

Making Software More Reliable by Uncovering Hidden Dependencies

Jonathan Bell

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2016

©2016

Jonathan Bell

All Rights Reserved

ABSTRACT

Making Software More Reliable by Uncovering Hidden Dependencies

Jonathan Bell

As software grows in size and complexity, it also becomes more interdependent. Multiple internal components often share state and data. Whether these dependencies are intentional or not, we have found that their mismanagement often poses several challenges to testing. This thesis seeks to make it easier to create reliable software by making testing more efficient and more effective through explicit knowledge of these hidden dependencies.

The first problem that this thesis addresses, reducing testing time, directly impacts the day-to-day work of every software developer. The frequency with which code can be built (compiled, tested, and package) directly impacts the productivity of developers: longer build times mean a longer wait before determining if a change to the application being build was successful. We have discovered that in the case of some languages, such as Java, the vast majority of build time is spent running tests. Therefore, it's incredibly important to focus on approaches to accelerating testing, while simultaneously making sure that we do not inadvertently cause tests to erratically fail (i.e. become *flaky*).

Typical techniques for accelerating tests (like running only a subset of them, or running them in parallel) often can't be applied soundly, since there may be hidden dependencies between tests. While we might think that each test should be independent (i.e. that a test's outcome isn't influenced by the execution of another test), we and others have found many examples in real software projects where tests truly have these dependencies: some tests require others to run first, or else their outcome will change. Previous work has shown that these dependencies are often complicated, unintentional, and hidden from developers. We have built several systems, VMVM and ELECTRICTEST, that detect different sorts of dependencies between tests and use that information to soundly reduce testing time by several orders of magnitude.

In our first approach, *Unit Test Virtualization*, we reduce the overhead of isolating each unit test with a lightweight, virtualization-like container, preventing these dependencies from manifesting. Our realization of Unit Test Virtualization for Java, VMVM eliminates the need to run each test in its own process, reducing test suite execution time by an average of 62% in our evaluation (compared to execution time when running each test in its own process).

However, not all test suites isolate their tests: in some, dependencies are allowed to occur between tests. In these cases, common test acceleration techniques such as test selection or test parallelization are unsound in the absence of dependency information. When dependencies go unnoticed, tests can unexpectedly fail when executed out of order, causing unreliable builds. Our second approach, ELECTRICTEST, soundly identifies data dependencies between test cases, allowing for sound test acceleration.

To enable more broad use of general dependency information for testing and other analyses, we created PHOSPHOR, the first and only portable and performant dynamic taint tracking system for the JVM. Dynamic taint tracking is a form of data flow analysis that applies labels to variables, and tracks all other variables derived from those tagged variables, propagating those tags. Taint tracking has many applications to software engineering and software testing, and in addition to our own work, researchers across the world are using PHOSPHOR to build their own systems. Towards making testing more effective, we also created PEBBLES, which makes it easy for developers to specify data-related test oracles on mobile devices by thinking in terms of high level objects such as emails, notes or pictures.

Table of Contents

List of Figures	v
List of Tables	vi
1 Introduction	1
2 Efficiently Isolating Test Dependencies	5
2.1 Motivation	8
2.1.1 MQ1: Do Developers Isolate Their Tests?	8
2.1.2 MQ2: Why Isolate Tests?	10
2.1.3 MQ3: The Overhead of Test Isolation	12
2.2 Approach	14
2.3 Implementation	15
2.3.1 Java Background	16
2.3.2 Offline Analysis	18
2.3.3 Bytecode Instrumentation	19
2.3.4 Logging Class Initializations	20
2.3.5 Dynamically Reinitializing Classes	20
2.3.6 Test Automation Integration	21
2.3.7 Supporting Class Reinitialization	21
2.3.8 Usage	22
2.4 Experimental Results	23
2.4.1 Study 1: Comparison to Minimization	24

2.4.2	Study 2: More Applications	26
2.4.3	Limitations and Threats to Validity	29
2.5	Related Work	30
2.6	Conclusions	32
3	Detecting Data Dependencies Between Tests	33
3.1	Motivation	35
3.1.1	A Study of Java Build Times	36
3.1.2	Danger of Dependent Tests	37
3.1.3	Feasibility of Existing Approaches	39
3.2	Detecting Test Dependencies	42
3.2.1	Detecting In-Memory Dependencies	43
3.2.2	Detecting External Dependencies	46
3.2.3	Reporting and Debugging Dependencies	46
3.2.4	Sound Test Acceleration	47
3.3	Evaluation	49
3.3.1	Accuracy	49
3.3.2	Overhead	51
3.3.3	Impact on Acceleration	52
3.3.4	Discussion and Limitations	55
3.4	Related Work	56
3.5	Conclusions	58
4	Dynamic Data-flow Analysis in the JVM	60
4.1	Motivation	62
4.1.1	Detecting injection attacks	62
4.1.2	Privacy testing and fine grained-access control	63
4.1.3	Testing and Debugging	64
4.2	Approach	64
4.2.1	JVM Background	66
4.2.2	High Level Design	66

4.2.3	Approach Limitations	69
4.3	Implementation	69
4.3.1	Taint Tag Storage	70
4.3.2	Propagating Taint Tags	72
4.3.3	Native Code and Reflection	76
4.3.4	Java-Specific Features	77
4.3.5	Optimizations	78
4.3.6	Application to Android and Dalvik	79
4.3.7	General Usage	80
4.4	Evaluation	84
4.4.1	Performance: Macro benchmarks	84
4.4.2	Performance: Micro Benchmarks	88
4.4.3	Soundness and Precision	91
4.4.4	Portability	91
4.4.5	Threats to Validity	92
4.5	Related Work	93
4.6	Conclusions	95
5	Detecting Hidden Object Structure	97
5.1	Motivation and Goals	99
5.1.1	Example Scenarios	99
5.1.2	Goals and Assumptions	100
5.2	Study: Android Storage Abstractions	103
5.3	The Pebbles Architecture	106
5.3.1	Overview	107
5.3.2	Building the Object Graph	108
5.3.3	LDO Construction and Semantics	111
5.3.4	From User-Level Objects to LDOs	111
5.4	Pebbles-based Tools	112
5.4.1	Breadcrumbs: Auditing Object Deletion	112
5.4.2	PebbleNotify: Tracking Object Exfiltration	114

