practice [31, 129] (along with other informal names such as "*flakey*", "*intermittent*", "*flapper*", etc.), and was adopted in research [85, 90] to informally refer to any test that fails due to uncontrolled/unknown factors that are perceived to not be a bug in the SUT. There is no bright-line rule of what constitutes a flaky failure: a test that fails because of network is not automatically flaky but may illustrate a bug in the SUT that manifests when the network is not available; similarly, a test that fails due to concurrency need not be flaky, but rather may expose a bug only under certain thread schedules (albeit both examples would strictly speaking meet the definition of "intermittently failing or passing without any change to the code"). The key intuition is that flaky tests expose bugs in the test (infrastructure) rather than the SUT.

1.2.2 Unreliable Tests

Throughout this dissertation we use the term *unreliable tests* to refer not only to tests that are flaky, but also to any tests that may fail intermittently in the future, but may have never had a flaky failure yet. Such unreliable tests may make undocumented assumptions that hold in the current environment, but they are not in any way guaranteed to hold in the future, e.g., performance of the underlying hardware or application programming interfaces (APIs) with underdetermined specifications. In other words, any test that does not either enforce the controlled regression testing assumption or make its oracle robust to allowed but uncontrolled changes in the environment is unreliable. Under this broad definition, is there any test that is reliable? Yes, of course! For example, tests that mock the network when they depend on the network control for network unreliability and therefore are reliable with respect to the network (mocks could also simulate network outages, if tests are meant to test for that). Note that we do not require test executions to be deterministic, but rather we require that the oracles are robust to allowed but uncontrolled changes in the environment for a test to be reliable.

1.2.3 Practical Considerations

One pragmatic consideration in dealing with unreliable tests is whether it is practical to require all tests to be reliable, because that obviously incurs a cost. There is a balance to be achieved between making tests resilient to issues that may or may not arise in the future and addressing current issues. We believe tools should empower developers to make this decision by providing the appropriate amount of information needed to make an informed decision. While it is rather hard for tools to predict the future and foresee whether something that is uncontrolled by the test will end up making the test fail, tools can still assist the developers to make a better decision. For example, when a developer makes an assumption on the environment that is not supported by the specification, a tool could inform the developer how strong the assumption is (the stronger the assumption, the likelier for it to not hold in the future). In general, tools that help developers identify causes of unreliability need to offer a way for developers to prioritize or focus their attention on the most relevant issues.

1.3 Wrong Assumptions on Underdetermined APIs

Several commonly used Java APIs, both in the Java Standard Library and in commonly used third-party libraries, have underdetermined specifications. Following Liskov [84], we say that a specification is underdetermined if it allows implementations to return different results for the same input, even if each implementation is itself deterministic and always returns the same result for the same input. We refer to an API with such a specification as an *underdetermined API*. An example underdetermined API is the `iterator` method in `java.util.HashSet`, whose Javadoc specification states, "The elements are returned in no particular order" [53]. Similarly, libraries for generating JavaScript Object Notation (JSON) typically do not guarantee any order for elements in a JSON document [65]. Having such underdetermined specifications has advantages because it gives implementers of the underdetermined APIs the flexibility to optimize the various implementations of the API for different goals, e.g., they may optimize performance in different ways. However, it is important to precisely document underdetermined specifications in the API documentation to express *all* expected behaviors of all the implementations of an API.

Unfortunately, even when underdetermined APIs have precise documentation, developers do make wrong assumptions about the underdetermined APIs. While such APIs could allow even nondeterministic implementations, each typical implementation is deterministic, i.e., two runs of the same implementation (in the same environment) give the same result for the same input. For example, two runs of a program that iterates over a `HashSet` may return the elements in the same order. However, such deterministic implementations can mislead the developers of API clients, who may assume that *all* API implementations are guaranteed to behave in the same deterministic manner. For `HashSet`, while one Java version could provide a deterministic iteration order, different Java versions provide different iteration orders (e.g., the order in Java 7 differs from the order in Java 8). If clients of an underdetermined API assume stronger-than-specified guarantees, the resulting code can fail when the API implementation changes, albeit the specification remains the same. A well-known example of such wrong assumptions is that many projects with JUnit tests relied on a particular order in which test methods are executed; when these projects upgraded from Java 6 to Java 7,

many tests failed because the order changed from Java 6 to Java 7 [69], albeit the specification of the API producing the order did not change between the two versions.

The state-of-the-practice in detecting the negative effects of wrong assumptions on underdetermined APIs is rather *reactive*. Most developers discover such assumptions only after failures happen (e.g., after environments or libraries are changed). Unexpected behaviors cause unreliable tests, which can pass or fail seemingly without any changes to the code. An unreliable test that assumes a certain behavior, which is not guaranteed by the API specification, can fail when the API implementation changes. The developers of several projects followed a reactive practice in the past by spending a lot of time fixing their own code as a result of test failures due to wrong assumptions [35, 37, 69, 70, 86, 91].

We describe NonDex, a technique to *proactively* detect wrong assumptions on underdetermined APIs by exploring different *allowed* behaviors during test execution. For example, `HashSet`, with its underdetermined iteration order, NonDex randomly explores different iteration orders, which can proactively detect failures of tests that iterate over `HashSet`, either directly in the test code itself or in the SUT. NonDex can also systematically explore the space of all possible behaviors allowed by underdetermined specifications when it is tractable. Once a test fails during exploration, that failure demonstrates that the code made some wrong assumption. NonDex also helps in debugging by pinpointing the location where the assumption was made; the debugging feature searches for the dynamic invocation of the API whose exploration caused the failure. Developers can run NonDex, e.g., during continuous integration, to check for wrong assumptions on Java APIs. It is often more cost-effective to detect bugs proactively, right when they are introduced, rather than reactively, after they manifest when the environment changes.

Our evaluation of NonDex on 195 open-source projects downloaded from GitHub and 72 student submissions from one homework assignment in a recent offering of our software-engineering course shows that NonDex is effective and efficient at pinpointing wrong assumptions in tests. We find NonDex to be highly effective at detecting unreliable tests in both open-source projects and student submissions. NonDex detected 60 unreliable tests in 21 of the 195 open-source projects. We reported to developers the

issues we found in 13 pull requests, and developers accepted 12. Further, the Checkstyle project, in which we found 5 bugs, integrated NonDex into their continuous-integration configuration to run on every push [15].

1.3.1 Main Contributions

The work on NonDex, detecting wrong assumptions on underdetermined APIs, makes the following contributions:

- Defines the problem of code that makes wrong assumptions on underdetermined APIs and identifies it as a cause of unreliable tests.
- Systematically explores, quantifies, and characterizes the state spaces of programs using underdetermined specifications.
- Presents the development of NonDex, a tool to explore underdetermined APIs and identify wrong assumptions.
- Evaluates NonDex on real-world programs.

1.4 State Pollution

One common cause of unreliable tests in regression test suites is dependency among tests [7, 58, 85, 105, 137, 148]. 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) [85], and that most of these dependencies are on the heap state, reported to range from 53% [85] to 61% [148] of all test dependencies. These dependencies make the outcome of regression test-suite runs unreliable: even for the same version of the SUT, the tests could pass when executed in one order but fail when executed in another order, leading to unreliable tests [85, 93, 148].

When tests fail due to test-order dependency, it is hard to pinpoint the root cause of the dependency, i.e., identify 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 here. First, when the test t' seemingly nondeterministically fails or passes for the same code, the culprit is not necessarily the test t' but the polluting test t, which makes debugging harder.

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 [70]. 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, e.g., as reported by several blog posts [69, 86, 91]. 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 describe 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 the form of failing tests. Conceptually, PolDet is a rather simple idea that 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, but it is not the most important constraint: 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, a prior study [85] shows that 78% of the polluting tests pollute the shared state right when they are added (i.e., only 22% of the polluting tests start polluting due to later changes in the test code or the SUT).

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.

1.4.1 Main Contributions

The work on PolDet, detecting state-polluting tests to prevent test dependencies, makes the following contributions:

- Defines the problem of state pollution and identifies it as a cause of unreliable tests.
- Presents the development of PolDet, a technique to identify tests that pollute the state shared across test executions and precisely pinpoint the polluted state.
- Evaluates PolDet on real-world programs.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows.

Chapter 2: Detecting and Debugging Wrong Assumptions on API Specifications

This chapter presents our work on NonDex, a technique to detect unreliable tests that make wrong assumptions on underdetermined APIs.

Chapter 3: Detecting State-Polluting Tests to Prevent Test Dependency

This chapter presents our work on PolDet, a technique to detect state-polluting tests, which can cause unreliable tests.

Chapter 4: Related Work

This chapter gives an overview of the related work in the area of regression testing in general and test reliability in particular.

Chapter 5: Future Work

This chapter presents several directions for future work that build on this dissertation.

Chapter 6: Conclusions

This chapter concludes the dissertation.

Chapter 2

Detecting and Debugging Wrong Assumptions on API Specifications

In this chapter we describe our approach to detecting tests that are unreliable because of wrong assumptions on underdetermined APIs. Section 2.1 provides an overview of the problem of ADIUS code, Section 2.2 precisely defines underdetermined specifications, Section 2.3 describes NonDex, our randomized technique for detecting wrong assumptions on underdetermined specifications, Section 2.4 presents some of the implementation details of NonDex, Section 2.5 presents the results of our evaluation of NonDex, and Section 2.6 presents our NonDex implementation that uses systematic exploration and our analysis of the state spaces resulting from exploring the underdetermined APIs.

2.1 Overview

Underdetermined specifications allow several different results for the same input. Underdetermined specifications are not uncommon for many methods, including in the standard libraries of many programming languages. For example, the specification for the `malloc` function in C allows to return a pointer that is not guaranteed to have any specific value (if there is space on the heap otherwise returns `NULL`); similarly, the specification for the `Object::hashCode` method in Java can return any integer and is not guaranteed to return a specific value. Underdetermined specifications are not restricted to simple APIs. For instance, the order in which elements of a set are returned by an iterator is not-specified—it can be any order. As another example, the order in which the entries resulting from a SQL query are returned is also sometimes not specified—it depends on the query. Also, any numerical API with ϵ-tolerance is underdetermined. Such specifications give implementers more freedom to develop various implementations for different

goals, e.g., to optimize performance, while still satisfying the specification.

Even when specifications allow for nondeterminism in implementations, typical implementations of such specifications are often deterministic, with respect to certain controlled sources. For example, `malloc` could return the same pointer on the same platform in two different runs (if one controls for all other sources, such as address space layout randomization, timing/multi-threaded effects, etc.). Similarly, `Object::hashCode` could return the same integer (if one controls for all other sources, e.g., OpenJDK Java 8 could return a deterministic value on the first call if the underlying `random` implementation in C is deterministic). Deterministic implementations are good because they allow easier debugging [9, 97]. The implementation of `HashSet` is such that iterating through the elements returns them in a deterministic order for one Java version, but that order can change between Java versions.

Code that *A*ssumes a *D*eterministic *I*mplementation of an *U*nderdetermined *S*pecification—which we call *ADIUS* code—is often bad. Such ADIUS code can behave unexpectedly when the implementation changes, even if the specification remains the same. For example, Java code that would assume `new Object().hashCode()` to always return `366712642` on the first call (as it happens to return for Oracle Java version 1.8.0_25-b17 running with glibc 2.12) is ADIUS and fairly not robust: any change in the Java implementation could easily invalidate that assumption. Similarly, code that assumes a specific iteration order of a `HashSet`, e.g., that a `HashSet` with elements `1` and `2` will be always represented as a string `{1, 2}` rather than `{2, 1}`, is ADIUS and not robust: the Java implementation of `HashSet` can change such that the iteration order of the elements changes and the string differs.

While ADIUS code can be a problem in general we are particularly interested in *unreliable tests*, which are tests that seem to nondeterministically pass or fail. Unreliable tests are bad as they can mask bugs (pass when there are bugs) or raise false alarms (fail when there are no bugs). A test that executes ADIUS code can be unreliable if it assumes that some values are deterministic even if they can change: when the assumptions hold, the test passes, but when the assumptions do not hold, the test may fail. Not all unreliable tests are due to ADIUS code, e.g., a test asserting that a file system contains `/tmp` could pass on one machine but fail on another. Unreliable tests are emerging as an active research topic, with recent work on characterizing [85], detecting [8, 24, 44, 54, 58, 148], and avoiding [7, 75] unreliable

tests. However, this work is the first to investigate ADIUS code as a cause for unreliable tests.

While the present works identified unreliable tests are an important problem in software *practice* and *research*, we also encountered them in *teaching* software development in general and software testing in particular. Typically, the teaching staff grades students' solutions to programming assignments using automated tests. The automated tests are either written by the teaching staff, or sometimes they are written by the students as part of the assignment. In either case, these tests can be unreliable, and as a result students with correct solutions may have failing tests (resulting in lost points, discussions with teaching staff, revision of grades, etc.), and students with incorrect solutions may have passing tests (resulting in full points, and extremely rarely in students complaining about not losing points when they should have lost). Albeit teaching staff can make their tests to be reliable, students are often asked to write their own tests that can be rather unreliable. We discuss more details from one recent course in Section 2.5.3 and give just a brief anecdote here.

We taught several courses on software engineering that require students to run their tests in Jenkins, a continuous integration system [60]. One representative example comment about an unreliable test is: "When we [...] test locally with Eclipse[...] all tests passed. But when we commit [...] and run tests on Jenkins, [some test fails]. We had no idea what happened here." Another example is: "I got this [test failure] information in the last few lines [... I]s it supposed to be like this or am I making something wrong here?". One way to reduce or avoid the problem of unreliable tests is to reduce the variability in the environments, e.g., (1) require all students to use virtual machines that run the same OS with the same Java version, but students prefer working on different local environments, and (2) more importantly, in the real development practice, have developers use the same system for development, testing, and deployment, but doing so just postpones the problem of detecting unreliable tests until later when the code itself inevitably evolves.

We present a novel technique, called NonDex, to detect unreliable tests due to ADIUS code. We implemented NonDex for Java, but it can be easily generalized to any other language (in fact one undergraduate student implemented a prototype for Python). In a nutshell, we identified 41 methods with

underdetermined specifications as discussed in Section 2.3.1, wrote models for these methods to produce various nondeterministic choices, and used an execution environment that can explore various combinations of these nondeterministic choices. Our NonDex tool, instruments the regular Java APIs to explore choices and reruns the test suites multiple times from scratch while exploring different behaviors.

We evaluated NonDex on two sets of programs: (i) 195 open-source projects from GitHub and (ii) 72 student submissions from one homework assignment in our software-engineering course. We find NonDex to be highly effective at detecting unreliable tests in both open-source projects and student submissions. NonDex detected 60 unreliable tests in 21 of the 195 open-source projects. Because our experiments used some older project revisions, three of these tests had been already fixed by the developers in the latest revision. (This fixing additionally confirms that unreliable tests are important and that developers are willing to address them.) We confirmed that 57 tests are still present in the respective projects' latest revision. For student submissions, NonDex detected that 34 submissions, representing almost half of 72 considered, fail due to some ADIUS code, with a total of 110 unreliable tests detected. It is important to note that the homework assignment was designed a few years ago by a teaching assistant who had no knowledge of our research on unreliable tests. We already devoted time in our teaching to expose students to NonDex and teach them to better detect and avoid ADIUS code and unreliable tests; increased training that raises awareness about unreliable tests may help the most with preventing unreliable tests.

This chapter makes the following contributions:

- **Problem.** We define the problem of ADIUS code, identify it as a cause of unreliable tests, and raise awareness about the problem of unreliable tests in both software development practice and software engineering education.

- **Technique and Implementation.** We propose a simple technique for detecting unreliable tests caused by ADIUS code and describe our nondeterministic models and a tool that embody this technique.

- **Evaluation.** We evaluated our NonDex technique on 195 open-source

Java projects and 72 student code submissions. NonDex detected 57 previously unknown unreliable tests in open-source projects and three unreliable tests that had been already fixed by the open-source software developers. NonDex also detected 110 unreliable tests in student submissions.