5.4.3	PebbleDIFC: Object Level Access Control	115
5.4.4	<i>HideIt</i> : Object Level Hiding	116
5.4.5	Other Pebbles-based Tools	116
5.5	Implementation	116
5.6	Evaluation	118
5.6.1	Pebbles Precision and Recall (Q1)	118
5.6.2	Performance Evaluation (Q2)	121
5.6.3	Case Study Evaluation (Q3)	123
5.6.4	Anecdotal User Experience	125
5.6.5	Summary	125
5.7	Discussion	126
5.8	Related Work	126
5.9	Conclusions	128
6	Future Work	130
6.1	Stable tests and stable builds	130
6.2	Supporting debugging and program understanding	131
6.3	Runtime monitoring of deployed software	131
6.4	Build Acceleration	132
7	Conclusions	135
	Bibliography	138
	Appendix: JVM Bytecode Opcode Reference	157

List of Figures

2.1	Typical test execution loop	6
2.2	Unit Test Virtualization at the high level	15
2.3	An example of a leaked reference between two tests	16
2.4	Implementation overview of VMVM	17
4.1	The high level architecture of PHOSPHOR	66
4.2	Code example showing the transformations applied by PHOSPHOR	67
4.3	Implicit flow code example	72
4.4	Operand stack modification example	75
4.5	Example of special case for PHOSPHOR	77
4.6	Arguments accepted by PHOSPHOR	82
4.7	Key API methods, classes and interfaces exposed by PHOSPHOR	83
5.1	OS storage abstraction evolution	103
5.2	Storage API usage in 98 Android applications	104
5.3	Pebbles Architecture	107
5.4	Android email object structure	108
5.5	Object graph construction rules	110
5.6	Breadcrumbs screenshot	113
5.7	Alert screenshots comparing TaintDroid and PebbleNotify	115
5.8	Pebbles performance overhead results on micro benchmarks	121
5.9	Pebbles performance overhead on SQLite microbenchmarks	122

List of Tables

2.1	Statistics for subjects studied in VMVM motivating studies	8
2.2	Number of projects that create a process per test	9
2.3	Overhead of isolating tests in new processes	13
2.4	Comparison of test suite optimization between VMVM and Test Suite Minimization	25
2.5	Results of applying VMVM to 20 projects' test suites	27
3.1	Typical distribution of time among the various phases of Java builds	37
3.2	Testing time and statistics for the 10 longest-running test suites studied	40
3.3	Dependency detection times for DTDetector and ELECTRICTEST	47
3.4	Number of dependencies detected by DTDetector and ELECTRICTEST	48
3.5	Dependencies detected by ELECTRICTEST on 10 large test suites	50
3.6	Relative speedups resulting from parallelizing test suites with and without ELEC- TRICTEST	54
4.1	Runtime overhead results for PHOSPHOR on macro benchmarks	85
4.2	Runtime overhead results for PHOSPHOR on micro benchmarks	86
4.3	Runtime overhead comparison between PHOSPHOR and Trishul [103]	90
5.1	Pebbles LDO API	111
5.2	Pebbles LDO precision and recall	119
5.3	Pebbles performance on application macro benchmarks	123
5.4	PebblesBreadcrumbs findings	124
6.1	Testing statistics for large maven projects	133

1	All JVM bytecodes, annotated with descriptive transformation information	157
---	--	-----

Acknowledgements

I would like to begin by thanking my advisor, Gail Kaiser, for the support and guidance that she has provided me throughout my studies. Gail: it's unlikely that I would have ever started off on this path if you hadn't convinced me when I was an undergrad to give it a try. I would also like to thank Roxana Geambasu, who at a pivotal time in my research helped me see many new perspectives on both my own work and the work of others. I would also like to thank Darko Marinov, who has been invaluable in the last year of my studies, providing an outside opinion on my work, how to present it, and academia as a whole.

Throughout my PhD I've had the privilege of working with many great collaborators and co-authors, both at Columbia and elsewhere, without whom this work could never have happened. In no particular order, I'd like to acknowledge Jason Nieh, Junfeng Yang, Riley Spahn, Swapneel Sheth, Nipun Arora, Chris Murphy, Fang-Hsiang Su, Michael Z Lee, Sravan Bhamidipati, Jeremy Andrus, Mathias Lecuyer, Alex Gyori, Jens Palsberg, Joe Cox, Eric Melski and Mohan Dattatreya. As part of this work, I've also had the pleasure of supervising a number of great undergraduate and graduate students at Columbia: Emilia Pakulski, Alana Ramjit, Jennifer Lam, Xingzhou Derek He, Sidharth Shanker, Miriam Melnick, Alison Yang, Mandy Wang, Winnie Narang, Nikhil Sarda, Ethan Hann, Jason Halpern, Evgeny Fedetov and John Murphy.

The work described in this thesis has been supported by NSF grants CCF-1302269, CCF-1161079, CNS-0905246, and NIH U54 CA121852.

Finally, I would like to thank my parents, siblings, and especially my soon-to-be wife Jackie, who put up with my crazy deadline-driven schedules and always pushed me to keep going.

To my family

Chapter 1

Introduction

As software grows in size and complexity, it also becomes more interdependent. Multiple internal components often share state and data. Whether these dependencies are intentional or not, we have found that their mismanagement often poses several challenges to testing. This thesis seeks to make it easier to create reliable software by making testing more efficient and more effective through explicit knowledge of these hidden dependencies.

Hypothesis: We can make it easier to create reliable software by making testing more efficient and more effective through explicit knowledge of these hidden dependencies.

Faster, More Reliable Builds

As software grows, the time and complexity of building (compiling, linking, testing) these applications has grown as well, and today's developers are faced with increasingly slow builds of hours to days per application. Moreover, sometimes these builds fail, often non-deterministically, after having run for a long time. These slow and flaky builds hurt programmer agility, as the frequency with which code can be built directly impacts the productivity of developers: longer build times mean a longer wait before determining if a change to the application being built was successful.

To characterize the problem better and gain insight into possible remedies, we performed an empirical analysis of build times of open source software, downloading the 1,966 largest and most popular Java projects from GitHub. We attempted to build them automatically “out-of-the-box” (simply using maven to build them using their existing build script), and instrumented the build system to record the amount of time spent in each phase of the build, successfully running 351

builds. Looking across all projects, 41% of the build time (per project) was spent testing, and testing was the single most time consuming build step. Eliminating the projects with particularly short build times (those taking less than 10 minutes to execute all phases of the build), the average testing time increased significantly to nearly 60%. In the projects that took more than an hour to build, nearly all time (90%) is spent testing. Clearly, running tests can take a very long time, and a reduction in testing time can be a big win for developers towards reducing total build time.

We have created two stable approaches to accelerate testing. These approaches leverage observations about *test dependencies* to improve different aspects of testing. While we might assume each test is independent (i.e., that a test's outcome is not influenced by the execution of another test), we and others have found many examples in real software of tests dependencies: some tests require others to run first, or their outcome will change. Previous work has shown that these dependencies are often complicated and hard to find [163].

Efficiently Isolating Tests. In our first approach, *Unit Test Virtualization* (published at ICSE where it received a distinguished paper award [20]), we looked at ways to speed up testing in projects that isolate the in-memory state of each test case in an attempt to prevent dependencies from occurring. One way to isolate JUnit tests is to execute each in its own JVM. However, unit tests are often fairly quick (for instance, taking only 100s of milliseconds), while the time to create a new JVM for each test is relatively steep: 1-2 seconds. With Unit Test Virtualization, we replace this heavyweight approach of running each test in its own JVM with a lightweight, virtualization-like container within a single JVM. Our realization of Unit Test Virtualization for Java, VMVM reduces test suite execution time by an average of 62% in our evaluation (compared to execution time when running each test in its own process).

After publishing VMVM and releasing it on GitHub, we began discussing commercialization with the build acceleration company, Electric Cloud. After reaching out to several of Electric Cloud's clients with very long running builds (some taking over 10 hours), we determined that VMVM would not be applicable in some of their environments because they did not isolate their test cases. In our evaluation of VMVM, we found that 41% of 591 open source projects studied isolate their tests — so we were aware that while VMVM would help many projects, it would not be applicable to all projects.

Detecting Data Dependencies Between Tests. For projects that do not isolate their tests, not

only would VMVM not be applicable, but out-of-the-box test acceleration techniques such as test selection or test parallelization would be unsound. When dependencies go unnoticed, tests can unexpectedly fail when executed out of order, causing unreliable builds. Our second approach, ELECTRICTEST (published at FSE [22]), identifies data dependencies between test cases, allowing for automatic and sound test acceleration.

ELECTRICTEST improves significantly on the prior state of the art in detecting dependencies between tests. The previous approach, DTDetector [163] required executing the various combinations of tests in order to expose dependencies. ELECTRICTEST monitors test execution, detecting data dependencies between tests, adding on average a 20x slowdown to test execution when detecting dependencies. In comparison, applying DTDetector to these same projects showed an average slowdown of 2,276x (using an unsound heuristic not guaranteed to find all dependencies), in most cases requiring more than 10^{308} times the amount of time needed to run the test suite normally in order to exhaustively find all dependencies. ELECTRICTEST makes automated test dependency detection feasible, allowing developers to soundly perform test selection, parallelization, or minimization by ensuring that dependencies are detected and respected.

Dynamic Data Flow Analysis

Dynamic taint tracking is a form of information flow analysis that identifies relationships between data during program execution. Inputs to the program are labeled with a marker (“tainted”), and these markers are propagated through data flow. Traditionally, dynamic taint tracking is used for information flow control, or detection of code-injection attacks. However, dynamic taint tracking also has many software engineering applications.

Taint Tracking in the JVM. In Java, associating metadata (such as tags) with arbitrary variables is very difficult: previous techniques have relied on customized JVMs or symbolic execution environments to maintain this mapping [34, 85, 103], limiting their portability and restricting their application to large and complex real-world software. Without a performant, portable, and accurate tool for performing dynamic taint tracking in Java, software engineering research can be restricted. For instance, Huo and Clause’s *OraclePolish* [85] uses the Java PathFinder (JPF) symbolic execution runtime to implement taint tracking to detect brittle test cases, and due to limitations in JPF, could only be used on 35% of the test cases studied.

To close this gap, we created PHOSPHOR, published originally at OOPSLA [19], with a followup formal demonstration at ISSTA [21]. PHOSPHOR provides taint tracking within the Java Virtual Machine (JVM) without requiring any modifications to the language interpreter, VM, or operating system, and without requiring any access to source code. PHOSPHOR can be easily configured to propagate taint tags through data flow only, or through data flow and control flow. PHOSPHOR can also be configured to combine tags through bitwise OR'ing, through arbitrary dynamic means (using a callback), or can be used simply for applying labels to variables (without propagation), enabling analyses like dynamic def-use pair detection. PHOSPHOR is released under an MIT license on GitHub, and in the year since its publication has been picked up by researchers at UCLA, The University Of Washington [143], The University of Lisbon, Penn State and Duke.

When using taint tracking in information flow control applications, we aim to enforce policies about where data goes. For instance, we may want to ensure that some sensitive data not be exfiltrated from our application or device, or audit that it is properly deleted from all storage locations at some point. Previous system-level tools that provide such functionality require that developers specify (in code) what data is sensitive: but this can be error-prone, and prevents end-users from specifying what data is sensitive to them. As an example of how taint tracking can be used to support testing, we created the PEBBLES system for Android (OSDI [132]), creating a new level of data abstraction called *logical data objects* (LDO), which system level data protection tools can use to track data in unmodified applications.

Chapter 2

Efficiently Isolating Test Dependencies

As developers fix bugs, they often create regression tests to ensure that should those bugs recur, they will be detected by the test suite. These tests are added to existing unit test suites and in an ideal continuous integration environment, executed regularly (e.g., upon code check-ins, or nightly). Because developers are often creating new tests, as software grows in size and complexity, its test suite frequently grows similarly. Software can reach a point where its test suite has gotten so large that it takes too long to regularly execute — previous work has reported test suites in industry taking several weeks to execute fully [119].

To cope with long running test suites, testers might turn to Test Suite Minimization or Test Suite Prioritization [159]. Test Suite Minimization techniques such as [38, 39, 78, 79, 87, 88, 137, 152] seek to reduce the total number of tests to execute by approximating redundant tests. However, identifying which tests are truly redundant is hard, and Test Suite Minimization approaches typically rely on coverage measures to identify redundancy, which may not be completely accurate, leading to a potential loss in fault-finding ability. Furthermore, Test Suite Minimization is an NP-complete problem [79], and therefore existing algorithms rely on heuristics. Test Suite Prioritization techniques such as [53, 55, 119, 134, 151] re-order test cases, for example so that given the set of changes to the application since the last test execution, the most relevant tests are executed first. This technique is useful for prioritizing test cases to identify faults earlier in the testing cycle, but does not actually reduce the total time necessary to execute the entire suite.