2.2 Underdetermined Specifications

An underdetermined specification allows for multiple implementations that can yield different outputs when executed with the same input; we consider "input" in a broad sense to include all interactions of the code with its environment. For example, consider the specification for the method `File::list` that returns a `String` array with the names of all the files and directories present in the directory on which the method was invoked. The method's Javadoc specification [29] states *"There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order."* This specification allows implementations to return names in any order even when executed with the exact same input (the state of the file system), hence this specification *is* underdetermined. In contrast, consider the specification for the method `File::exists` that returns a `boolean` value indicating whether or not the file on which the method was invoked exists on the file system. This specification *is not* underdetermined; while the returned value depends on the input (the state of the file system) and can be `true` or `false` on different machines, when executed on the same input (including the file system), any implementation that conforms to the specification must return the same value.

2.2.1 An Example Unreliable Test

Code that (transitively) calls methods with underdetermined specifications can be ADIUS and lead to unreliable tests. Figure 2.1 shows an example unreliable test simplified from a student submission. The `Book` class has two fields, `title` and `author`. The method under test, `getStringRepresentation`, uses a third-party JSON library that turns an object into a string. The test asserts that the resulting string equals a hard-coded string that has

```
1  class Book {
2    String title;
3    String author;
4    String getStringRepresentation() { ... }
5  }
6  class BookTest {
7    @Test
8    public void testGetStringRepresentation() {
9      Book b = new Book("book", "name");
10     assertEquals("{\"title\":\"book\",\"author\":\"name\"}",
11                  b.getStringRepresentation());
12   }
13 }
```

Figure 2.1: Example unreliable test simplified from student code

the two fields in a particular order, first `title` and then `author`. However, the library uses a `HashMap` to store the mapping from fields to values, and iterates over this map to produce the resulting string. The iteration order over elements in a `HashMap` is not specified, so while this test can pass for one implementation, it can fail for another implementation that puts `author` before `title`. NonDex can detect such wrong assumptions by running the tests with different choices for the `HashMap` iteration order.

2.2.2 Levels of Underdeterminedness

Some underdetermined specifications, especially when written in a natural language, can allow for multiple *levels* of underdeterminedness. Figure 2.2 presents an example: the class `HashSet` has an underdetermined specification that can be (mis)interpreted in different ways. The Javadoc specification [53] states *"[HashSet] makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time."* Hence, the order of the elements in the array returned by `HashSet::toArray` can be any. The code first constructs an `Integer HashSet` object `s` with the elements `1` and `2` (lines 1–2). Because the specification allows for any iteration order, a deterministic implementation could return either of the two orders shown in the two assertions on lines 4 and 5, and one of the assertions should pass, while the other should fail.

```
1  Set<Integer> s = new HashSet<Integer>();
2  s.add(1); s.add(2);
3  Integer[] a = s.toArray();
4  // assertArrayEquals(a, new Integer[]{1, 2});
5  // assertArrayEquals(a, new Integer[]{2, 1});
6
7  // assertArrayEquals(a, s.toArray()); // differ from "a"?
8
9  s.contains(1); // observer calls on "s" may matter
10 // assertArrayEquals(a, s.toArray());
11
12 s.add(3); s.remove(3); // "s" modified and restored
13 // assertArrayEquals(a, s.toArray()); // differ from "a"?
14
15 Set<Integer> t = new HashSet<Integer>();
16 t.add(1); t.add(2); // "t" constructed same way as "s"
17 // assertArrayEquals(a, t.toArray()); // differ from "a"?
18
19 Set<Integer> u = new HashSet<Integer>();
20 u.add(3); u.add(4); // "u" with different elements
21 Integer[] b = u.toArray();
22 // assertEquals(a[0] < a[1], b[0] < b[1]); // order?
```

Figure 2.2: Different levels of underdetermineness may fail different assertions

Whether the remaining assertions pass or fail is more open to different interpretations of this underdetermined specification. First, a developer could assume that two iterations on the same unchanged set object should yield the same order. However, the specification states that the order can vary "over time", which could mean that the order in which elements are returned can change from one invocation to another even for the same set. Hence, the assertion on line 7 may get a different order and fail. Second, one could assume that the order should not change if the set is only read but not modified. Hence, the assertion on line 10 could pass or fail depending on whether the assumption holds. Third, one could assume that if a set is modified and then restored to its original state, the order in which the elements are iterated can change from that before the modification of the set. Hence, the assertion on line 13 could either pass or fail. Fourth, one could assume that two sets constructed in exactly the same way would yield the same order,

but if that does not hold, the assertion on line 17 can fail. Fifth, one could assume that elements are iterated in the order of addition that is consistent with the original set `s`; line 19 creates a new set using different elements but added in the same order as in set `s`. One could assume that both sets will be iterated in the same order—in which elements are added, or the natural order; depending on whether this assumption holds, the assertion on line 22 can pass or fail. (This final assumption is not completely unrealistic; the specification for `LinkedHashSet` indeed guarantees the iteration order over elements to be the same as that in which the elements are added [83].)

None of the (mis)interpretations are unambiguously supported by the documentation, but some of them may correspond to more reasonable assumptions than others. Developers may be more willing to remove wrong assumptions that are, in their intuition, least reasonable. While the specification allows for a lot of nondeterminism, most implementations are not nondeterministic; identifying assumptions that are more likely to break serves also as a prioritization mechanism when deciding which assumptions to remove.

2.3 Technique

Our NonDex technique detects unreliable tests due to ADIUS code making wrong assumptions on underdetermined specifications. Section 2.3.1 describes how we identified several underdetermined APIs in the Java Standard Library. Section 2.3.2 describes the models we developed for those underdetermined APIs. Section 2.3.3 presents some implementation details of NonDex.

2.3.1 Identifying Underdetermined APIs

Finding methods which have underdetermined *specifications* is challenging; in particular, one cannot easily look for nondeterministic *implementations* as individual implementations are deterministic most of the time. Rather, nondeterminism occurs when underdetermined specifications allow *multiple* implementations to behave differently from one another while still meeting the specification, even if each implementation is deterministic. For example, upgrading from Java 6 to Java 7 changed the order in which the method

Table 2.1: Underdetermined APIs in the Java Standard Library

Class(es)::method(s)	Kind
java.lang.Object::hashCode	*random*
java.util.{Weak,Identity,}HashMap::keySet, values, entrySet	*permute*
java.util.concurrent.ConcurrentHashMap::	*permute*
keySet, values, entrySet, keys, elements	
java.util.PriorityQueue::iterator, toArray, toString	*permute*
java.util.concurrent.{Delay, PriorityBlocking}Queue::	*permute*
iterator, toArray, toString	
java.io.File::list, listFiles, listRoots	*permute*
java.lang.Class::	*permute*
getClasses, getFields, getDeclaredFields, getConstructors	
getAnnotations, getMethods, getDeclaredConstructors	
getDeclaredMethods, getDeclaredClasses, getDeclaredAnnotations	
java.lang.reflect.Method::	*permute*
getParameterAnnotations, getExceptionTypes	
getGenericExceptyonTypes, getDeclaredAnnotations	
java.lang.reflect.Field::	*permute*
getAnotationsByType, getDeclaredAnnotations	
java.text.DateFormatSymbols::getAvailableLocales	*permute*
java.text.BreakIterator::getAvailableLocales	*permute*
java.text.Collator::getAvailableLocales	*permute*
java.text.DecimalFormatSymbols::getAvailableLocales	*permute*
java.text.NumberFormat::getAvailableLocales	*permute*
java.text.DateFormat::getAvailableLocales	*permute*
java.text.DateFormatSymbols::getZoneStrings	*extend*

`Class::getDeclaredMethods` from the Java `Reflection` API returned the list of methods in a class. JUnit uses the `Reflection` API for obtaining the list of methods to run. Thus, when run on Java 6, methods were returned in one order, but were returned in a completely different order in Java 7. This seemingly innocuous change caused tests run by JUnit to fail [70] due to test-order dependencies [7, 8, 44, 58, 75, 148]. Finding underdetermined APIs solely from the executable code is infeasible; one must reason about the specification itself to find if a specification is underdetermined because the method implementation need not be nondeterministic. This makes it inherently hard for any static or dynamic analysis technique to find such underdetermined APIs from one implementation.

To find underdetermined APIs in the Java Standard Library, we first searched for methods whose documentation indicates that they may have such specifications and then carefully reasoned from their Javadoc to determine if their specifications are indeed underdetermined. We used two queries, based on (1) Javadoc keywords and (2) return types. Specifically, the first query searches through Javadoc for the following keywords that could indicate underdetermined specifications: *"order"*, *"deterministic"*, and *"not specified"*. The second query searches for all public methods that return arrays. These queries produced many false positives, e.g., because not every method that mentions *"order"* is underdetermined, and some methods that return arrays must return elements in a specified order. Our search is definitely not complete, and we leave as future work to develop better approaches to find underdetermined specifications.

After inspection, we found the underdetermined APIs summarized in Table 2.1. We tabulate the class name(s), method name(s), and the kind of specification underdeterminedness. We found three kinds, which we call "*random*", "*permute*", and "*extend*". For *random*, the specific `int` returned by `Object::hashCode` is not specified, so relying on it to return some specific value is ADIUS. For *permute*, the specifications of some methods that return arrays or collections can have an unspecified order of elements. For *extend*, the specification of one method specifies just a lower bound on the length of the returned array but not the precise length.

We next describe some specific underdetermined APIs that we found. For class `Object`, it is well known that `hashCode` is nondeterministic. In contrast, the less known inner class `HashMap$HashIterator` does not have a specified it-

eration order and can return the map's elements in any order; this inner class is exposed to the clients via some methods from Table 2.1 (`keySet`, `entrySet` and `values`), so code that calls these methods can be ADIUS. Moreover, `HashMap` is the underlying data structure for many other data structures, e.g., `HashSet`; we do *not* count separately the other underdetermined APIs, e.g., `HashSet::iterator`, that could lead to ADIUS code. However, changing one piece of code in `HashMap` can affect many types of objects. The specification for iterating through `WeakHashMap`, `IdentityHashMap`, `PriorityQueue`, and `ConcurrentHashMap` is similar to the specification for iterating through `HashMap`. The `File` class has multiple `list` methods that return an array of files in a given directory; the specification allows these arrays to be in any order. The classes `Class`, `Method`, and `Field` provide several reflection methods that return arrays of elements, e.g., an array of all methods in a class or an array of all annotations on a field; the specifications for most of these methods allow these arrays to be in any order. The classes in the package `java.text` return arrays of available locales and zone strings which can be in any order. Finally, the `DateFormatSymbols::getZoneStrings` method returns an array of arrays, each of which has length at least five; these arrays are indeed of length five in Java 7, but their length changed to seven in Java 8.

We also briefly explored an option of automatically finding underdetermined APIs in the Java Standard Library. We attempted to automatically generate tests that could show a behavior difference between Java 7 and Java 8. To that end, we used Randoop [99] to generate tests. We first instructed Randoop to generate tests for a large number of classes in the Java Standard Library on Java 8 and then ran the generated tests (that still compile) on Java 7. However, the tests (and assertions) that Randoop generated were unable to detect *any* changes in the behavior of the two Java versions. Even focusing Randoop on only one class, `HashMap`, did not generate (after one hour) a single test for Java 8 that would fail when run on Java 7. The reason is that the search space for `HashMap` is large, with 29 methods, and only a tiny ratio of method sequences in that space can show the difference between the two Java versions. In the end, we were able to generate tests that can reveal differences between Java 8 and Java 7 only after manually focusing Randoop to only four methods in the `HashMap`. In the future, one could try more advanced techniques for finding differences among (Java) implementations [17].

2.3.2 Nondeterministic Models

We first discuss different models with different levels of nondeterminism that could satisfy an underdetermined specification. We next discuss approaches to model nondeterminism in specifications of methods that return arrays whose order could permute. We finally describe the models for methods whose return values can be randomized, or whose return arrays can have their sizes extended.

We developed models to explore potential nondeterminism allowed by the underdetermined specifications of the identified methods. NonDex has a model for each underdetermined API, and each model has up to four different levels of nondeterminism: FULL, ID, EQ, and ONE.

FULL is the most nondeterministic level as it alters the regular execution most aggressively. Every invocation of an underdetermined API, even with the exact same object, can return a different result because the model allows all the different behaviors to be explored. This level corresponds to checking that code makes no wrong assumption.

ID is a level that constrains FULL to only explore the same behavior on the same unchanged object for all different invocations of the same underdetermined API (this same behavior can be different from the native behavior, but it is consistent across two different invocations). In other words, this behavior corresponds to the intuition that implementations are largely deterministic and explores only behaviors that preserve deterministic results as long as the input to the API does not change.

EQ is a level that further constrains ID to explore the same behavior for all input objects that are equal (not necessarily the same object, although the same object is equal to itself) but allows different behaviors to be explored for objects that are not equal.

ONE is the most deterministic level after the first invocation. It does not introduce any additional nondeterminism to the execution; it only changes the executions to explore a different behavior, which it keeps as deterministic as the original execution would.

Table 2.2 shows which of the assertions in Figure 2.2 (referred to by line numbers in the column headers) can fail under the four levels that NonDex supports.

Either of the assertions on lines 4 and 5 can fail on any of the levels, be-

Table 2.2: Levels that can fail (✓) assertions from Figure 2.2

Levels \ Assertion	4,5	7	10	13	17	22
FULL	✓	✓	✓	✓	✓	✓
ID	✓	-	-	✓	✓	✓
EQ	✓	-	-	-	-	✓
ONE	✓	-	-	-	-	-

cause all levels explore different orderings of the elements in the `HashSet` than the orders in both assertions (recall that in a deterministic JVM, one of the assertions will always pass and one will always fail, whereas in our exploration, they can both pass, both fail, or swap the order of pass/fail during different executions). Assertions 7 and 10 can only fail in FULL because they will only fail in levels that allow different orders on two successive invocations (including invocations of observer methods). Assertions 13 and 17 can fail in FULL and ID because these levels explore different orderings of objects based on their identity. Assertion 22 can fail in all levels except ID which would permute elements in the same way for both objects.

For underdetermined APIs of the *random* kind, when using the `Object` class, it should not be assumed that the `hashCode` method returns a specific integer value. In particular, it should not be expected to return the same value across different runs. However, the returned value should be unique for an object in the same run. We model these potentially different values by randomizing the value returned by `hashCode` on the initial invocation and then caching this value for future calls. For underdetermined APIs of the *extend* kind, we model the possibility that the lengths of arrays returned are increased nondeterministically on any invocation.

2.3.3 Implementations of Models

We implemented our NonDex technique for the Java programming language by instrumenting the regular implementations of the APIs in the Java Standard Library, in particular, the OpenJDK JVM version b132, which corresponds to Java 8. We also downloaded the publicly available OpenJDK code, which consists of the C/C++ code that implements the core virtual

```
1  class HashMap {
2    Node<K,V>[] table; // internal table of key-value pairs
3    int modCount = ... // stores modification count
4    class Node<K,V> { ... } // stores a key-value pair
5    class HashIterator { // inner class of HashMap
6      Node<K,V> next; // next entry to return
7      Node<K,V> current; // current entry
8      int expectedModCount; // for fast-fail
9      int index; // current slot
10     final boolean original_hasNext() {
11       return next != null; // original code
12     }
13     final Node<K,V> original_nextNode() {
14       // original code, advances "next", "current", and "index"
15     }
16     final void original_remove() {
17       // original code, can modify the entire "table"
18     }
19     HashIterator() {
20       // The code is shown in Figure 2.4
21     }
22     Iterator<Node<K, V>> NonDex_iter;
23     public final boolean hasNext() {
24       return NonDex_iter.hasNext();
25     }
26     final Node<K, V> nextNode() {
27       if (modCount != expectedModCount)
28         throw new ConcurrentModificationException();
29       current = NonDex_iter.next();
30       return current;
31     }
32     public final void remove() {
33       original_remove();
34     }
35   }
36 }
```

Figure 2.3: NonDex model for `HashMap`

```
1  HashIterator() {
2    expectedModCount = modCount;
3    Node<K,V>[] t = table;
4    current = next = null;
5    index = 0;
6    if (t != null && size > 0) { // advance to first entry
7      do {}
8        while (index < t.length && (next = t[index++]) == null);
9    }
10   /** all (and only) the code below is NonDex extension **/
11   List<Node<K, V>> original = new ArrayList<>();
12   while (original_hasNext())
13     original.add(original_nextNode());
14   NonDex.shuffle(original, HashMap.this);
15   NonDex_iter = original.iterator();
16 }
```

Figure 2.4: NonDex model implementation for `HashIterator` constructor

machine, and the Java code for the Java Standard Library; for `hashCode` we modified the C++ implementation to return different values. For each of the other APIs we apply instrumentation that calls NonDex and shuffles the returned value in the library code.

Figure 2.3 shows the model we use for exploring different orderings when iterating over a `HashMap` object. The iteration is done using the inner class `HashIterator`. We kept the original code and renamed its methods with the prefix `original_`. The constructor, with its implementation presented in Figure 2.4, computes, starting at line 11, the order that the original code would have normally returned, applies a permutation depending on the NonDex level, and stores the resulting order in an `Iterator` object called `NonDex_iter`. The `next` method returns the elements in the permuted order and updates the internal state as required. The `hasNext` method is now based on the new elements order and delegates the call to the new `Iterator` object. The `remove` method just delegates the call to the original method that changes the table.

Figure 2.5 shows how NonDex performs shuffling depending on the level. For FULL, NonDex uses the same `Random` object to perform all shufflings, which means two consecutive shufflings of the same object can yield different orders. For ID, NonDex considers a value representing the identity of the

```
1  class NonDex {
2    static Level level; // FULL, ID, EQ, or ONE
3    int seed = ...;
4    static Random full = new Random(seed);
5
6    public static <T> List<T> shuffle(List<T> l, Object o) {
7      int size = l.size();
8      Random rand = (level == FULL) ? full : // Full
9          (level == ID) ? new Random(seed +
10                 System.identityHashCode(o)) : // Same object
11         (level == EQ) ? new Random(seed + o.hashCode()) : // Equal object
12         (level == ONE) ? new Random(seed); // Once
13     for (int i = 0; i < size - 1; i++) {
14       int s = rand.getNext(i, size);
15       if (s == i) continue;
16       T obj = l.get(i);
17       l.set(i, l.get(s));
18       l.set(s, obj);
19     }
20     return l;
21   }
22 }
```

Figure 2.5: Implementation of exploration

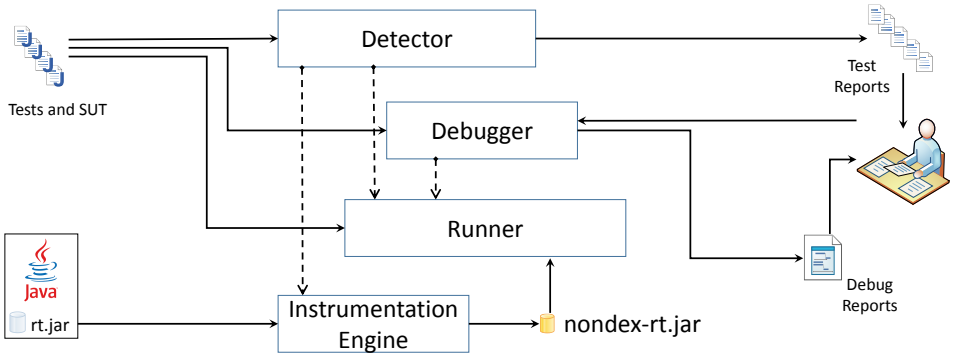

Figure 2.6: High-level architecture of the key NonDex components; see Section 2.3 for the description

object; for `HashMap`, this value is the sum of the identity hash code (a likely unique number for each Java object provided by the Java Virtual Machine) and the `modCount` field that counts the number of modifications, which means that for the same object with the same `modCount`, NonDex uses a fresh `Random` object with the same seed, therefore this `Random` object returns the same sequence of values for permuting the order. Similarly, for EQ, NonDex considers the value-based hash code of the object to produce a new `Random` object. For ONE, NonDex always creates a fresh `Random` object using the same seed therefore producing always the same ordering.

2.4 Implementation

NonDex is a Maven plugin that has two user-facing phases: (i) *detection* finds tests that pass without NonDex but fail when NonDex explores different allowed behaviors—such failures indicate wrong assumption(s) made on underdetermined APIs; and (ii) *debugging* searches through detected failures to find the underdetermined APIs on which wrong assumptions were made and to identify the invocation(s) making such assumptions. Currently, NonDex exploration handles 41 underdetermined APIs that we manually identified from the following packages `java.lang`, `java.util`, `java.io`, and `java.text` shown in Table 2.1; we first identified only 30 of these underdetermined APIs in our earlier paper [120, Table I][1].

[1]The publicly released NonDex tool does not handle the native `hashCode`, because it did not expose any bugs during our experiments and would unnecessarily complicate the tool implementation and portability.

Internally, NonDex consists of four components: (1) the *instrumentation engine* modifies the API classes in the Java Standard Library to add code for random exploration, (2) the *runner* executes the program on the instrumented library, (3) the *detector* reruns the program a specified number of times to randomly explore different behaviors, and (4) the *debugger* identifies the API invocation(s) where a wrong assumption was made. Figure 2.6 shows an architectural overview of the NonDex components, and the following subsections describe each component.

2.4.1 Instrumentation Engine

The goal of the instrumentation engine is to modify the Java Standard Library classes from the Java Standard Library to allow random exploration. The challenge is to develop instrumentation that can automatically handle a large number of Java versions. For our original prototype [120], we manually modified the Java sources of the relevant files for one version, compiled them, and used them in place of the original files. However, this solution was brittle, because the tool would often not work unless the exact same Java version (e.g., `1.8.0-b132`) was used for the run as the version for which we manually modified the sources. The reason our initial prototype did not work was that some internal parts of the modified files changed between Java versions, even when the signatures of the public APIs we modified did not change. Hence, we developed our current solution based on instrumentation which is much more robust, and we have tested it on 14 different versions of OpenJDK and Oracle's JDK implementations of Java 8, on Linux, OS X, and Windows.

The instrumentation engine takes as input the `rt.jar` file containing the classfiles of the Java Standard Library that will be used when running the tests. The instrumentation engine selects from `rt.jar` the classfiles corresponding to the APIs that should be modified to add random exploration. For APIs that should be modified and return an array, our instrumentation simply adds a call to our NonDex helper method to explore different orders of the returned array (effectively randomly permuting the array before returning it). This modification is robust as long as the type signature of the API does not change. The instrumentation is much more involved for the `(Concurrent)HashMap` classes, because their iterators are lazy, implemented

as private data structures that change even within the same Java major version, e.g., the `HashMap` iterator was implemented using a class called `Entry` until OpenJDK version `1.8.0-b108` [59] and using a class called `Node` since then; we developed customized instrumenters that can generate appropriate modified code based on whether `rt.jar` uses `Entry` or `Node`. This modification would need to change in the future if the Java Standard Library implements `HashMap` using a third approach. We used ASM [11] to implement all classfile manipulation.

Performing instrumentation from scratch on every run is unnecessary, so we reuse each previously instrumented class in subsequent runs, as long as the instrumented class from `rt.jar` did not change. (The original class does not change until/unless the user switches to another version of Java.) To decide when to reuse the instrumented classes, NonDex stores for each instrumented class the checksum of the classfile from the `rt.jar` that was instrumented.

2.4.2 Runner

The runner is a thin layer of code that enables random execution for APIs instrumented by NonDex. On every invocation of an instrumented API, the runner randomly chooses one behavior from the behaviors appropriate for that API. NonDex currently supports two kinds of behaviors: (1) *permutation* for APIs where order is underdetermined, and (2) *extensions* for APIs where only lower bounds on array size(s) are specified. The runner takes as inputs (i) a random seed, which completely determines the choice of behaviors, (ii) the mode of exploration—ONE or FULL (the two modes differ in the kind of wrong assumptions they can detect, as described in detail in Section 2.2[2]), and (iii) optionally the range of choices to be randomized (which is used by debugging).

[2]We originally evaluated four different modes, but the publicly released NonDex offers only two modes, ONE and FULL, because they are the easiest to understand and correspond to the two extremes of nondeterminism.

2.4.3 Detector

The detector first runs all tests once without randomization and then calls the NonDex runner a number of times, with different random seeds, to rerun all the tests. The detector reports tests that *pass without* NonDex randomization but *fail with* NonDex randomization; such tests likely[3] make wrong assumptions on underdetermined APIs. The detector first runs the tests without NonDex because tests that fail on their own are due to some other causes and should not be reported as failures due to wrong assumptions. After the first run, the detector invokes the instrumentation engine to create the instrumented APIs (or reuses cached copies of previously instrumented APIs) before it starts running tests with NonDex.

The detector stores information about failing tests in a `.nondex` directory which also contains information about each execution, without and with NonDex, as well as the configuration used for test executions, the seed needed to reproduce the failure, and the number of invocations of the runner's choice generator; the latter number helps the debugging phase to search for the invocation(s) that caused the failure(s).

2.4.4 Debugger

When a test fails with NonDex, the test may invoke several underdetermined APIs, e.g., it may iterate over several `HashSet` objects. Many of these invocations are correct, making no wrong assumptions, so manually locating the invocation(s) that caused the detected failure can be tedious. NonDex's debugging phase automatically identifies such invocation(s).

To identify such invocation(s), NonDex uses a binary search that keeps track of a range of API invocations and selectively enables exploration for half of them. Even for disabled invocations, our search advances the random-number generator, i.e., NonDex still calls the random-number generator to shuffle the order of elements, but NonDex returns the original, not the shuffled, order. (Without this control, the search could get different behaviors for the same random seed, making it harder to reproduce the failure.) Debugging continues until a single invocation is identified or the remaining range cannot be further halved. If a single invocation cannot be identified from

[3]The tests may be unreliable [85] due to other reasons and fail irrespective of NonDex.

running just one test method, NonDex re-starts debugging for the entire test class, and if again a single invocation cannot be identified, NonDex re-starts debugging for the entire test suite. Debugging is repeated for each failing test reported by the detector.

The debugging phase reports to the user an API call that causes the detected failure together with the call stack of the API's invocation which further helps in localizing the context in which the wrong assumption was made. In our prior work [120], we performed all debugging manually; after implementing automated debugging, we found that we had made an error in manually identifying the root cause of one failure, which anecdotally shows that the automated debugging helps to more reliably identify the root causes.

2.5 Evaluation

We evaluated our NonDex technique on 195 open-source projects and 72 student submissions from a software-engineering course. Section 2.5.1 describes our experiments with the open-source projects and Section 2.5.2 describes our early efforts and results for NonDex adoption. Section 2.5.3 describes our experiments with the student code.

2.5.1 Experiments on Open-Source Projects

We evaluated NonDex on 195 open-source projects. We selected these projects and their specific revisions from our previous studies with open-source projects [24, 79, 121]. All these projects are from GitHub [36], use Maven to build [88], and compile successfully using Java 8. For each project, we first ran NonDex with 10 randomly generated seeds, using the FULL level. If any test failed with these 10 seeds, we examined it to determine what caused the failure.

We detected 60 unreliable tests in the 21 projects listed in Table 2.3. We tabulate a short PID for ease of reference, the project name, and the project revision on which we ran NonDex. For each project with an unreliable test (found with 10 random seeds), we then reran that project's tests again using NonDex with 100 randomly generated seeds, using all nondeterministic levels. We obtained the number of times each unreliable test fails out of the

Table 2.3: 21 projects (out of 195) with at least one unreliable test

PID	Project	SHA
P1	EsotericSoftware/reflectasm	455f612e
P2	EsotericSoftware/yamlbeans	2ccfbd9d
P3	JodaOrg/joda-time	07002501
P4	OryxProject/oryx	833c3fea
P5	Thomas-S-B/visualee	410a80f0
P6	apache/commons-cli	a0dcd6a0
P7	apache/commons-lang	fad946a1
P8	benas/easy-batch	4761ba5a
P9	bpsm/edn-java	c1d891d6
P10	caelum/vraptor	443cf0ed
P11	fernandezpablo85/scribe-java	0311a435
P12	geosolutions-it/geoserver-manager	a4268dda
P13	jknack/handlebars.java	83dd013a
P14	joel-costigliola/assertj-core	e8a696e8
P15	jscep/jscep	a224cc25
P16	junit-team/junit	1d63100e
P17	ning/org-json	9be37018
P18	qos-ch/slf4j	52fcbbe8
P19	sematext/ActionGenerator	10f4a3e6
P20	stickfigure/objectify	819eb72f
P21	versly/wsdoc	89480c5d

100 seeds.

Table 2.4 shows a partial list of the 60 tests that we examined. We tabulate the PID (from Table 2.3), the name of the test class and its unreliable test method, the number of failures detected for each of the four levels, and the underdetermined API that causes the failures. The `apache/commons-lang` project has 14 tests similar to `MultilineRecursiveToStringStyleTest::boolArray`, and the `caelum/vraptor` project has 13 tests similar to `XStreamSerializerTest::shouldSerializeCollection`, so the two table rows show the total number of failures for each level across all 14 and 13 tests, respectively. Our evaluation started on older revisions of these projects, and three tests (`GenericTest::testWrite`, `TestDateTimeZone::testGetShortName`, and `TagTypeTest::collectSectionAndVars`) are already fixed on the current re-

visions of their respective projects.

Running NonDex using the FULL level may introduce too much nondeterminism, and one might initially consider some detected unreliable tests to be false alarms. However, Table 2.4 shows that only six unreliable tests (`FieldAccessTest::testIndexSetAndGet`, `OptionGroupTest::testToString`, `FieldUtilsTest::testGetAllFields`, `FieldUtilsTest::testGetAllFieldsList`, `MethodSorterTest::testJvmMethodSorter`, and `EventLoggerTest::testEventLogger`) fail sometimes for the FULL level but not fail at all for any of the 100 randomly generated seeds for any of the other levels. The remaining `54` unreliable tests are also detected by the other levels, suggesting that these are not false alarms.

For each unreliable test, the table shows the number of seeds/runs on which it fails. For most unreliable tests, the number of seeds is fairly high, with only 8–14 unreliable tests failing for fewer than 50 seeds for each level (not counting unreliable tests that have 0 failures for a given level), and only two of those tests fail fewer than 30 times for each level. These high numbers suggest that it is likely that an unreliable test can be detected by running NonDex with just a few seeds.

Assume that the actual probability of an unreliable test failing for a seed is equal to the percentage of seeds that fail out of the 100 seeds that were run. For example, if the probability of an unreliable test failing for a seed is 30%, then the probability of the unreliable test *not* failing for 10 different, independent seeds is $(1 - 0.3)^{10} = 0.028$; in other words, there is a less than 3% chance of NonDex missing to detect that unreliable test running with 10 seeds. In the most extreme case we detected, in the `Thomas-S-B/visualee` project, the expected probability of an unreliable test failing for a seed in the FULL level is only 8%, so the chance of NonDex missing this unreliable test running with 10 seeds is $(1 - 0.08)^{10} = 0.434$. Even in this case, there is more than 50% chance of detecting such an unreliable test with 10 seeds, despite the chance of it failing for any one seed being rather low.

In summary, a developer using NonDex to detect unreliable tests may not need to run with many seeds and can still have some confidence that NonDex does not miss to detect any unreliable tests. Therefore, albeit NonDex runs 3 times by default to minimize the user wait time, for stronger guarantees we recommend that NonDex by default be run for 10 seeds while increasing the level of nondeterminism, from ONE to FULL.