Rather than focus our approach on reducing the number of tests executed in a suite, we have set our goal broadly on minimizing the total amount of time necessary to execute the test suite as a

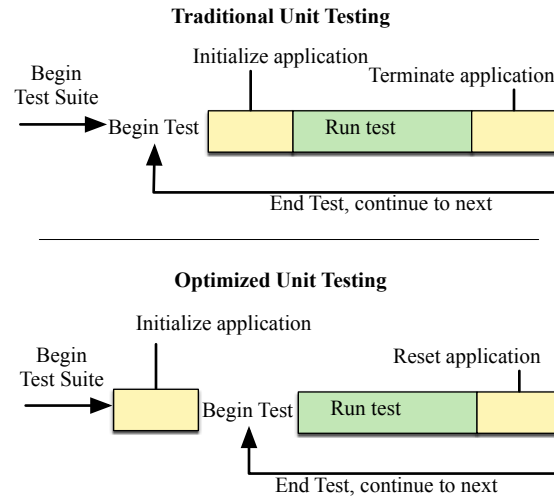


Fig. 2.1: The test execution loop: In traditional unit testing, the application under test is restarted for each test. In optimized unit testing, the application is started only once, then each test runs within the same process, which risks in-memory side effects from each test case.

whole, while still executing all tests and without risking loss in fault-finding ability. We conducted a study on approximately 1,200 large and open source Java applications to identify bottlenecks in the unit testing process. We found that for most large applications each test executes in its own process, rather than executing multiple tests in the same process. We discovered that this is done to isolate the state-based side effects of each test from skewing the results for future tests. The upper half of Figure 2.1 shows an example of a typical test suite execution loop: before each test is executed, the application is initialized and after each test, the application terminates. In our study we found that these initialization steps add an overhead to testing time of up to 4,153% of the total testing time (on average, 618%).

At first, it may seem that the time spent running tests could be trivially reduced by removing the initialization step from the loop, performing initialization only at the beginning of the test suite. In this way, that initialized application could be reused for all tests (illustrated in the bottom half of Figure 2.1), cutting out this high overhead. In some cases this is exactly what testers do, writing pre-test methods to bring the system under test into the correct state and post-test methods to return the system to the starting state.

In practice, these setup and teardown methods can be difficult to implement correctly: develop-

ers may make explicit assumptions about how their code will run, such as permissible in-memory side-effects. As we found in our study of 1,200 real-world Java applications (described further in §2.1), developers often sacrifice performance for correctness by isolating each test in its own process, rather than risk that these side-effects result in false positives or false negatives.

Our key insight is that in the case of memory-managed languages (such as Java), it is not actually necessary to reinitialize the entire application being tested between each test in order to maintain this isolation. Instead, it is feasible to analyze the software to find all potential side-effect causing code and automatically reinitialize only the parts necessary, when needed, in a “just-in-time” manner.

In this chapter we introduce *Unit Test Virtualization*, a technique whereby the side-effects of each unit test are efficiently isolated from other tests, eliminating the need to restart the system under test with every new test. With a hybrid static-dynamic analysis, Unit Test Virtualization automatically identifies the code segments that may create side-effects and isolates them in a container similar to a lightweight virtual machine. Each unit test (in a suite) executes in its own container that isolates all in-memory side-effects to contain them to affect only that suite, exactly mimicking the isolation effect of executing each test in its own process, but with much lower overhead. This approach is relevant to any situation where a suite of tests is executed and must be isolated such as regression testing, continuous integration, or test-driven development.

We implemented Unit Test Virtualization for Java, creating our tool VMVM (pronounced “vroom-vroom”), which transforms application bytecode directly without requiring modification to the JVM or access to application source code. We have integrated it directly with popular Java testing and build automation tools JUnit [3], ant [9] and maven [11], and it is available for download via GitHub [17].

We evaluated VMVM to determine the performance benefits that it can provide and show that it does not affect fault finding ability. In our study of 1,200 applications, we found that the test suites for most large applications isolate each unit test into its own process, and that in a sample of these applications VMVM provides up to a 97% performance gain when executing tests. We compared VMVM with a well known Test Suite Minimization process and found that the performance benefits of VMVM exceed those of the minimization technique without sacrificing fault-finding ability.

The primary contributions of this chapter are:

1. A categorical study of the test suites of 1,200 open source projects showing that developers

isolate tests

2. A presentation of Unit Test Virtualization, a technique to efficiently isolate test cases that is language agnostic among memory managed languages
3. An implementation of our technique for Java, VMVM (released freely via GitHub [17]), evaluated to show its efficacy in reducing test suite runtime and maintaining fault-finding properties

2.1 Motivation

This work would be unnecessary if we could safely execute all of an application’s tests in the same process. Were that the case, then the performance overhead of isolating test cases to individual processes could be trivially removed by running each test in the same process. We have discovered, however, that developers rely on process separation to ensure that their tests are isolated and execute correctly.

In this section, we answer the following three motivation questions to underscore the need for this work.

MQ1: Do developers isolate their unit tests?

MQ2: Why do developers isolate their unit tests?

MQ3: What is the overhead of the isolation technique that developers use?

2.1.1 MQ1: Do Developers Isolate Their Tests?

To answer **MQ1** we analyzed the 1,200 largest open source Java projects listed by Ohloh, a website that indexes open source software [5]. At time of writing, Ohloh indexed over 5,000 individual

	Min	Max	Avg	Std dev
LOC	268	20,280.14k	519.40k	1,515.48k
Active Devs	3.00	350.00	15.88	28.49
Age (Years)	0.17	16.76	5.33	3.24

Table 2.1: Statistics for subjects retrieved from Ohloh

# of Tests in Project	# of Projects Creating New Processes Per Test		Lines of Code in Project	# of Projects Creating New Processes Per Test	
0-10	24/71	(34%)	0-10k	7/42	(17%)
10-100	81/235	(34%)	10k-100k	60/200	(30%)
100-1000	97/238	(41%)	100k-1m	115/267	(43%)
> 1000	38/47	(81%)	> 1m	58/82	(71%)
All Projects	240/591	(41%)	All Projects	240/591	(41%)

Table 2.2: Projects creating a process per test, grouped by tests per project and by lines of code per project

sources such as GitHub, SourceForge and Google Code, comprising over 550,000 projects and over 10 billion lines of code [26]. We restricted ourselves to Java projects in this study due to the widespread adoption of test automation tools for Java, allowing us to easily parse configuration files to determine if the project isolates its test cases (a process described further below).