Table 2.4: Unreliable tests detected in open-source projects

PID	TestClass::testName	FULL	ID	EQ	ONE	Cause
P1	FieldAccessTest::testIndexSetAndGet	48	0	0	0	Class::getDeclaredFields
P2	GenericTest::testWrite	73	75	54	53	HashMap::entrySet
P3	TestDateTimeZone::testGetShortName	35	53	53	53	DateFormatSymbols::getZoneStrings
P4	TextUtilsTest::testJSONMap	51	52	60	53	HashMap::entrySet
P5	JPAExaminerTest::testFindAndSetAttributesManyT...	8	5	5	6	Class::getDeclaredMethods
P5	JavaSourceTest::testGetDependenciesOfType	12	12	12	4	Class::getDeclaredMethods
P6	OptionGroupTest::testToString	42	0	0	0	HashMap::values
P6	BugCLI162Test::testPrintHelpLongLines	51	55	55	53	HashMap::values
P7	MultilineRecursiveToStringStyleTest::boolArray	100	100	100	100	Class::getDeclaredFields
P7	...other 14 similar tests, total failures...	1296	1216	1215	1138	Class::getDeclaredFields
P7	FieldUtilsTest::testGetAllFields	100	0	0	0	Class::getDeclaredFields
P7	FieldUtilsTest::testGetAllFieldsList	100	0	0	0	Class::getDeclaredFields
P7	FieldUtilsTest::testGetFieldsWithAnnotation	56	51	53	45	Class::getDeclaredFields
P8	GsonRecordMarshallerTest::marshal	86	77	77	84	Class::getDeclaredFields
P8	JacksonRecordMarshallerTest::marshal	87	81	81	84	Class::getDeclaredFields
P8	XstreamRecordMarshallerTest::marshal	96	94	94	97	Class::getDeclaredFields
P9	PrinterTest::testPrettyPrinting	69	73	54	53	HashMap::entrySet
P10	XStreamSerializerTest::shouldSerializeCollection	41	48	45	52	Class::getDeclaredFields
P10	...other 13 similar tests, total failures...	736	709	737	764	Class::getDeclaredFields
P11	MapUtilsTest::shouldPrettyPrintMap	97	94	97	97	HashMap::entrySet
P12	GSLayerEncoder21Test::testMetadata	84	81	71	100	HashMap::entrySet
P13	TagTypeTest::collectSectionAndVars	100	100	100	100	HashMap::keySet
P14	Maps_format_Test::should_format_Map_containing...	76	50	62	53	HashMap::entrySet
P15	DefaultCertStoreInspectorTest::example	92	94	59	53	HashMap::keySet
P15	HarmonyCertStoreInspectorTest::example	95	96	59	53	HashMap::keySet
P16	MethodSorterTest::testJvmMethodSorter	100	0	0	0	Class::getDeclaredMethods
P17	TestSuite::testJSONStringerObject	79	77	83	84	Class::getFields
P18	EventLoggerTest::testEventLogger	100	0	0	0	Class::getDeclaredMethods
P19	BulkJSONDataESSinkTest::testGetBulkData	49	37	47	43	HashMap::entrySet
P19	JSONUtilsTest::testGetElasticSearchAddDocument	35	35	43	47	HashMap::entrySet
P19	XMLUtilsTest::testGetSolrAddDocument	36	43	43	47	HashMap::entrySet
P20	CollectionTests::testBasicSets	100	96	91	84	HashMap::keySet
P20	CollectionTests::testCustomSet	85	79	91	84	HashMap::keySet
P21	JaxRSRestAnnotationProcessorTest::stabilitySet...	71	86	51	53	HashMap::keySet
P21	SpringMVCRestAnnotationProcessorTest::stabilit...	76	75	51	53	HashMap::keySet
Unreliable Tests Found		60	54	54	54	
Total Failures		4362	3744	3643	3590	
Min Failures		8	0	0	0	
Max Failures		100	100	100	100	

The common threats to validity apply to our study, therefore our results may not generalize to other projects or unreliable tests. Of particular concern is that our experiments could have missed some unreliable tests even in the projects that we ran with 10 seeds. If some test fails infrequently, it may be missed; there might be many such tests that NonDex missed, so we could not have even studied them in more detail. In the future, we plan to evaluate more systematic exploration to check whether this is indeed the case.

We next discuss in more detail three unreliable tests detected by NonDex in open-source projects.

Overly Nondeterministic Level

A case where the FULL level detects an unreliable test that is never detected for any other level is `OptionGroupTest::testToString` from the project `apache/commons-cli`. Figure 2.7 shows that unreliable test. Lines 4 and 5 add two *options* to the `OptionGroup g1`. `OptionGroup` stores options in a `HashMap` (line 16), and its `toString` method (lines 19–27) iterates over this map. The code encodes that the iteration order over the `HashMap` is not guaranteed, so lines 6 and 7 check that the result of calling `toString` on `g1` is either of the two hard-coded strings. However, `toString` is invoked twice, and in the FULL level, NonDex can reshuffle the order differently for the two invocations, causing the assertion to potentially fail. The developer made a reasonable assumption that calling `toString` on the same, unchanged object twice returns the same string both times; we see that the other levels of NonDex never flag this test as unreliable. Nevertheless, the test could be still changed to call `toString` only once, capture the result, and then assert that it is one of the two possible values.

Example New Unreliable Test

We detected 57 unreliable tests that were not fixed on the then-current revision of the projects, and Figure 2.8 shows one such unreliable test, `MapUtilsTest::shouldPrettyPrintMap` from the `fernandezpablo85/scribe-java` project. The test (lines 3–7) makes and populates a `HashMap` and then compares the result of calling `MapUtils::toString` with a hard-coded string (lines 8–10). However, `MapUtils::toString` calls `entrySet` on its input `Map`, and the

```
1  public class OptionGroupTest {
2    public void testToString() {
3      OptionGroup g1 = new OptionGroup();
4      g1.addOption(new Option(null, "foo", false, "Foo"));
5      g1.addOption(new Option(null, "bar", false, "Bar"));
6      if (!"[--bar Bar, --foo Foo]".equals(g1.toString())) {
7        assertEquals("[--foo Foo, --bar Bar]", g1.toString());
8      }
9      ...
10   }
11 }
12
13 public class OptionGroup ... {
14   Map<String, Option> om = new HashMap<String, Option>();
15   public OptionGroup addOption(Option option) {
16     om.put(option.getKey(), option);
17     return this;
18   }
19   public String toString() {
20     StringBuilder buff = new StringBuilder();
21     Iterator<Option> iter = getOptions().iterator();
22     buff.append("[");
23     while (iter.hasNext()) {
24       /* ... populate buff with the values in iter ... */
25       return buff.toString();
26     }
27   }
28 }
```

Figure 2.7: Example unreliable test from `apache/commons-cli`

order of iteration is not fixed, so the assertion on lines 8–10 can sometimes fail. More precisely, it fails in all but one of the 4! orderings, i.e., in about 96% of cases, as also obtained in our experiments.

2.5.2 Practical Impact and Adoption

Detecting Failures

To test the NonDex tool in general and the NonDex Maven plugin in particular, we integrated NonDex in the `pom.xml` files of several Maven-based

```
1  public class MapUtilsTest {
2    @Test public void shouldPrettyPrintMap() {
3      Map<Integer, String> map = new HashMap<>();
4      map.put(1, "one");
5      map.put(2, "two");
6      map.put(3, "three");
7      map.put(4, "four");
8      assertEquals(
9         "{ 1 -> one , 2 -> two , 3 -> three , 4 -> four }",
10         MapUtils.toString(map));
11   }
12 }
13
14 public class MapUtils {
15   public static <K,V> String toString(Map<K,V> map) {
16    ...
17     StringBuilder result = new StringBuilder();
18     for(Map.Entry<K,V> entry : map.entrySet()) {
19       result.append(String.format(", %s -> %s ",
20           entry.getKey().toString(),
21           entry.getValue().toString()));
22     }
23     return "{" + result.substring(1) + "}";
24   }
25 }
```

Figure 2.8: Example unreliable test from `fernandezpablo85/scribe-java`

projects from GitHub. Our goal was to test whether NonDex works with these projects "out-of-the-box" and not necessarily to detect any new unreliable tests. We found that integrating NonDex into these projects was indeed easy, and that by just adding a few lines to `pom.xml`, we could run NonDex on all these projects. NonDex worked well with projects that use different testing frameworks (e.g., JUnit 4, JUnit 3, and TestNG) and even various test runners (e.g., parameterized tests [114,127]). Along the way, we also detected 21 new unreliable tests in eight projects (eight tests in `alibaba/fastjason`, five tests in `checkstyle/checkstyle`, three tests in `nutzam/nutz`, and one test in each of `alibaba/druid`, `bukkit/bukkit`, `jankotek/mapdb`, `pedrovgs.algorithms/algorithms`, and `perwendel/spark`).

Debugging Failures

We further applied the automated NonDex debugging on these 21 newly detected and 54 previously detected failing tests to determine the root cause of each failure. The number of underdetermined API invocations that NonDex randomized per failure ranged from 5 to 9,710. The results showed that our simple binary-search debugging works extremely well for these cases—for 74 out of 75 failures, NonDex minimized the cause down to only one invocation; the remaining failure is for a test written in JUnit 3 for which the Surefire Maven plugin (used by NonDex to run tests) cannot easily run single test methods. We also counted the number of wrong assumptions on various APIs supported by NonDex; the invocations causing the failures were `getDeclaredFields` (41 cases), `HashMap` iteration (32 cases), and `getGenericExceptionTypes` (1 case). Because binary search is simple, we were surprised that it sufficed to identify precisely one invocation in all but one of the cases we tried. In the future, we plan to explore more sophisticated search strategies, such as delta debugging [145], and automated fixing.

Case Studies and Adoption

We opened 13 pull requests (PRs) for failures detected by NonDex, reporting the issue and providing a fix, in four open-source projects: five PRs in `alibaba/fastjson`, five PRs in `checkstyle/checkstyle`, two PRs in `scribejava/scribejava`, and one PR in `square/retrofit`. We did not open PRs for all unreliable tests that NonDex detected because we are not experts in the projects and could not easily provide a fix for each unreliable test. All PRs we opened were accepted by developers except one PR in `alibaba/fastjson`. One of the developers of Checkstyle was quite pleased with the PRs we opened, asked us about the tool we used to detect the issues, and recommended that we integrate NonDex in their continuous integration; we indeed integrated NonDex in both `pom.xml` and `.travis.yml` for Checkstyle [15]. Furthermore, we have piloted the use of NonDex in a software testing course to educate students about wrong assumptions on underdetermined APIs. Students have used NonDex to find issues both in their own code and in open-source projects they are familiar with. Overall, we found our currently released NonDex tool to be robust enough for use

```
1  public class DefaultNameProvider implements NameProvider {
2    public String getName(...) {
3      String[] nameSet = getNameSet(...);
4      return nameSet[0];
5    }
6    private synchronized String[] getNameSet(...) {
7      String[][] z = DateTimeUtils.getDateFormatSymbols(...).getZoneStrings();
8      String[] setEn = null;
9      ...
10     for (String[] s : z) {
11       if (s != null && s.length == 5 && id.equals(s[0])) {
12         setEn = s;
13         break;
14       }
15     }
16     ...
17   }
18 }
```

Figure 2.9: Code for `DefaultNameProvider` from `JodaOrg/joda-time`

both in real-world projects and in teaching.

Example Fixed Unreliable Test

We next describe an unreliable test that NonDex detected when run on an older revision of the `JodaOrg/joda-time` project; the test has been fixed since then. Figure 2.10 shows `TestDateTimeZone::testGetShortName` and the relevant portions of the SUT. The call to `getShortName` on line 4 eventually leads to a call to the `DefaultNameProvider::getNameSet` method defined on lines 6–17. The problem is the guard condition, `s.length == 5` on line 11. In Java 7, the call to `DateFormatSymbols::getZoneStrings` on line 7 indeed returned each array element of `z` of exactly length five. However, the specification of that method only guarantees that each element of `z` has length of *at least* five. In fact, in Java 8, the implementation changed such that each array element has length exactly seven, which still satisfies the specification but is different from what was the case in Java 7. This change in the implementation revealed the developer's reliance on the length of the elements of `z`. NonDex was able to detect this on an older revision of the code, and the

```
1  public class TestDateTimeZone extends TestCase {
2    public void testGetShortName() {
3      DateTimeZone zone = DateTimeZone.forID(...);
4      assertEquals("BST", zone.getShortName(...));
5      ...
6    }
7  }
8
9  public abstract class DateTimeZone ... {
10   public String getShortName(...) {
11     String name;
12     NameProvider np = getNameProvider();
13     if (np instanceof DefaultNameProvider) {
14       name = ((DefaultNameProvider) np).getShortName(...);
15     }
16     ...
17     return name;
18   }
19   private static NameProvider getDefaultNameProvider() {
20     NameProvider nameProvider = null;
21     ...
22     if (nameProvider == null) {
23       nameProvider = new DefaultNameProvider();
24     }
25     return nameProvider;
26   }
27 }
```

Figure 2.10: Example unreliable test from `JodaOrg/joda-time`

developers have since fixed this problem by changing checks such as the one shown on line 11 to be `s.length >= 5` instead.

Unrelated Unreliable Tests

We discuss one example unreliable test that we accidentally detected during our evaluation. Although detecting unreliable tests is a positive outcome in general, we are careful to mark this unreliable test as a false alarm (FA) in our evaluation because the source of flakiness is not related to any of the models that we are evaluating in NonDex.

Figure 2.11 shows the unreliable test `T1` that nondeterministically passes or

```
1  public class ClassLoaderTest extends TestCase {
2    public void testAutoUnloadClassloaders () throws Exception {
3      int ic = ACL.activeACLs();
4      ClassLoader tcLoader1 = new TestClassLoader1();
5      Class testClass1 = tcLoader1.loadClass(..);
6
7      ClassLoader tcLoader2 = new TestClassLoader2();
8      Class testClass2 = tcLoader2.loadClass(...);
9
10     tcLoader1 = null; testClass1 = null; ...
11     tcLoader2 = null; testClass2 = null; ...
12
13     // Force GC to reclaim unreachable
14     // (or only weak-reachable) objects
15     System.gc();
16     ...
17     System.gc();
18     while (ACL.activeACLs() > 1 && times < 50) {
19       Thread.sleep(100); // test again
20     }
21     // Yeah, both reclaimed!
22     assertEquals(Math.min(ic, 1), ACL.activeACLs());
23   }
24 }
```

Figure 2.11: Example unreliable test `T1` detected during our experiments

fails because it (incorrectly) assumes deterministic behavior of the garbage collector. Specifically, the calls to `System::gc` on lines 14 and 15 do not force garbage collection, per the official Java API documentation. Thus, the assertion on line 22 can sometimes fail when `ACL::activeACLs` does not return 1, as the developers of the tests assume.

2.5.3 Experiments on Student Code

We also evaluated NonDex on 72 student submissions for a programming assignment. We first describe the assignment that the students were supposed to do. We then describe how we set up our experiments for the student submissions. We finally describe high-level results concerning our findings of running NonDex on the student submissions.

Assignment

The assignment asked the students to create a simple library-management application. This library-management assignment was first created four years ago and has been minimally updated by different teaching staff members over the years; this year's iteration of the assignment was updated by two teaching assistants who were not involved in this study. The students were expected to write both code that implements such an application and unit tests using JUnit [68] to test the different components of the application.

The teaching staff provided the students some skeleton code outlining the basic expected components of the application. The application should represent a library containing books which can be organized into collections. The `Book` class represents a book and has only two fields, a title and an author, both represented by `String` objects. This `Book` class extends the abstract class `Element`. The `Collection` class represents a collection of such `Element` objects that are stored in a `List`. Furthermore, the `Collection` class also extends `Element`, so a `Collection` is allowed to contain other `Collection` objects, creating a hierarchy that illustrates the composite design pattern [33]. Finally, at the top level, there is a `Library` class that can hold a `List` of `Collection` objects.

Students were expected to implement several methods and constructors for each of these classes. We discuss those that are most relevant for this study. For both `Book` and `Collection`, students must implement a method `getStringRepresentation` that returns `String` representations of objects of those classes. Given such a string representations, students must implement a constructor for `Book` that takes the string representation and constructs the corresponding `Book` object. For `Collection`, students must similarly implement a static method `restoreCollection` that takes a string representation of a `Collection` and constructs the corresponding `Collection` object. For `Library`, students must implement (1) the constructor that takes a file containing string representations of a sequence of `Collection` objects and constructs the corresponding `Library` and (2) the method `saveLibraryToFile` that writes out the `Library` to a file.

Along with the skeleton code and implementation requirements, the teaching staff made further restrictions and suggestions. First, the students' code must build successfully in a common environment used by the entire class.

This environment uses OpenJDK Java 7, so students' code must also compile to Java 7 bytecode and run successfully using the OpenJDK Java 7 JVM. Students must also write tests for each of the three classes they implement, with at least nine tests for the entire application. Finally, the staff strongly encouraged the students to use some third-party library to handle the pretty-printing/parsing of objects to/from strings, as the `Library` can potentially have complex structures involving deeply nested `Collection` objects. However, the staff did not restrict the students to a specific third-party library, so the students chose whatever library they felt comfortable with. Many used various libraries for JSON or XML.

Experimental Setup

For our evaluation on student code, we started from the 89 submissions that built successfully (both compiled and had all tests pass) in the common environment that uses OpenJDK Java 7. With these 89 submissions, we ran the tests in another environment that is exactly the same as the environment provided to the students, except this other environment uses OpenJDK Java 8 instead. By running the students' tests against their own code an an environment using Java 8, we already detected some students' tests to be unreliable as they assumed specific behavior of the libraries (either the Java Standard Library or the third-party libraries used), and most likely failing due to the presence of ADIUS code.

Running the students' tests in this Java 8 environment, we found 17 submissions that fail. In fact, in the past, running in multiple environments (e.g., on Linux virtual machines and on Mac and Windows laptops from teaching assistants) was the only approach that we could use to detect (some) unreliable tests. Using NonDex, we can have a more thorough detection of unreliable tests; even if some tests pass on both Java 7 and Java 8, it does not imply they do not contain any ADIUS code that could fail on some future Java 9 (or even on Java 8 on another OS or by another JVM provider, say, IBM). We therefore focus the rest of our evaluation on the remaining 72 submissions.

Table 2.5: Unreliable tests detected in student submissions

	FULL	ID	EQ	ONE
Unreliable Tests Found	110	88	34	34
Total Failures	8159	6785	2031	1827
Min Failures	37	0	0	0
Max Failures	100	100	81	78

Results

We ran the student submissions using our NonDex tool in all four nondeterministic levels and for 100 randomly generated seeds. (The tests from students submissions run much faster than the open-source projects, so we could immediately use 100 seeds.) NonDex detected 34 student submissions with at least one unreliable test. In total, NonDex detected 110 unreliable tests. Table 2.5 summarizes the results. We tabulate the number of unreliable tests detected in each level (up to 110), and the total, minimum, and maximum number of the 100 seeds that cause a failure for one of those unreliable tests in each level. We elide detailed results for each individual test as in Table 2.4 because there are too many tests.

From the table, we see that the FULL level detects the most unreliable tests, followed by ID, and then by EQ and ONE, which both detect the same number of unreliable tests. Unlike for open-source projects where all three partial levels behaved the same (either all three had at least one failure or all three had no failure), for student submissions, ID detected more unreliable tests than either EQ or ONE that detected exactly the same unreliable tests. Considering the total number of failures, like for open-source projects, we see that the FULL level detects more failures than the ID level, followed by the EQ level and finally by the ONE level.

Discussion

In the 17 cases where the students' tests fail just by switching from Java 7 to Java 8, the unreliable tests check the functionality of the methods that get the string representation of a `Book` or a `Collection` object. The tests generally construct some `Book` or `Collection` objects and assert that the return of the method that gets the string representation matches some hard-coded

string value. In all but one of these submissions, students either directly use a Java `HashMap` as part of their implementation for constructing a string representation, or they use a third-party library (e.g., JSON in Java [66] or JSON.Simple [67]) where the serialization is backed by a Java `HashMap`. The assertions against the hard-coded strings succeed in Java 7 because the order remains consistent across different runs of the JVM, but in Java 8, the underlying implementation of `HashMap` changed such that the iteration order can differ from that of Java 7. The one remaining failing submission uses an XML serialization library (XStream [141]) to construct a string representation of a `Collection` object, but the order of the declared fields for a class is also not guaranteed, so the comparison with a hard-coded string value here once again fails in this later version of Java. In summary, all these 17 submissions have ADIUS code and fail due to relying on some assumed order that is not guaranteed to hold.

In the student submissions that do not fail on Java 8, NonDex detected additional unreliable tests that fail due to the nondeterminism in the ordering provided by the iterator for a `HashMap`. As with the tests that fail on Java 8, these unreliable tests generally construct `Book` and `Collection` objects and assert their string representation to be equal to a hard-coded string. Similar to some cases in the open-source projects, some failures are due to "too much" nondeterminism in the orderings, e.g., when a test calls `getStringRepresentation` on an object and then compares the string against another call of `getStringRepresentation` of an equal object rather than asserting the string to be the same as a hard-coded string. In such a case, the FULL or ID level would shuffle both calls to `getStringRepresentation` (because FULL shuffles always and ID shuffles when objects are different) and potentially end up failing the assertion where the other two levels do not fail. Moreover, the FULL level also detects as unreliable some cases that depend on the field ordering, which other levels never detect.

2.6 Systematic Exploration using Java PathFinder

Running NonDex even with 100 different seeds may not detect all unreliable tests, because the random choices explored can miss some cases that would cause a test to fail. To more systematically explore these tests, we used

JPF [64, 132]. JPF provides a specialized JVM, implemented in Java, that can explore all nondeterministic choices. However, JPF cannot handle all Java code out-of-the-box. In particular, it cannot handle code that depends on native methods, such as those in the `Gson` or `XStream` libraries that some students used. We only ran the student's code on JPF because JPF did not work with the open-source projects used in our study.

Our NonDex implementation for systematic exploration uses the Java PathFinder [64,132] tool and can *systematically* explore the choices by model checking the nondeterministic models. We focused on the `HashMap` iterator, because it can find a large number of ADIUS code and unreliable tests. We used the JPF's provided facility for nondeterministic choices, based on `Verify.getInt` which we use to return a permutation of the original iteration order, to encode a model that is between our partial and fully nondeterministic models. The advantage of using JPF is that it can provide guarantee on the completeness of the exploration, unlike random exploration. As a result, we had to extend JPF support for the Java Standard Library to attempt to run it on more code. But in the end we were able to run it only on a subset of student submissions.

2.6.1 Motivating Example

Figure 2.12 shows some test simplified from a student homework in the software engineering class used in our study. Recall that the students were asked to write code for a `Book` class and tests for their code. The method `testGetStringRepresentation1` aims to test that the `Book` object produces a correct string representation: the test checks that the round-trip from the string representation of a `Book` to a `Book` object and back to its string representation yields the same `String` result used to construct the `Book` object.

The problem with the test in Figure 2.12 is that it assumes the order of the fields in the JSON representation of the `Book` object to be the same every time, either {`author="Die...", title="Cos..."`} or {`title="Cos...", author="Die..."`}. This assumption is wrong; it is not supported by the JSON specification: the `author` and `title` could appear in any order in the resulting string. In fact, the data structures used to implement the underlying `JSONObject` do not guarantee the order assumed by this test. Specifically, a

```
1  public class BookTest {
2    private String toJSON(String s) throws JSONException {
3      JSONObject obj = new JSONObject();
4      String[] info = s.split(",");
5      obj.put("author", info[0].trim());
6      obj.put("title", info[1].trim());
7      return obj.toString();
8    }
9    @Test
10   public void testGetStringRepresentation1()
11       throws JSONException {
12     Book book = new Book(toJSON("Diego et al., Costization"));
13     assertEquals(toJSON("Diego et al., Costization"),
14                  book.getStringRepresentation());
15   }
16 }
```

Figure 2.12: Example test that fails due to an underdetermined specification

`HashMap` is used to store a mapping between field names and their values, and the code in `JSONObject` (not shown here) iterates the `HashMap` to produce the `String` representation. The specification of the `HashMap` explicitly states: *"This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time."* [53]. Code making such wrong assumptions, unsupported by the specification, is brittle because whenever the library changes, the assumptions may stop holding, and the code can break [58, 70].

Our NonDex technique finds assumptions on certain APIs by exploring different behaviors permitted by the specification. If exploring these different behaviors triggers a failure, it indicates that the code makes some wrong assumption on the API. In the example in Figure 2.12, NonDex would explore different orders of iteration for the underlying `HashMap` of each `JSONObject`. Note that it is necessary to explore an execution where the *two* iteration orders for the two `JSONObject` objects differ, i.e., {`author="Die..."`, `title="Cos..."`} and {`title="Cos..."`, `author="Die..."`}. We manually create models for APIs based on their specifications, and we use JPF to explore these models for all allowed behaviors to find wrong assumptions.

Figure 2.13 shows the entire state-space graph resulting from the JPF's ex-

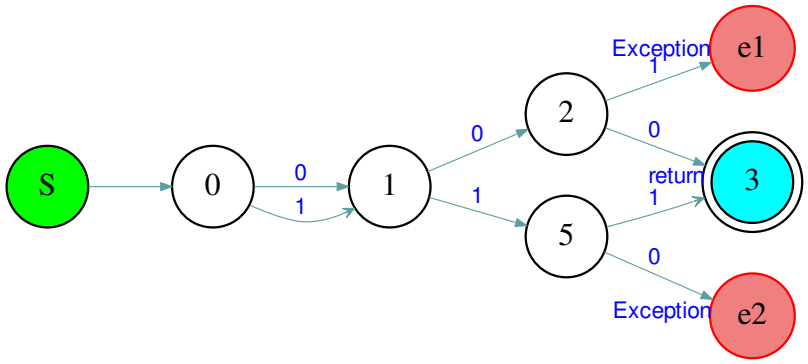

Figure 2.13: State-space graph for the example test

ploration of different behaviors of APIs with underdetermined specifications in this example. In the execution of the test `testGetStringRepresentation1`, the program executes three underdetermined APIs, corresponding to the three choice points. Two of these are in the translation of the `JSONObject` to `String` in the method `toJSON` (called twice from the test), and one is in the body of the method `getStringRepresentation` (not shown here). Each of these choice points is over a collection with two elements (corresponding to the fields `author` and `title`), hence it has two possible orders.

Even this simple graph illustrates some interesting properties. For example, the nondeterministic choice point in state 0 is rather local, and both of its orders lead to the same state 1. The reason is that the first call to `toJSON` can produce two different string objects, but both of them produce the same `Book` object. Effectively, this choice point does not matter for the failure. What does matter is the relationship between the second and third choice points: if they choose the same order, the test passes, but if they choose different orders, the test fails. The probability that a uniformly randomly selected execution finds this failure is exactly 50%. Moreover, a simple strategy that always switches between choosing the natural order (the first outgoing edge, marked 0) and its opposite (the last outgoing edge, in this case, marked 1) would definitely find the failure in this example, but this is not always the case. We discuss in Section 2.6.3 the results from more examples.

```
1  // in jpf-core/src/classes/java/lang/Class.java
2  ... class Class ... {
3    ...
4    public native Field[] getDeclaredFields() throw...;
5  }
6  // in jpf-core/src/peers.../JPF_java_lang_Class.java
7  ... class JPF_java_lang_Class extends NativePeer {
8    ...
9    @MJI
10   public int getDeclaredFields_____3Ljava_lang_reflect_Field_2
11       (MJIEnv env, int objRef) {
12     ...
13     for (i=0; i<nStatic; i++) {
14       FieldInfo fi = ci.getStaticField(i);
15       ...
16     }
17     for (i=0; i<nInstance; i++) {
18       FieldInfo fi = ci.getDeclaredInstanceField(i);
19       ...
20     }
21   }
22 }
```

Figure 2.14: Original `Class::getDeclaredFields` in JPF

2.6.2 Technique and Implementation

Recall that the overall NonDex technique is rather simple: we first manually find methods in the Java Standard Library with underdetermined specifications, then manually build models of these methods, and finally use an appropriate execution environment to explore various behaviors of these models. We next describe how we implemented NonDex models in JPF. We presented already one implementation of the `HashMap` iterator in Section 2.3. Thus, we illustrate here the implementation of another method, and also mention one change we made in the former implementation of the `HashMap` iterator. The key goal of our implementation of NonDex in JPF is to enable *systematic exploration* of all possible behaviors of methods with underdetermined specifications.

To illustrate our encoding of models in JPF, consider the `getDeclaredFields` method from the class `java.lang.Class`. This method returns an array of

```
1  // modified Class.java
2  ... class Class ... {
3    ...
4    public Field[] getDeclaredFields() throw... {
5      return NonDex.shuffle(getDeclaredFields0());
6    }
7    public native Field[] getDeclaredFields0() ...;
8  }
9  // modified JPF_java_lang_Class.java
10 ... class JPF_java_lang_Class extends NativePeer {
11   ...
12   @MJI
13   public int getDeclaredFields0_____3Ljava_lang_reflect_Field_2
14       (MJIEnv env, int objRef) {
15     /* body the same as was in getDeclaredFields */
16   }
17 }
```

Figure 2.15: Modified `Class::getDeclaredFields`

the type `Field[]` which represents all the fields declared by the class (but excludes inherited fields). The Javadoc for this method states: *"The elements in the returned array are not sorted and are not in any particular order."* [19].

A typical implementation of this method is deterministic and returns the fields in some particular order. For example, in JPF, this method is implemented as a native peer with the relevant parts shown in Figure 2.14. The `Class` implementation declares only that the method `getDeclaredFields` is native, and the actual implementation in `JPF_java_lang_Class.java` returns the array that has static fields before instance fields. Interestingly, the same `JPF_java_lang_Class.java` uses a different order in the method `getFields` which returns an array which represents all the public fields in the class and includes inherited fields—that method returns instance fields before static fields and has a comment *"the spec says there is no guaranteed order so we keep it simple"* [20].

To support NonDex, we modify `getDeclaredFields` such that JPF can explore all possible orders of the fields. We modified the implementation directly at the JPF level as shown in Figure 2.15: (1) we renamed the original `getDeclaredFields` peer to `getDeclaredFields0` and kept its body and (2) we added the method `getDeclaredFields` to first obtain the original array of

```
1  import gov.nasa.jpf.vm.Verify;
2  class NonDex {
3    public static <T> T[] shuffle(T[] objs) {
4      return shuffle(Arrays.asList(objs)).toArray(objs);
5    }
6    public static <T> List<T> shuffle(List<T> objs) {
7      int permutation = Verify.getInt(0, factorial(objs.size()) - 1);
8      return nthPermutation(permutation, objs);
9    }
10   public static <T> List<T> shuffleOld(List<T> objs){
11     int k = objs.size();
12     for (int i = 0; i < k - 1; i++) {
13       Collections.swap(objs, i, Verify.getInt(i, k-1));
14     }
15     return objs;
16   }
17   ...
18 }
```

Figure 2.16: NonDex methods for shuffling

fields and then shuffle it using our NonDex method `shuffle` (described in the next paragraph). Note that we effectively modified the behavior of an existing native method to add shuffling, which is easy to do in JPF because the native methods are themselves implemented in Java.

We next describe how we implemented the `NonDex::shuffle` methods. Figure 2.16 shows the key parts of our implementation. The `shuffle` method for arrays is the one invoked from `getDeclaredFields`, but many other methods require shuffling a list, so our key logic is in the `shuffle` method for lists. Its implementation is straightforward: given a list `objs`, it computes the total number of permutations of this list ($k!$, where k is the length of the list) and then selects one particular permutation to explore in each invocation, using the JPF library method `Verify::getInt`. (Note that both bounds in `getInt` are inclusive, hence subtracting one from the number of permutations.) The method `nthPermutation` computes the n-th permutation of a given list in the lexicographic order, using a traditional algorithm [117]. Note that several methods in the NonDex library modify their given arguments in place, but we ensure that they are called only when the arguments are copies that can be modified without affecting Java semantics. (While our goal is to explore

all *possible* orders using NonDex, we do *not* want to generate some *impossible* order.) For example, the Javadoc for several `Class` methods explicitly states: *"The caller of this method is free to modify the returned array; it will have no effect on the arrays returned to other callers."*

Figure 2.16 also shows an old shuffle method that we used in our first NonDex paper [120]. This method also enumerates all $k!$ permutations of the input `objs` list of length k, but it creates a different state-space graph that does not precisely capture the nondeterminism inherent in these permutations. This method uses the Knuth shuffle [74] for random permutations but applies it to systematically explore all possible permutations. For each position i, it chooses some position between i and $k-1$ to swap with i.