Using the Ohloh API, we identified the largest open source Java projects, ranked by number of active committers in the preceding 12 months.

From the 1,200 projects, we downloaded the source code for 2,272 repositories (each project may have several repositories to track different versions or to track dependencies). We captured this data between August 15 and August 20, 2013. Basic statistics (as calculated by Ohloh) for these projects appear in Table 2.1, showing the aggregate minimum, maximum, average and standard deviation for lines of code, active developers, and age in years. A complete description of the entire dataset appears in our technical report on VMVM [18].

The two most popular build automation systems for Java are ant [9] and maven [11]. These systems allow developers to write build scripts in XML, with the build system managing dependencies and automatically executing pre-deployment tasks such as running tests. Both systems can be configured to either run all tests in the same process or to create a new process for each test to execute in. From our 1,200 projects, we parsed these XML files to identify those that use JUnit as part of their build process and of those, how many direct JUnit to isolate each test in its own process. Then, we parsed the source files for each of the projects that use JUnit to determine the number of tests in each of these projects.

Next, we broke down the projects both by the number of tests per project and by the number

of lines of code per project. Table 2.2 shows the result of this study. We found that 81% of those projects with over 1,000 tests create a new process for each test when executing it — only 19% do not isolate their tests in separate processes. When grouping by lines of code, 71% of projects with over one million lines of code create new processes for each test case. Overall, 41% of those projects in our sample that use JUnit create separate processes for each test. With these findings, we are confident in our claim that it is common practice, particularly among large applications (which may have the longest running test suites), to isolate each test case into its own process.

2.1.2 MQ2: Why Isolate Tests?

Understanding now that it is common practice for developers to isolate unit tests into separate processes, we next sought to answer **MQ2** — why developers isolate tests.

Perhaps in the ideal unit testing environment each unit test could be executed in the same application process, with pre-test and post-test methods ensuring that the application under test is in a “clean” state for the next test. However, handwritten pre-test and post-test teardown methods can place a burden on developers to write and may not always be correct. When these pre-test and post-test methods are not correct tests may produce false negatives, missing bugs that should be caught or false positives, incorrectly raising an exception when the failure is in the test case, not in the application being tested.

For example, Muşlu et al. [100] discuss a bug in the Apache Commons CLI library that took approximately four years from initial report to reach a confirmed fix. This bug could be detected by running the application’s existing tests independently of each other, but when running on the same instance of the application (using only the developer-provided pre and post-test methods to reset the application), it did not present because it was masked by a hidden dependency between tests that was not automatically reset.

There can be many confounding factors that create such hidden dependencies between tests. For instance, methods may have side effects that are undocumented. In a complex codebase with hundreds of thousands of lines of code, it may be very difficult to identify all potential side effects of an action. When a tester writes the test case for a method, they will be unable to properly reset the system state if they are unaware of that method’s implicit side effects. To avoid this sort of confusion, testers may decide to simply execute each test in a separate process — introducing

significant runtime overhead to their test suite.

In the remainder of this subsection, we describe these dependencies as they appear in the Java programming language and show a real-world example of one such dependency. Although some terminology is specific to Java, these concepts apply similarly to other languages.

Consider the following real Java code snippet from the Apache Tomcat project shown in Listing 2.1. This single line of code is taken from the “CookieSupport” class, which defines a series of configuration constants. In this example, the field “ALLOW_EQUALS_IN_VALUE” is defined with the modifiers `static final`. `static` signifies that it can be referenced by any object, regardless of position in the object graph. The `final` modifier indicates that its value is constant — once it is set, it can never be changed. The value that it is assigned on the right hand side of the expression is derived from a “System Property” (a Java feature that mirrors environmental variables).

This initializer is executed only once in the application: when the class containing it is initialized. If a test case depends on the value of this field then it must set the appropriate system property *before* the class containing the field is initialized. Imagine the following test execution: first, a test executes and sets the system property to false. Then the initializer runs, setting the field `ALLOW_EQUALS_IN_VALUE` to false. Then the next test executes, setting the system property to true, expecting that `ALLOW_EQUALS_IN_VALUE` will be set to true when the field is initialized. However, because the value has already been set it will remain as it is: false, causing the second test to fail unexpectedly. This scenario is exactly what occurs in the Tomcat test suite and in fact, in the source code for several tests that rely on this property the following comment appears: “Note because of the use of static final constants in Cookies, each of these tests must be executed in a new JVM instance” [1].

Although the above example was from a Java application, the sort of leakage that occurred could happen in practically any language, provided that the developers follow a similar pattern. In any situation where a program reads in some configuration from a file and stores it in memory, there is the potential for such leakage.

```
1 public static final boolean ALLOW_EQUALS_IN_VALUE = Boolean.valueOf(System.getProperty(
    ``org.apache.tomcat.util.http.ServerCookie.ALLOW_EQUALS_IN_VALUE``, ``false``)).
    booleanValue();
```

Listing 2.1: CookieSupport.java: An example of Java code that breaks test independence

There are certainly other potential sources of leakage between test executions. For instance in Java, the system property interface mentioned above allows developers to set properties that are persisted for the entire execution of that process. There are also various forms of registries provided by the Java API to allow developers to register services and lookup environments — these too, provide avenues through which data could be leaked between executions.

While in some cases it is possible (although perhaps complicated and time consuming) to write post-test methods to efficiently reset system state, take note that our example, the `static final` field can not be manually reset. The only option left to developers is to re-architect their codebase to make testing easier, for example by removing such fields (at the cost of the time to re-architect it and potential defects introduced by the new implementation) or to isolate each test to a separate process.

2.1.3 MQ3: The Overhead of Test Isolation

To gauge the overhead of test isolation we compared the execution time of several application test suites running in isolation with the execution time running without isolation. From the set of approximately 50 projects that include build scripts with JUnit tests that executed without modification or configuration on our test machine, we selected 20 projects for this study with the aim of including a mix of both widely used and recognizable projects (e.g., the Apache Tomcat project, a popular JSP server with 8537 commits and 15 recent 47 contributors overall), and smaller projects as well (e.g., JTor, an alpha-quality Tor implementation in Java with only 445 commits and 6 contributors overall). Additional information about each project including a direct link to the project repository can be found in our technical report [18].