To illustrate the difference between the methods `shuffle` and `shuffleOld`, consider a list with 4 elements. The current shuffle creates a single choice point with 4! = 24 outgoing edges, i.e., the state-space graph has 25 nodes (1 choice point and 24 successor states). In contrast, the old shuffle would create one choice point with 4 outgoing edges of which each leads to a choice point with 3 outgoing edges of which each leads to a choice point with 2 outgoing edges, creating a factorial tree. This also gives 24 choices in the end, but the state-space graph now has 40 edges and 41 nodes, i.e., 16 more edges and 16 more nodes than our current nondeterministic choice tree. These additional edges and nodes do not properly capture the amount of nondeterminism but are just the consequence of how permutations are computed. For this reason, all our experiments use the current `shuffle` implementation, not only for the new methods that we added but also for the `HashMap` iterator.

2.6.3 Evaluation

We next present the results of our experiments on 46 student-written tests; we know from our previous work that (1) JPF can run these tests, at least for some executions, and (2) the tests contain wrong assumptions on APIs (Section 2.5.3). In the past, we ran these tests in JPF with only one underdetermined method and to find only one error state, thus we stopped the exploration on the first failure. In the current evaluation, our key goal is to analyze the state-space graphs, thus we run JPF with `search.multiple_errors=true`, and we also run with all 11 models of methods with underdetermined speci-

fications (`HashMap` iterator and 10 methods similar to `getDeclaredFields`).

Table 2.6 shows the statistics about the state-space graphs. We obtained the full graphs for 46 failing tests. We previously had five additional tests in Section 2.5.3. During the exploration of two tests, JPF ran out of memory (the default 1GB) after finding 450,463 and 1,321,584 errors, respectively. Two tests were affected by a real bug in JPF, namely the JPF native peers in `JPF_java_lang_StringBuilder.java` and `JPF_java_lang_StringBuffer.java` do not work with the latest Java versions. The fifth test was mistakenly reported as failing in the past, because the SUT throws some exceptions that are caught, printed, and "swallowed"; the code does have some bugs but not of the kind that NonDex should find.

State-Space Graph Size

We tabulate the graph size (number of nodes and edges) as a measure of the uses of underdetermined APIs. We find that many tests have rather simple graphs, similar to the example from Section 2.6.1. However, a few tests have large graphs, with the largest (T36) having 6,438,913 nodes and 12,747,262 edges. Note that all the code is single-threaded, so the choice points are due only to the methods with underdetermined specifications. The largest choice point that we allow to be exhaustively explored is for collections with six elements, i.e., 720 outgoing transitions. For larger collections, we explore only one order, as provided by the underlying implementation.

Failure Probability

We also show the number of failing nodes and the failure probability. The latter is computed under the assumption that each (local) choice for each choice point is equally likely, e.g., if a choice point has 6 outgoing edges, each has 1/6 probability to be chosen. The overall failure rate is computed over a reverse topological sort of the graph: each failing node has the failure probability of 1.0, each passing node has the failure probability of 0.0, and an inner node with n children has the failure probability $(p_1 + \ldots + p_n)/n$, where $p_1, \ldots, p_n$ are failure probabilities of the successor nodes. The failure probability of the start node in the graph gives the overall failure probability for the graph. We can see that it can be as high as 99.61%, and is at least

Table 2.6: Statistics of tests exploration

ID	#Nodes	#Edges	#Fail	P_f[%]	#Merges	#Crit
T1	7	7	2	50.00	0	2
T2	7	7	2	50.00	0	2
T3	208	283	64	75.00	29	32
T4	16	19	4	50.00	1	4
T5	7	7	2	50.00	0	2
T6	23	26	8	62.50	1	4
T7	5	4	1	50.00	0	1
T8	941099	950699	875520	98.96	386	9216
T9	53	71	36	72.22	4	6
T10	8	9	2	50.00	1	2
T11	8	9	2	50.00	1	2
T12	8130	8192	4032	98.44	0	64
T13	8	9	2	50.00	1	2
T14	35	42	12	75.00	5	4
T15	140	164	56	87.50	18	8
T16	150279	169994	65280	99.61	1797	256
T17	1124	1348	448	87.50	158	64
T18	10468	13252	3840	93.75	864	256
T19	8	8	2	50.00	0	2
T20	155	194	56	87.50	17	8
T21	224	332	56	87.50	22	8
T22	4	3	1	50.00	0	1
T23	6	5	2	75.00	0	1
T24	8825	9711	3968	96.88	700	128
T25	4	3	1	50.00	0	1
T26	885	1175	296	99.22	47	16
T27	17221	18311	8064	98.44	964	128
T28	7	7	2	50.00	0	2
T29	8	8	2	50.00	0	2
T30	8	8	2	50.00	0	2
T31	2645	3365	960	93.75	222	64
T32	9	8	3	87.50	0	1
T33	11	10	4	87.50	0	1
T34	15	15	6	75.00	0	2
T35	8	9	2	50.00	1	2
T36	6438913	12747262	65280	99.61	2113793	256
T37	5	4	1	50.00	0	1
T38	32	53	3	75.00	10	1
T39	24	37	3	75.00	6	1
T40	6	6	1	50.00	1	1
T41	5	4	1	50.00	0	1
T42	11	12	2	50.00	1	2
T43	1056	1106	552	95.83	28	24
T44	9	9	2	50.00	0	2
T45	20	23	6	87.50	3	2
T46	160	331	56	88.89	41	8

50% in all cases; it means that a uniformly random local selection of choices has a good chance to find any of these unreliable tests, which confirms why

our results with NonDex on JVM are already quite good (Section 2.5).

Irrelevant Nondeterminism

We further measure how much of the nondeterminism becomes irrelevant as the execution leads to the same state irrespective of the choices made at some choice point. Specifically, we count the number of "merge" nodes that have in-degree greater than one. (These are only the internal nodes and do not include the final, pass or fail, nodes.) While some tests have no merge nodes, other tests have quite a few, even up to almost one third of all nodes (T36 and T38). These merge nodes post-dominate some choice points that can be safely ignored when debugging the cause of failures due to underdetermined specifications in these cases.

Critical States

Collecting the entire state space enables us to determine the number of *critical* states, i.e., states with choice points from which at least one choice leads to paths that end either only in failure(s) or only in pass(es), while other choices lead to paths with different outcomes. In other words, these are the points where the exploration diverges, and so these are the key points for the developer to focus on when debugging failures that NonDex detects. We find that the number of critical states is relatively small compared to all states, the highest ratio being 32/208 for T3. Many cases have just one or two critical states. When JPF can analyze some code, our NonDex tool in JPF can greatly complement our NonDex tool in JVM: we envision a system where the tool in JVM is run first (because it can check all Java code and runs much faster for one execution) for some random choices, and if it detects a failure, then JPF is used to explore the neighborhood around this failure to determine which choice points are critical.

Choice Prioritization

Random exploration has a good chance to find the failure (e.g., with 50% failure probability for each path, trying just 7 independent paths gives over $1 - (1/2)^7 > 99\%$ probability to find the failure), but we evaluate whether

some prioritization heuristics could increase that chance. One seemingly good heuristic could be to first explore for each choice point the order that is *opposite (O)* of the *natural (N)* order, e.g., if some collection naturally returns `foo, bar, baz`, we could first explore `baz, bar, foo`. The intuition is that most tests pass for the natural order, and the opposite may create a completely unexpected situation. However, this heuristic finds failures in only 9 out of 46 tests. The reason is that many cases require *two* choices to be related for the failure (e.g., our running example requires two choices to differ). Additional heuristics are then to explore orders that alternate O and N, i.e., $ONONON...$ or $NONONO....$ All three heuristics together can find failures in 37 out of 46 cases, which is greater than 9 but still not perfect. In the future, we hope to identify heuristics that are even more likely to produce failures in most if not all cases.

Chapter 3

Detecting State-Polluting Tests to Prevent Test Dependency

In this chapter we describe our approach to detecting tests that pollute the state shared across test executions. Section 3.1 presents an overview of the problem of polluting tests, Section 3.2 presents our motivating example, Section 3.3 outlines PolDet, our general approach to detecting polluting tests, Section 3.4 presents some details on our implementation of PolDet, Section 3.5 presents the results of our evaluation, and Section 3.6 discusses the threats to the validity of our experiments.

3.1 Overview

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 run of the regression test-suite produces a failure, developers need to debug it. For a reliable test suite, failures should indicate a problem introduced by the code change and not a problem in the test suite itself. If the problem is indeed in the SUT, then it is highly beneficial that a test in the test suite failed. However, if the problem is in the test code itself, then the test code should be changed.

One common problem [7, 58, 85, 105, 137, 148] 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) [85], and that most dependencies are on the heap state, reported to range from 53% [85] to 61% [148] of all test dependencies. These dependencies make the outcome of regression test-suite runs unreliable: even for the same version

of the SUT, the tests could pass when executed in one order but fail when executed in another order leading to unreliable tests [85, 93, 148].

Several research groups have started developing techniques that can combat test dependencies. We discuss related work in Section 4 but highlight two techniques here. Zhang et al. [148] 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 [7] 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 nondeterministically 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 [70]. 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 [69, 86, 91]. Ideally a polluting test should be caught *right when the developer is about to add it to*

[1]While this dissertation 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 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 present 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 library 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, a prior study [85] 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 SUT).

This chapter makes the following contributions:

- ⋆ **Problem** We formalize the problem of test pollution
- ⋆ **Technique and Implementation** We present the PolDet technique that detects pollutions on shared heap or file system and tool for Java
- ⋆ **Evaluation** We evaluate PolDet 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 indeed relevant polluting tests. The runtime overhead of our PolDet prototype is a geometric mean of 4.50x, on a machine representative of a

powerful 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.

3.2 Motivating Example

We next discuss a real example of a polluting test that was added to the Apache Hadoop project [47] at one revision and then created problems in the test suite much later. Figure 3.1 shows a simplified code snippet from the `TestPathData` class. This snippet includes two tests of interest with their full names—`testAbsoluteGlob` and `testWithStringAndConfForBuggyPath`; for brevity, we will refer to them as FT and PT, respectively. The bug issue HADOOP-8695 [46] 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 [45], 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 technique (e.g., Zhang et al.'s technique [148] based on test reordering or Huo and Clause's technique [58] based on taint analysis) prior to our work 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; because 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 seemingly nondeterministic, which led to FT failing seemingly nondeterministically. This failure is reported in HADOOP-

[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.

```
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 3.1: `apache/hadoop` example of a polluting test that led to the failure of other tests later on

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 from when the polluting test is added, 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, when it is likely disrupting to their workflow to debug failing tests.

PolDet helps developers find polluting tests early. If PolDet 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 from revision 1186529 and lasted until revision 1374447.

3.3 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 description of the state 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 persis-

[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.

tent 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 dissertation, we focus on pollutions via heap state and file system; prior studies show these two to be the most prevalent causes of test dependency [85, 148].

We first discuss program 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.

3.3.1 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 3.4.2. Interestingly, in our experiments we find 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.

3.3.2 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 or holds a primitive value represented by node o_2. Each object has a field representing its class, hence some nodes represent classes themselves. 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* $\mathrm{V} \subseteq \mathcal{O} \cup \mathcal{C} \cup \mathcal{P}$, $\mathrm{E} \in 2^{(\mathcal{O} \cup \mathcal{C}) \times \mathcal{F} \times (\mathcal{O} \cup \mathcal{C} \cup \mathcal{P})}$, *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 test 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, e.g., a node representing a set may have multiple outgoing edges labeled `element`. In general, PolDet users can define abstractions specific to their program; by default, our implementation uses some generic abstractions from the XStream library [142] as described in Section 3.4.5.

3.3.3 Finding Heap-Shared State Differences

PolDet compares heap-graphs using graph isomorphism based on node bijection [139]. 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`, e.g., 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 [82] or CLR [72] 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 specify precisely 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 do not 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* $\rho(v) = v'$*.*

3.3.4 Class Loading

The use of common-roots isomorphism to compare heap states before and after a test run 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 class-specific 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 set of 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 higher 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.

3.3.5 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 fin-

ishes, 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 in a given portion of the file system, hashing their content, and checking the file/directory last-modified timestamps provided by the operating system. Before a test starts, PolDet iterates through each file, 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 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.

3.4 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.

3.4.1 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 used JUnit [68]. 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 3.2 shows an example test class with two tests and illustrates how JUnit invokes the constructor and methods of this class

```
1  class T {
2    @Before void setUp() { ... }
3    @Test void t1() { ... }
4    @Test void t2() { ... }
5    @After void tearDown() { ... }
6  }
7  // before constructor
8  T t = new T();
9  // before setup
10 t.setUp();
11 // after setup
12 t.t1(); // main test body
13 // before teardown
14 t.tearDown();
15 // after teardown
16
17 t = new T();
18 t.setUp();
19 t.t2();
20 t.tearDown();
```

Figure 3.2: `JUnit` workflow for running tests and illustration of capture points

for each test.

First, JUnit creates a new instance of the test class. Next, it invokes on the instance the setup method(s) annotated with `@Before`, if any. Then, it invokes the test method itself on the instance, running the test. Finally, it invokes the cleanup method(s) annotated with `@After`, if any. 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 only of all objects reachable from static fields.

3.4.2 Capture Points

PolDet extends the JUnit's test running mechanism to capture the state before and after each test executes. Figure 3.2 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` method runs, and the state *after* the test is run can be captured at the point before or after the `tearDown` method runs. 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.

3.4.3 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 [142], 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.

3.4.4 Comparing Heap-Shared States

After obtaining the serialized XML strings of the captured states, PolDet diffs them using XMLUnit [140], 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 definitely 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.

3.4.5 Abstracting Heap State for Java

As discussed in Section 3.3.2, 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 SUT.

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 SUT.) 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.

3.4.6 Eager Class Loading

We implemented 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.

3.4.7 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; other choices could have included the user's home directory or the parent of the current directory. 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, conceptually executing `touch f` to create a fresh file `f`. When the test finishes, PolDet conceptually 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 the timestamp `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.

3.5 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?

3.5.1 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 by open-source projects from GitHub. Because PolDet builds on top of JUnit, we integrated PolDet 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(s) 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 selected 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 3.1 shows some statistics about these projects.

PolDet has four main configuration options:
-capture_points determines where to capture the states to be compared. Figure 3.2 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 3.1 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 SUT only from the packages whose source belongs to the project source code (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 finding the source files in the project's source code and inferring the packages from the directory structure.

`-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 the Java Standard Library, and automatically generated classes that have `$$` in their name (but not the inner classes that have only one `$` 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).

3.5.2 Results

Table 3.2 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).

Project	LOC	Classes w/ Static Fields		Number of Static Fields	
		All	SUT	All	SUT
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

Table 3.1: Project statistics; the upper half had no pollution and the lower half had some pollution

	Test Methods					Test Classes							
Project	#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

Table 3.2: State pollution results; the columns are described in Section 3.5.2

3.5.3 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 each 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 access path is long or the polluted field is in some third-party library code, then the SUT likely cannot directly observe the value of the field, suggesting it to be a false positive. Indeed, we used the length of the path and the location of the field to prioritize our inspection of the reported polluting tests; we examine first the reports where the polluted field has a relatively short access path and is in the SUT. 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, we mark the true positive cell in Table 3.2 as `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 `$$`. We also want to avoid such classes.

Our default configuration for the PolDet tool is therefore 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 classes often have 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 Table 3.2 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 [13]. Bukkit implements a server for the popular Minecraft game. The root static field `PotionEffectType.byName` (declared in the SUT) has type `java.util.Map` and tracks the added potion effects (which are one of the game features to modify game entities).

Figure 3.3 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 to a list of potions. 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 `unreliableTest`, 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`.

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 3.4 shows an example from the Java APNS project [96]. 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

```
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 unreliableTest() { // we added this test
12     PotionEffectType.registerPotionEffectType(new
13       PotionEffectType(18) { ... }
14     });
15     assertNull(PotionEffectType.getByName(new
16       PotionEffectType(19) { ... }
17     });
18   }
19 }
```

Figure 3.3: The `bukkit/bukkit` true positive example with a test written to confirm the pollution

`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.

3.5.4 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 the test execution for each test, whereas a real tool would collect states at

```
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 3.4: The `notnoop/java-apns` false positive example

two or even fewer points.

The last column of Figure 3.2 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.

3.5.5 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 (which is higher than 4.50x because test suites are run twice, and bigger heap-graphs are created and compared). The new reports stem from common-roots isomorphism in lazy loading 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 test(s) 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 135) 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 nondeterministic and not controlled by the SUT. 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.

3.5.6 File-System Pollution

Table 3.2 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 when running only file pollution checks, without heap pollution checks. 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 temporary file, using Java's `File.createTempFile` method, which creates a 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 because 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 the number of inodes. Computing hashes of 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 few tests pollute the file system and just create fresh temporary files in `/tmp`.

3.6 Threats to Validity

There are several threats to the validity of our evaluation. As usual, 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 [85, 148], 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. Several collaborators on our study [44] 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 reordering the existing test suite [148] to find a failure due to test-order dependency may not work in many cases because 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.

Chapter 4

Related Work

In this chapter we discuss research related to unreliable tests in general and to the techniques this dissertation has presented in particular.

4.1 General Studies Related To Test Unreliability

Luo et al. [85] performed the first extensive study of unreliable tests; the study identifies common root causes of unreliable tests and common patterns that developers use to fix the unreliable tests. The study identifies wrong assumptions on the environment and test-order dependency as some root causes of unreliability, and it served as motivation and inspiration for this dissertation. NonDex and PolDet build on this study and presented techniques to help developers proactively identify some kinds of unreliable tests. Waterloo et al. [134] performed a broader study to identify bugs in tests; test unreliability is just one category of bugs in tests. Vahabzadeh et al. [131] also performed an empirical study of bugs in tests and identify unreliable tests as a class of bugs in tests and presented several causes of unreliability. This dissertation focuses mostly on designing techniques to detect unreliable tests and less on quantifying the extent of the problem in practice.

Regression testing techniques such as test selection [10, 25, 26, 38, 40, 51, 52, 111, 149], prioritization [16, 23, 50, 73, 87, 92, 123, 144], reduction [49, 57, 112, 147], and parallelization [8, 76, 102] are hindered by unreliable tests because any of the techniques can change the environment in which tests are run. Lam et al. [75] are the first to quantify the effect of test-order dependency on regression testing techniques. The findings motivate the research presented in this dissertation and show that unreliable tests are an important problem. Zhang et al. [148] empirically studied test-order dependency and proposed a technique to find dependent tests in 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 random 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.

Haidry and Miller [48] presented a set of prioritization techniques that take into account test-order dependencies and schedule tests to run in an order that preserves the dependencies. Bell et al. [8] presented ElectricTest, a technique to automatically detect dependencies between tests. ElectricTest instruments the JVM and finds data dependencies between tests (in contrast to manifest dependencies, data dependencies may or may not cause unreliable tests); the technique accounts for the dependencies when scheduling tests for parallel execution and ensures that the dependencies are preserved at run-time either by scheduling tests in the required order, or by simulating the data (if the data that the dependency is on is of a primitive type). Parsa et al. [102] use an Ant Colony System to optimize the scheduling of tests for parallelization in the presence of dependencies; their goal is to achieve a close to optimal schedule while preserving the dependencies. Similarly, Kappler [71] proposes an algorithm to work around test-order dependencies. This line of research demonstrates that dealing with unreliability after the fact is painful; our techniques are related in that we also aim to help developers deal with unreliable tests, but our approach is to help developers identify problems early and enable them to remove the problems, rather than make downstream techniques resilient to unreliable tests after they were introduced.

Shamshiri et al. [119] show that even automated test generation is impacted by the issue of unreliable tests. The empirical study shows that overall, over 15% of tests that are automatically generated by state-of-the-art tools such as Randoop [99], EvoSuite [32], and AgitarOne [1] are unreliable. Arcuri et al. [5] improve EvoSuite by enforcing that generated tests not make certain assumptions on the OS environment. The improvements isolate generated tests from the file system, console input, system state, and heap state.

Another related area of research is finding bugs due to environment. Parizek et al. [100] explore different environments to find faults in Java systems; the technique uses model checking to explore different environments. Gao et al. [34] performed a study to quantify how differently software behaves in

different environments. The study shows that when tests run in different environments, their coverage could vary wildly (up to 184 lines covered), showing that even evaluating the quality of tests in the presence of unreliable tests is challenging.

4.2 Techniques to Detect Unreliable Tests

Herzig and Nagappan [54] developed a technique based on association rules to identify when a test failure is a false alarm, i.e., the failure is not due to the SUT but rather to the test code or test infrastructure; the approach focuses developers' inspection and debugging effort by alerting the developer that a failure is likely to be a false alarm. This dissertation takes a different approach: rather than waiting for tests to fail and then decide if the failures are false alarms, we proactively detect unreliable tests. Our approaches are complementary, because it is likely uneconomical to proactively fix all unreliable tests (further, some causes may not be known a priori); our set of techniques could be used even after a false alarm is identified, to provide to developers more precise debugging information. Another more general technique that may help developers identify when a failure is unreliable was developed by Jiang et al. [61] who designed a technique that leverages information retrieval to identify what causes a test alarm; different from Jiang et al.'s technique, we help developers detect and debug unreliable tests.

4.2.1 Assumptions on APIs

Detecting problems due to wrong assumptions that developers make about specifications and implementations has been explored in many domains. For example, Jin et al. [62] reported how wrong assumptions about code can lead to performance bugs, in particular, they find the second most common reason for the introduction of performance bugs to be that "developers misunderstand the performance feature of certain functions". NonDex does not target performance bugs but helps detect another class of bugs that are due to specification misunderstanding. As another example, from a security perspective, Wang et al. [133] proposed a technique to analyze implicit assumptions that are necessary for the secure use of libraries. Their work involves

building models of methods which are then used to find bugs in software that fail to meet these implicit assumptions, finding serious security vulnerabilities in the process. Their techniques are mostly *static*, while NonDex uses a *dynamic* exploration of methods with nondeterministic specifications.

Randomness has been applied in different contexts to detect bugs, with many of these applications for concurrent code. For example, Eytani et al. [28] developed a tool that monitors shared variable accesses and applies random context switching when shared variables are accessed in order to trigger bugs in concurrent code. Parizek and Kalibera [100] used an abstract environment in software model checkers that randomly selects sequence of method calls in each thread to detect bugs in concurrent programs. Nistor et al. [95] randomly generated test sequences for concurrent programs in order to expose concurrency bugs. Joshi et al. [63] applied randomness in thread scheduling to create resource deadlocks in multi-threaded programs. Moreover, JPF can also control thread schedules to potentially explore all paths in the code [132]. In contrast, NonDex focuses on sequential code and exploration of underdetermined specifications.

Nondeterminism has been also studied for various other domains. For example, for map-reduce programs, Xiao et al. [138] studied nondeterminism that arises due to non-commutative reducers and found many bugs due to non-commutative reducers that make assumptions on the order of input data rows. For GUI code, Memon and Cohen [90] showed various factors that may cause nondeterminism and hence impact the results of analyses and experiments based on GUI software.

For state machines, testing conformance of deterministic implementations against nondeterministic specifications has a long history [55, 103, 104, 116]. More recently, Cook and Koskinen [22] aim to design a unified approach to reason about nondeterminism in real time systems; they apply their technique to examples drawn from real code. NonDex explores nondeterminism in the context of abstract data-type specifications using concrete exploration of real code.

API unit tests have previously been proposed for use in helping developers learn how to use the API, in addition to the documentation of the API [94]. Such an approach can be potentially made even more effective by adding "negative" unit tests that show *how not to use* the API, in the sense that they show what can happen when developers make deterministic

assumptions on nondeterministic API specifications. Applying our NonDex to code that currently uses an API can help to generate such tests, by first running NonDex on the existing tests and writing new API unit tests to capture the behavior of the code whenever deterministic assumptions do not hold.

Various research projects proactively detect software problems. For example, Shacham et al. [118] proposed a technique that finds atomicity violations that can lead to potential bugs after software changes; Lin et al. [81] proposed a technique for retrofitting parallelism into existing applications to prevent performance problems; and Yabandeh et al. [143] proposed 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.

Detecting differences between implementations and finding what is the impact of those differences is a well-established area of research. Change-impact analysis has been widely explored in static and dynamic program analysis context [2,3,14,41,77,78,80,98,108,113] to find the entities impacted by a change. Such techniques could be used to find when changes in libraries may affect any of the tests.

4.2.2 State Pollution

Researchers have developed techniques that compare states. For example, Cleve and Zeller [21] and Sumner and Zhang [125] used 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 and also helps in debugging by pinpointing the pollution.

Researchers have also proposed techniques to refactor shared state into private state. For example, Wloka et al. [135] proposed a program transformation for re-entrant programs to refactor shared state to thread-local state, and Wrigstad et al. [136] proposed 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.

Bell and Kaiser [7] presented 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. [93] 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 and not only avoid/tolerate test dependence, and proactively encourages developers to fix polluting tests.

Huo and Clause [58] 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.

Chapter 5

Future Work

There are many causes for unreliable tests that our techniques do not directly help with. In this section we discuss alternative approaches to tackling the problem of unreliable tests.

In NonDex we use a dynamic exploration approach to find when code makes wrong assumptions on APIs. Most of the underdetermined APIs are related to iteration of unordered collections; static analysis techniques that can show that the body of a loop iterating over an unordered data structure is an associative-commutative operation can identify safe and unsafe iterations over unordered collections and could help identify unreliable tests. Our JPF results in Section 2.6.3 show that the failure probability is high when there is a wrong assumption therefore just random sampling of some behaviors may expose wrong assumptions.

Our NonDex technique was most useful to identify unreliable tests, but wrong assumptions could also be bugs in the SUT; developers may make wrong assumption in their SUT which may cause code to break in production. Because most APIs are deterministic, many issues are hard to spot. An alternative and complementary solution is to make the underdetermined APIs exhibit different behaviors in different runs and provide developers a knob allowing them to natively turn on or off the nondeterminism in the execution environment. Python 3 [106] allows developers to control the hash seed which controls the iteration order for most of the unordered collections [107]. Unfortunately, the JVM does not offer this functionality; furthermore, more generally, because of the more pervasive overriding of `hashCode` in Java, it would be more challenging to introduce uniform and controllable nondeterminism in the JVM.

Many times, developers are unaware that a specification is underdetermined. There is a need for more tools and specification languages to precisely document underdetermined specifications in practice. In our research,

we did not benefit from having formally specified APIs, but rather we had to rely on imprecise documentation, written in English; while it is good that this documentation exists, having formal specifications would make it easier to check that code does not make wrong assumptions. Having formal specifications would enable a whole set of static techniques that could alert developers when they are likely to misuse an API. An area of future work is to automatically mine or infer specifications of underdetermined APIs; a technique may leverage multiple implementations of the same API to infer an underdetermined specification that is satisfied by either implementation—for Java Standard Library APIs there is the advantage that there are multiple implementation of each API that are available, but there is also the challenge that sometimes the implementations of these underdetermined APIs is native and written in C. A technique to effectively mine underdetermined specifications needs to consider these factors.

Our NonDex technique could be generalized to document underdetermined APIs. Library developers may annotate underdetermined APIs in their own library with annotations indicating the kind of underdetermineness the API has. We plan to generalize NonDex to support these annotations and add exploration capabilities to the annotated APIs. This capability enables users of the libraries to test that they are not misusing the APIs. NonDex easily supports APIs that return collections that need to be permuted. We foresee that other kind of return values may need different kind of randomization.

NonDex can be enhanced with better debugging support. Our implementation assumes a single wrong assumption is enough to expose a test failure. This assumption indeed held in our experiments, but we foresee that it does not always hold. We could explore techniques such as delta debugging [146] which do not assume a single assumption to be enough to expose the wrong assumption.

Our PolDet technique detects when tests pollute the state and informs the developer, but we foresee that it is possible to synthesize `teardown` methods that clean-up the environment for a test. PolDet can naturally help with this goal as it can provide to a synthesis algorithm the input-output pairs, i.e., the clean and polluted state. The challenge is to synthesize meaningful and readable code for recreating the clean state—in theory one could clean-up the state using reflection but that would yield unmaintainable code.

Cleaning up the state after every test may break subsequent tests that depend on the polluted state; to aid with this issue, `setup` methods could be synthesized that set-up the required state for a test.

PolDet is unaware (by design) of whether a test exists that depends on the polluted state because PolDet aims to encourage developers to follow good engineering practices as they write their tests. Information flow analysis techniques could assist developers in determining which tests if any depend on the polluted state.

PolDet sometimes reports pollution when the polluted state is likely inaccessible because the access path is long or the state is stored in private fields and it is hard to reach to. PolDet could be improved to add heuristics that filter out the false positives based on the length of the access path and the accessibility of the polluted data.

PolDet works well for heap pollution and even file system pollution, but it runs into limitations if other kinds of local environments get polluted, e.g., system environment variables. Another challenge is represented by remote environments, e.g., remote storage services or databases. Sometimes for remote systems it may be easier to simply reset the remote system than to check whether there was any pollution, but while this is easier, it can be also very expensive, so it may prove beneficial to detect and clean-up only the polluted state.

We conjecture that one of the most beneficial and impactful ways of reducing the pervasiveness of unreliable tests may be through education. Raising awareness about unreliable tests and exposing students to tools that help with detecting unreliable tests may make tomorrow's developers less likely to introduce unreliable tests in their test suites. We did incorporate discussions about unreliable tests and exposed our students to NonDex throughout our teaching. More research is required to establish what are the best ways to educate our students about unreliable tests.

Chapter 6

Conclusions

Unreliable tests are an important problem in practice, teaching, and research because unreliable tests slow down the testing and development process. We live in a world where fast software evolution is the norm, and unreliable tests slow down development and the necessary quality assurance process by failing without exposing a bug in the SUT.

This dissertation argues for a proactive approach to detecting unreliable tests by detecting unreliability causes as soon as they are introduced. The alternative—postponing the fixing of unreliability causes—increases the costs later when unreliable tests can impact many developers and slow down the development process unnecessarily. Fixing the unreliability as soon as it is introduced even before it can manifest and negatively impact developers is desirable. Fixing the unreliability as soon as it is introduced is also the easiest for the developer that introduced the unreliability.

Underdetermined specifications are good because they allow implementers to provide various implementations. However, wrong assumptions on underdetermined specifications are bad because they can result in seemingly random failures. In particular, ADIUS code that assumes a deterministic implementation of nondeterministic specification is susceptible to failures that arise from changing implementations. Tests that depend on ADIUS code can become unreliable tests that seemingly nondeterministically pass or fail. We presented a novel NonDex technique to detect unreliable tests due to ADIUS code. NonDex detected many unreliable tests in both larger, open-source projects and small-sized student code submissions.

When a test fails without exposing a bug in the SUT, 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 states 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.

To conclude, this dissertation introduces two techniques that assist developers to proactively identify causes of unreliable tests. Bugs in tests are important because they not only slow down developers but also hinder the developers' trust in their regression testing process because of false alarms in test failures. This dissertation argues that problems with unreliable tests should be addressed in a proactive fashion (this is not a controversial idea—developers do it with most other kinds of bugs because it appears cost-beneficial). There are many future directions in the area of unreliable tests that look interesting and useful. First, detecting more causes of unreliable tests would help developers identify other unreliable tests, e.g., due to network flakiness. Second automated fixing, while challenging, would ease the developers' work by assisting in removing unreliability from their test suites. Finally, putting more effort in educating today's students and tomorrow's developers about unreliable tests is a very effective way to act in a proactive fashion to prevent the introduction of unreliable tests in test suites.