Modifying each project’s build scripts, we ran the test suite for each project twice: once with all tests executing in the same process, and once with one process per test. Then we calculated the overhead of executing each test in a separate process as $100 \times \frac{T_n - T_o}{T_o}$, where T_n is the absolute time to execute all of the tests in their own process, and T_o is the absolute time to execute all of the tests in the same process. We performed this study on our commodity server running Ubuntu 12.04.1 LTS and Java 1.7.0.25 with a 4-core 2.66Ghz Xeon processor and 8GB of RAM.

Table 2.3 shows the results of this study. For each project studied, we have included the total lines of code in the project (as counted by Ohloh), the overhead of isolating each test in its own pro-

Project	LOC (in k)	Test Classes	Overhead
Apache Ivy	305.99	119	342%
Apache Nutch	100.91	27	18%
Apache River	365.72	22	102%
Apache Tomcat	5692.45	292	42%
betterFORM	1114.14	127	377%
Bristlecone	16.52	4	3%
btrace	14.15	3	123%
Closure Compiler	467.57	223	888%
Commons Codec	17.99	46	407%
Commons IO	29.16	84	89%
Commons Validator	17.46	21	914%
FreeRapid Downloader	257.70	7	631%
gedcom4j	18.22	57	464%
JAXX	91.13	6	832%
Jetty	621.53	6	50%
JTor	15.07	7	1,133%
mkgmap	58.54	43	231%
Openfire	250.79	12	762%
Trove for Java	45.31	12	801%
upm	5.62	10	4,153%
Average	475.30k	56.4	618%

Table 2.3: Overhead of isolating tests in new processes. Bolded applications normally isolate each test case. Additional descriptions of each subject appear in Table 2.5.

cess, and an indicator as to whether that project executes each test in its own process by default. On average, the overhead of executing each test in its own process is stunningly high: 618% on average. We investigated further the subjects “Bristlecone” and “upm,” the subjects with the lowest and highest overhead respectively. We observed that Bristlecone had a low number of tests total (only four test classes in total), with each test taking on average approximately 20 seconds. Meanwhile, in the upm subject, there were 10 test classes total, and each test took on average approximately 0.15 seconds. In general, in test suites that have very fast tests (such as upm), the testing time can be easily dominated by setup and teardown time to create new processes. On the other hand, for test suites with longer running tests (such as Bristlecone), the setup and teardown time is masked by the long duration of the tests themselves.

2.2 Approach

Our key insight that enables Unit Test Virtualization is that it is often unnecessary to completely reinitialize an application in order to isolate its test cases. As shown in Figure 2.2, Unit Test Virtualization fits into a traditional unit testing process. During each test execution, Unit Test Virtualization determines what parts of the application will need to be reset during future executions. Then, during future executions, the affected memory is reset just before it is accessed. This section describes how we determine which parts of the application need to be reset and how we reset just those components.

Unit Test Virtualization relies on both static and dynamic analyses to detect what memory areas need to be reset after each test execution. This approach leverages the runtime performance benefits of static analysis (i.e., that the analysis is precomputed) with the precision of dynamic analysis.

Before test execution, a static analysis pass occurs, placing each addressed memory region into one of two categories: M_s (“safe”) and M_u (“unknown”). Memory areas that are in M_s can be guaranteed to never be shared between test executions, and therefore do not need to be reset. An area might be in M_s because we can determine statically that it is never accessed, or that it is always reset to its starting condition at the conclusion of a test. This static analysis can be cached at the module-level, only needing to be recomputed when the module code changes. All stack memory can be placed in M_s because we assume that the test suite runner (which calls each individual test)

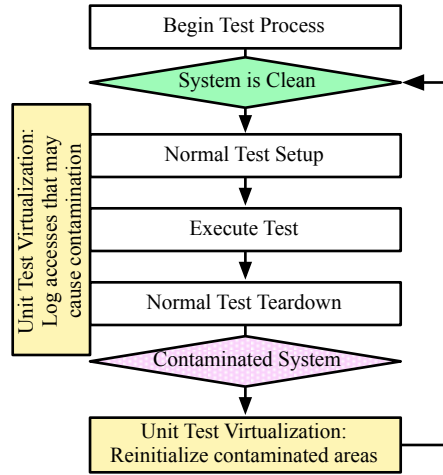


Fig. 2.2: Unit Test Virtualization at the high level

does not pass a pointer to the same stack memory to more than one test (we also assume that code can only access the current stack frame, and no others). We find this reasonable, as it only places a burden on developers of test suite runners (not developers of actual tests), which are reusable and often standardized.

Memory areas that are placed in M_u are left to a runtime checker to identify those which are written to and not cleared. As each test case executes, memory allocations and accesses are tracked, specifically tracking each allocation that occurs in M_u . During future executions we ensure that accesses to that same location in M_u are treated as if the location hadn't been accessed before.

This is a general approach and indeed is left somewhat vague, as the details of exactly how M_s is built and how M_u is checked at runtime will vary from language to language. Further detail for the implementation of Unit Test Virtualization as applied to Java programs is provided in the Implementation section that follows.

2.3 Implementation

To evaluate the performance of Unit Test Virtualization we created a fully-functioning implementation of it for Java. We call our implementation VMVM, named after its technique of building a Virtual Machine-like container within the Java Virtual Machine. VMVM (pronounced “vroom-vroom”) is released under an MIT license and is available on GitHub [17]. VMVM is compatible with any Java bytecode, but the runtime depends on newer language features, requiring a JRE ver-

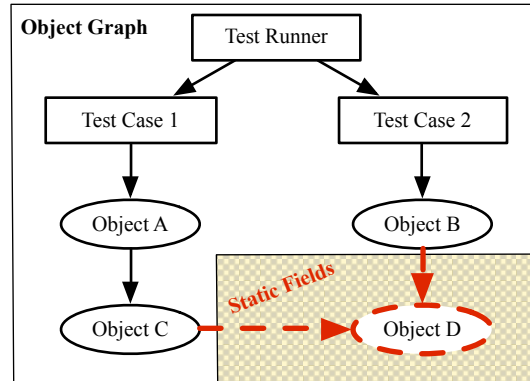


Fig. 2.3: A leaked reference between two tests. Notice that the only link between both test cases is through a static field reference.

sion 5 or newer. We integrated VMVM with the popular test utility JUnit and two common build systems: ant and maven, to reset the test environment between automated test executions with no intervention. VMVM requires no modification to the host machine or JVM, running in a completely unmodified environment. This detail is key in that VMVM is portable to different JVMs running on different platforms. VMVM requires no access to source code, an important feature when testing applications that use third party libraries (for which the source may not be available).