References

[1] AgitarOne. http://www.agitar.com/solutions/products/automated_junit_generation.html.

[2] Apiwattanapong, T., Orso, A., and Harrold, M. J. A differencing algorithm for object-oriented programs. In *ASE* (2004), pp. 2–13.

[3] Apiwattanapong, T., Orso, A., and Harrold, M. J. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE* (2005), pp. 432–441.

[4] Appveyor. https://www.appveyor.com/.

[5] Arcuri, A., Fraser, G., and Galeotti, J. P. Automated unit test generation for classes with environment dependencies. In *ASE* (2014), pp. 79–90.

[6] AWS CodeBuild. https://aws.amazon.com/codebuild/.

[7] Bell, J., and Kaiser, G. Unit test virtualization with VMVM. In *ICSE* (2014), pp. 550–561.

[8] Bell, J., Kaiser, G., Melski, E., and Dattatreya, M. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE* (2015), pp. 770–781.

[9] Bocchino, Jr., R. L., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. A type and effect system for Deterministic Parallel Java. In *ASPLOS* (2009), pp. 97–116.

[10] Briand, L., Labiche, Y., and He, S. Automating regression test selection based on UML designs. In *IST* (2009), pp. 16–30.

[11] Bruneton, E., Lenglet, R., and Coupaye, T. ASM: A code manipulation tool to implement adaptable systems. In *AECS* (2002).

[12] Buck. http://facebook.github.io/buck/.

[13] Bukkit. http://bukkit.org/.

[14] Cai, H., and Santelices, R. A comprehensive study of the predictive accuracy of dynamic change-impact analysis. In *JSS* (2015), pp. 248–265.

[15] Checkstyle pull request integrating NonDex. https://github.com/checkstyle/checkstyle/pull/3393.

[16] Chen, T. Y., and Lau, M. F. A simulation study on some heuristics for test suite reduction. In *IST* (1998), pp. 777–787.

[17] Chen, Y., Su, T., Sun, C., Su, Z., and Zhao, J. Coverage-directed differential testing of JVM implementations. In *PLDI* (2016), pp. 85–99.

[18] CircleCI. https://circleci.com/.

[19] Class::getDeclaredFields Javadoc. https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html#getDeclaredFields--.

[20] Class::getFields in Java PathFinder. http://babelfish.arc.nasa.gov/hg/jpf/jpf-core/file/0069194b1048/src/peers/gov/nasa/jpf/vm/JPF_java_lang_Class.java#l580.

[21] Cleve, H., and Zeller, A. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM, pp. 342–351.

[22] Cook, B., and Koskinen, E. Reasoning about nondeterminism in programs. In *PLDI* (2013), pp. 219–230.

[23] Elbaum, S., Malishevsky, A., and Rothermel, G. Prioritizing test cases for regression testing. In *ISSTA* (2000), pp. 102–112.

[24] Eloussi, L. Detecting flaky tests from test failures. Master's thesis, UIUC, 2015.

[25] Engström, E., Runeson, P., and Skoglund, M. A systematic review on regression test selection techniques. In *I&ST-J* (2010), pp. 14–30.

[26] Engström, E., Skoglund, M., and Runeson, P. Empirical evaluations of regression test selection techniques: a systematic review. In *ESEM* (2008), pp. 22–31.

[27] Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., and Kandula, S. CloudBuild: Microsoft's distributed and caching build service. In *ICSE* (2016), pp. 11–20.

[28] Eytani, Y., Farchi, E., and Ben-Asher, Y. Heuristics for finding concurrent bugs. In *IPDPS* (2003), pp. 8–15.

[29] File - list JavaDoc. `http://docs.oracle.com/javase/8/docs/api/java/io/File.html#list--`.

[30] Flakiness dashboard HOWTO. `http://www.chromium.org/developers/testing/flakiness-dashboard`.

[31] Fowler, M. Eradicating non-determinism in tests. `http://martinfowler.com/articles/nonDeterminism.html`.

[32] Fraser, G., and Arcuri, A. Evosuite: Automatic test suite generation for object-oriented software. In *ESEC/FSE* (2011), pp. 416–419.

[33] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[34] Gao, Z., Liang, Y., Cohen, M. B., Memon, A. M., and Wang, Z. Making system user interactive tests repeatable: When and what should we control? In *ICSE* (2015), pp. 55–65.

[35] Fix internal data ordering. `https://github.com/geosolutions-it/geoserver-manager/commit/5447c06`.

[36] GitHub. `https://github.com/`.

[37] Must Use LinkedHashMap and LinkedList... `https://github.com/EsotericSoftware/yamlbeans/commit/1517822`.

[38] Gligoric, M., Eloussi, L., and Marinov, D. Practical regression test selection with dynamic file dependencies. In *ISSTA* (2015), pp. 211–222.

[39] Google build in the cloud. `http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html`.

[40] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. In *IEEE TSE* (2001), pp. 184–208.

[41] Gyori, A., Lahiri, S. K., and Partush, N. Refining interprocedural change-impact analysis using equivalence relations. In *ISSTA* (2017), pp. 318–328.

[42] Gyori, A., Lambeth, B., Khurshid, S., and Marinov, D. Exploring underdetermined specifications using java pathfinder. In *JPF Workshop* (2016), pp. 1—5.

[43] Gyori, A., Lambeth, B., Shi, A., Legunsen, O., and Marinov, D. NonDex: A tool for detecting and debugging wrong assumptions on Java API specifications. In *FSE Demo 2016*, pp. 993–997.

[44] Gyori, A., Shi, A., Hariri, F., and Marinov, D. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA* (2015), pp. 223–233.

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

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

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

[48] Haidry, S.-e.-Z., and Miller, T. Using dependency structures for prioritization of functional test suites. In *IEEE TSE* (2013), vol. 39, pp. 258–275.

[49] Hao, D., Zhang, L., Wu, X., Mei, H., and Rothermel, G. On-demand test suite reduction. In *ICSE* (2012), pp. 738–748.

[50] Hao, D., Zhao, X., and Zhang, L. Adaptive test-case prioritization guided by output inspection. In *COMPSAC* (2013), pp. 169–179.

[51] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. Regression test selection for Java software. In *OOPSLA* (2001), pp. 312–326.

[52] Harrold, M. J., Rosenblum, D. S., Rothermel, G., and Weyuker, E. J. Empirical studies of a prediction model for regression test selection. In *IEEE TSE* (2001), pp. 248–263.

[53] HashSet JavaDoc. `https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html`.

[54] Herzig, K., and Nagappan, N. Empirically detecting false test alarms using association rules. In *ICSE SEIP* (2015), pp. 39–48.

[55] Hierons, R., and Harman, M. Testing conformance of a deterministic implementation against a non-deterministic stream X-machine. In *TCS* (2004), pp. 191–233.

[56] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. Usage, costs, and benefits of continuous integration in open-source projects. In *ASE* (2016), pp. 426–437.

[57] Hsu, H.-Y., and Orso, A. Mints: A general framework and tool for supporting test-suite minimization. In *ICSE* (2009), pp. 419–429.

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

[59] Improvements to HashMap/LinkedHashMap use of bins/buckets and trees (red/black), Sept. 2013. `http://hg.openjdk.java.net/jdk8/jdk8/jdk/rev/d62c911aebbb`.

[60] Jenkins. `https://jenkins-ci.org/`.

[61] Jiang, H., Li, X., Yang, Z., and Xuan, J. What causes my test alarm?: Automatic cause analysis for test alarms in system and integration testing. In *ICSE* (2017), pp. 712–723.

[62] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In *PLDI* (2012), pp. 77–88.

[63] Joshi, P., Park, C.-S., Sen, K., and Naik, M. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI* (2009), pp. 110–120.

[64] JPF home page. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[65] JSON. `http://www.json.org/`.

[66] JSON in Java. `http://www.json.org/java/`.

[67] JSON-Simple. `https://code.google.com/p/json-simple/`.

[68] JUnit. `http://junit.org/`.

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

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

[71] Kappler, S. Finding and breaking test dependencies to speed up test execution. In *FSE* (2016), pp. 1136–1138.

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

[73] Kim, J.-M., and Porter, A. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE* (2002), pp. 119–129.

[74] Knuth, D. Seminumerical algorithms, the art of computer programming, vol. 2 1997.

[75] Lam, W., Zhang, S., and Ernst, M. D. When tests collide: Evaluating and coping with the impact of test dependence. Tech. rep., University of Washington Department of Computer Science and Engineering, 2015.

[76] Larsson, E., Arvidsson, K., Fujiwara, H., and Peng, Z. Integrated test scheduling, test parallelization and tam design. In *ATS* (2002), pp. 397–404.

[77] Law, J., and Rothermel, G. Whole program path-based dynamic impact analysis. In *ICSE* (2003), pp. 308–318.

[78] Le, W., and Pattison, S. D. Patch verification via multiversion interprocedural control flow graphs. In *ICSE* (2014), pp. 1047–1058.

[79] Legunsen, O., Marinov, D., and Rosu, G. Evolution-aware monitoring-oriented programming. In *ICSE NIER* (2015), pp. 615–618.

[80] Lehnert, S. A review of software change impact analysis.

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

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

[83] LinkedHashSet JavaDoc. `http://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html`.

[84] Liskov, B., and Guttag, J. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

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

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

[87] Malhotra, R., and Tiwari, D. Development of a framework for test case prioritization using genetic algorithm. In *ACM SEN* (2013), pp. 1–6.

[88] Maven Surefire plugin. `http://maven.apache.org/surefire/index.html`.

[89] Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., and Micco, J. Taming Google-scale continuous testing. In *ICSE* (2017), pp. 233–242.

[90] Memon, A. M., and Cohen, M. B. Automated testing of GUI applications: models, tools, and controlling flakiness. In *ICSE* (2013), pp. 1479–1480.

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

[92] Mirarab, S., and Tahvildari, L. A prioritization approach for software test cases based on Bayesian networks. In *FASE* (2007), pp. 276–290.

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

[94] Nasehi, S., and Maurer, F. Unit tests as API usage examples. In *ICSM* (2010), pp. 1–10.

[95] Nistor, A., Luo, Q., Pradel, M., Gross, T. R., and Marinov, D. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *ICSE* (2012), pp. 727–737.

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

[97] Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: Efficient deterministic multithreading in software. In *ASPLOS* (2009), pp. 97–108.

[98] Orso, A., Apiwattanapong, T., and Harrold, M. J. Leveraging field data for impact analysis and regression testing. In *FSE* (2003), pp. 128–137.

[99] Pacheco, C., Lahiri, S., Ernst, M., and Ball, T. Feedback-directed random test generation. In *ICSE* (2007), pp. 75–84.

[100] Parizek, P., and Kalibera, T. Efficient detection of errors in Java components using random environment and restarts. In *TACAS* (2010), pp. 451–465.

[101] Parnin, C., Helms, E., Atlee, C., Boughton, H., Ghattas, M., Glover, A., Holman, J., Micco, J., Murphy, B., Savor, T., et al. The top 10 adages in continuous deployment. In *IEEE Software* (2017), pp. 86–95.

[102] Parsa, M., Ashraf, A., Truscan, D., and Porres, I. On optimization of test parallelization with constraints. In *Software Engineering Workshops* (2016), pp. 164–171.

[103] Petrenko, A., Yevtushenko, N., and Bochmann, G. V. Testing deterministic implementations from nondeterministic FSM specifications. In *IWTCS* (1996), pp. 125–141.

[104] Petrenko, A., Yevtushenko, N., Lebedev, A., and Das, A. Nondeterministic state machines in protocol conformance testing. In *IWPTS* (1994), pp. 363–378.

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

[106] Python. `https://www.python.org/`.

[107] Python Hash Seed. `https://docs.python.org/3/using/cmdline.html#envvar-PYTHONHASHSEED`.

[108] Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices* (2004), vol. 39, pp. 432–448.

[109] Rossi, C., Shibley, E., Su, S., Beck, K., Savor, T., and Stumm, M. Continuous deployment of mobile software at Facebook (showcase). In *FSE* (2016), pp. 12–23.

[110] Rothermel, G., and Harrold, M. J. Analyzing regression test selection techniques. In *IEEE TSE* (1996), pp. 529–551.

[111] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. In *TOSEM* (1997), pp. 173–210.

[112] Rothermel, G., Harrold, M. J., von Ronne, J., and Hong, C. Empirical studies of test-suite reduction. In *STVR* (2002), pp. 219–249.

[113] Ryder, B. G., and Tip, F. Change impact analysis for object-oriented programs. In *PASTE* (2001), pp. 46–53.

[114] Saff, D., Boshernitsan, M., and Ernst, M. D. Theories in practice: Easy-to-write specifications that catch bugs, 2008.

[115] Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. Continuous deployment at Facebook and OANDA. In *ICSE* (2016), pp. 21–30.

[116] Savor, T., and Seviora, R. E. Supervisors for testing non-deterministically specified systems. In *ITC* (1997), pp. 948–953.

[117] Sedgewick, R. Permutation generation methods. *ACM CSUR* (1977), 137–164.

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

[119] Shamshiri, S., Just, R., Rojas, J. M., Fraser, G., McMinn, P., and Arcuri, A. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *ASE* (2015), pp. 201–211.

[120] Shi, A., Gyori, A., Legunsen, O., and Marinov, D. Detecting assumptions on deterministic implementations of non-deterministic specifications. In *ICST* (2016), pp. 80–90.

[121] Shi, A., Yung, T., Gyori, A., and Marinov, D. Comparing and combining test-suite reduction and regression test selection. In *FSE* (2015), pp. 237–247.

[122] Spring Repeat Annotation. `http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/test/annotation/Repeat.html`.

[123] Srivastava, A., and Thiagarajan, J. Effectively prioritizing tests in development environment. In *ISSTA* (2002), pp. 97–106.

[124] Sudarshan, P. No more flaky tests on the Go team. `http://www.thoughtworks.com/insights/blog/no-more-flaky-tests-go-team`.

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

[126] Testing at the speed and scale of Google, Jun 2011. `http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html`.

[127] Tillmann, N., and Schulte, W. Parameterized unit tests. In *FSE* (2005), pp. 253–262.

[128] Tools for continuous integration at Google scale, October 2011. `http://www.youtube.com/watch?v=b52aXZ2yi08`.

[129] TotT: Avoiding flakey tests. `http://googletesting.blogspot.com/2008/04/tott-avoiding-flakey-tests.html`.

[130] Travis CI. `https://travis-ci.com/`.

[131] Vahabzadeh, A., Fard, A. M., and Mesbah, A. An empirical study of bugs in test code. In *ICSME* (2015), pp. 101–110.

[132] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. *ASE Journal* (2003), 203–232.

[133] Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., and Gurevich, Y. Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In *USENIX* (2013), pp. 399–414.

[134] Waterloo, M., Person, S., and Elbaum, S. Test analysis: Searching for faults in tests. In *ASE* (2015), pp. 149–154.

[135] Wloka, J., Sridharan, M., and Tip, F. Refactoring for reentrancy. In *ESEC/FSE* (2009), pp. 173–182.

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

[137] Wuttke, J., Muşlu, K., Zhang, S., and Notkin, D. Test dependence: Theory and manifestation. Tech. rep., University of Washington, 2013.

[138] Xiao, T., Zhang, J., Zhou, H., Guo, Z., McDirmid, S., Lin, W., Chen, W., and Zhou, L. Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs. In *ICSE SEIP* (2014), pp. 44–53.

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

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

[141] XStream. `http://x-stream.github.io/`.

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

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

[144] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: A survey. *STVR* (2012), 67–120.

[145] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *TSE* (2002), 183–200.

[146] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.

[147] Zhang, L., Marinov, D., Zhang, L., and Khurshid, S. An empirical study of JUnit test-suite reduction. In *ISSRE* (2011), pp. 170–179.

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

[149] Zheng, J., Robinson, B., Williams, L., and Smiley, K. Applying regression test selection for COTS-based applications. In *ICSE* (2006), pp. 512–522.