Architecturally, VMVM consists of a static bytecode instrumenter (implemented with the ASM instrumentation library [31]) and a dynamic runtime. The static analyzer and instrumenter identify locations that may require reinitializing and insert code to reinitialize if necessary at runtime. The dynamic runtime tracks what actually needs to be reset and performs this reinitialization between each JUnit test. These components are shown at a high level in Figure 2.4.

2.3.1 Java Background

Before describing the implementation details for VMVM, we first briefly provide some short background on memory management in Java. In a managed memory model, such as in Java, machine instructions can not build pointers to arbitrary locations in memory. Without pointer manipulation, the set of accessible memory S to a code region R in Java is constrained to all regions to which R has a pointer, plus all pointers that may be contained in that region. In an object oriented language, this is referred to as an *object graph*: each object is a node, and if there is a reference from object A to object B , then we say that there exists an edge from A to B . An object can only access other

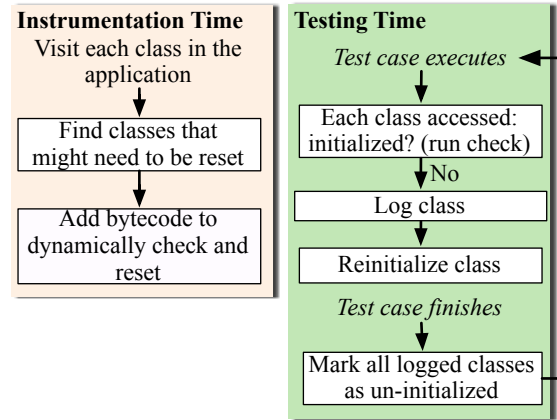


Fig. 2.4: Implementation overview of VMVM

objects which are children in its object graph, with the exception of objects that are referred to by fields declared with the `static` modifier. The `static` keyword indicates that rather than a field belonging to the instances of some object of some class, there is only one instance of that field for the class, and therefore can be referenced directly, without already having a reference to an object of that class. It is easy to see how to systematically avoid leaking data between two tests through non-static references:

Consider the simple reference graph shown in Figure 2.3. Test Case 1 references Object A which in turn references Object C. For Test Case 2 to also reference Object A, it would be necessary for the Test Runner (which can reference Object A) to explicitly pass a reference to Object A to Test Case 2. As long as the test runner never holds a reference to a prior test case when it creates a new one, then this situation can be avoided easily. That is, the application being tested or the tests being executed could not result in such a leak: only the testing framework itself could do so, therefore, this sort of leakage is not of our concern as it can easily be controlled by the testing framework. Therefore, all memory accesses to non-`static` fields are automatically placed in M_s by VMVM, as we are certain that those memory regions will be “reset” between executions.

The leakage problem that we are concerned with comes from `static` fields: in the same figure, we mark “Object D” as an object that is statically referenced. Because it can be referenced by any object, it is possible for Test Case 1 and Test Case 2 to both indirectly access it - potentially leaking data between the tests. It is then only `static` fields that VMVM must analyze to place in M_u or M_s .

2.3.2 Offline Analysis

VMVM must determine which `static` fields are safe (i.e., can be placed in M_s). For a `static` field to be in M_s , it must not only hold a constant value throughout execution, but its value must not be dependent on any non-constant values. This distinction is important as it prevents propagating possibly leaked data into M_s . Listing 2.2 shows an example of a class with three fields that meet these requirements: the first two fields are set with constant values, and the third is set with a value that is non-constant, but dependent only on another constant value. We determine that a field holds a constant value if it is a `final` field (a Java keyword indicating that it is of constant value) referencing an immutable type (note that this is imprecise, but accurate).

In normal operation, when the JVM initializes a class, all `static` fields of that class are initialized. To emulate the behavior of stopping the target application and restarting it (in a fresh JVM), VMVM does not reinitialize individual `static` fields, instead reinitializing entire classes at a time. Therefore, to reinitialize a field, we must completely reinitialize the class that owns that field, executing all of the initialization code of that class (it could be possible to only reinitialize particular fields, but for simplicity of implementation, we did not investigate this approach). As a performance optimization, VMVM detects which classes need never be reinitialized. In addition to having no `static` fields in M_u , the initialization code for these classes must create no side-effects for other classes. If these conditions are met then the entire class is marked as safe and VMVM never attempts to reinitialize it.

This entire analysis process can be cached per-class file, and as the software is modified, only the analysis for classes affected need be recomputed. Even if it is necessary to execute the analysis on the entire codebase, the analysis is fairly fast. We measured the time necessary to analyze the entire Java API (version 1.7.0_25, using the `rt.jar` archive) and found that it took approximately 20 seconds to analyze all 19,097 classes. Varying the number of classes analyzed, we found that the

```
1 public class StaticExample {
2   public static final String s = "abcd";
3   public static final int x = 5;
4   public static final int y = x * 3;
5 }
```

Listing 2.2: Example of static fields

duration of the analysis ranged from 0.16 seconds for 10 classes to 2.74 seconds for 1,000 classes, 12.07 seconds for 10,000 classes, and finally capping out at 21.21 seconds for all 19,097 classes analyzed.

2.3.3 Bytecode Instrumentation

Using the results of the analysis performed in the previous step, VMVM instruments the application code (including any external libraries, but excluding the Java runtime libraries to ensure portability) to log the initialization of each class that may need to be reinitialized. Simultaneously, VMVM instruments the application code to preface each access that could result in a class being initialized with a check, to see if it should be reinitialized by force. Note that because we initialize all static fields of a class at the same time, if a class has at least one non-safe static field, then we must check every access to that class, including to safe fields of the class. The following actions cause the JVM to initialize a class (if it hasn't yet been initialized):

1. Creation of a new instance of a class
2. Access to a static method of a class
3. Access to a static field of a class
4. Explicitly requesting initialization via reflection

VMVM uses the same actions to trigger re-initialization. Actions 1-3 can occur in “normal” code (i.e., by the developer writing code such as `x.someStaticMethod()` to call a method), or dynamically through the Java reflection interface, which allows developers to reference classes dynamically by name at runtime. Regardless of how the class is accessed, VMVM prefaces each such instruction with a check to determine if the class needs to be reinitialized. This check is synchronized, locking on the JVM object that represents the class being checked. This is identical to the synchronization technique specified by the JVM [94], ensuring that VMVM is fully-functional in multithreaded environments. Note that programmers can also write C code using the JNI bridge that can access classes — in these cases, VMVM can not automatically reinitialize the class if it is first referenced from JNI code (instead, it would not be reinitialized until it is first referenced from Java code). In these cases, it would require modification of the native code (at the source level) to

function with VMVM, by providing a hint to VMVM the first time that native code accesses a class. None of the applications evaluated in §2.4 required such changes.

2.3.4 Logging Class Initializations

Each class in Java has a special method called `<clinit>` which is called upon its initialization. For classes that may need to be reinitialized, we insert our logging code directly at the start of this initializer, recording the name of the class being initialized.

We store this logged information in two places for efficient lookup. First, we store the initialization state of the class in a `static` field that we add to the class itself. This allows for fast lookups when accessing a class to determine if it's been initialized or not. Second, we store an index that contains all initialized classes so that we can quickly invalidate those initializations when we want to reinitialize them.

2.3.5 Dynamically Reinitializing Classes

To reinitialize a class, VMVM clears the flag indicating that the class has been initialized. The next time that the class is accessed (as described in §2.3.3), the initializer is called. However, since we only instrument the application code (and not the Java core library set), the above process is not quite complete: there are still locations within the Java library where data could be leaked between test executions.

For instance, Java provides a “System Property” interface that allows applications to set process-wide configuration properties. We scanned the Java API to identify public-facing methods that set `static` fields which are internal to the Java API, first using a script to identify possible candidates, then verifying each by hand to identify false positives. In total, we found 48 classes with methods that set the value of some `static` field within the Java API. For each of these methods, VMVM provides copy-on-write functionality, logging the value of each internal field before changing it, and then restoring that value when reinitializing the application. To provide such support, VMVM prefaces each such method with a wrapper to record the value in a log, and then scans the log at reinitialization time to restore the values.

2.3.6 Test Automation Integration

VMVM plugs directly into the popular unit testing tool JUnit [3] and build automation systems ant [9] and maven [11]. This integration is important as it makes the transition from isolating tests by process separation to isolating tests by VMVM as painless as possible for developers.

Both ant and maven rely upon well-formed XML configuration files to specify the steps of the build (and test) process. VMVM changes approximately 4 lines of these files, modifying them to include VMVM in the classpath, to execute all tests in the same process, and to notify VMVM after each test completion so that shared memory can be reset automatically. As each test completes VMVM marks each class that was used (and not on its list of “safe” classes) as being in need of reinitialization.

Although we integrated VMVM directly into these popular tools, it can also be used directly in any other testing environment. Both the ant and maven hooks that we wrote consist of only a single line of code: `VirtualRuntime.reset()`, which triggers the reinitialization process.

2.3.7 Supporting Class Reinitialization

VMVM makes several technical modifications to the statically mutable classes to allow them to be reinitialized. First, VMVM performs some housekeeping: renaming the static re-initializer from the internal name `<clinit>` to a configurable name (by default, `_vmvm_clinit_`) and adding a shell `<clinit>` method that calls the original. This is necessary because `<clinit>` is a reserved method name that cannot be called explicitly: it would be impossible to call this method otherwise. At the same time, VMVM adds instructions to dynamically log the initialization of the class.

VMVM must also make modifications to support reinitializing `final static` fields. Although a `final` field can’t be changed, if it is a reference to an object, the contents of that object may change. For this reason, VMVM strips the `final` modifier from all mutable static fields of these classes. Note that this occurs *after* the code has been compiled though, so the compiler will still verify that there are no attempts to modify the value of a final field. We manually protect against the code that tries to dynamically set the value of a (no longer) final field with reflection by wrapping all reflective calls with a check to ensure that they do not reference a previously final field.

VMVM makes special accommodations for static fields of interfaces, since Java imposes several requirements on interfaces that limit VMVM from accomplishing the above tasks. In Java, all static

fields of interfaces must be `final`, preventing VMVM from removing the modifier. Also, the only static method allowed is the `<clinit>` method, preventing VMVM from renaming the method. In these cases, VMVM modifies all mutable static fields to be a wrapper object that contains the original object. In this way, the actual value of the `final` field (now, a reference to a wrapper object) does not change when performing a reinitialization: only the contents of that wrapper object change (which represents the original field).

2.3.8 Usage

VMVM is available for download via github [17], and is designed to be easy for developers to use, in a two step process.

First, developers use VMVM to instrument their applications (including dependent libraries) with instructions to support efficient re-initialization. This process uses the ASM [31] byte code manipulation library to automate the instrumentation. VMVM provides a simple interface for instrumenting applications, taking as input a folder containing an application (and all of its dependent libraries) and outputting another folder containing a replica of the input, but with VMVM instrumentation added. The specific usage syntax is `java -cp lib/asm-all-4.1.jar:vmvm.jar edu.co`

```
lumbia.cs.psl.vmm.Instrumenter <folder-to-instrument> <dest>.
```

The second step, after instrumenting their application, is for developers to modify their application test scripts to execute test cases in the same process, and to notify VMVM when a test case completes so that it can reinitialize the effected portions of the application. This notification is made by simply calling our API method, `VirtualRuntime.reset()`. This entire process is simplified, as there are two common build automation systems used in Java, `ant` and `maven`, for which we provide the necessary code and detailed instructions to include VMVM in the testing process.

2.3.8.1 Using VMVM with ant projects

For `ant` projects, developers must only modify their `ant build.xml` file to add to the classpath two jars (the VMVM jar and the VMVM-ANTMVN-LISTENER jar file, which contains classes specific for interacting with `ant`), to add our `ant JUnit` test listener to the configuration, and to execute all test cases in the same process. Specifically, the following lines are added:

```

1 <classpath>
2   <pathelement path="ant-mvn-formatter.jar" />
3   <pathelement location="vmvm.jar" />
4 </classpath>
5 <jvmarg value="-Xbootclasspath/a:vmvm.jar:asm-all-4.1.jar" />
6 <formatter classname="edu.columbia.cs.psl.vmvm.AntJUnitTestListener" extension=".xml" />

```

To modify ant's configuration to execute all of the test cases in the same process, a developer would add the option `forkMode="once"` to the `junit` tag of the `build.xml` file.

2.3.8.2 Using VMVM with maven projects

Developers can modify their maven projects to use the VMVM isolation mechanism by the same two jar files in their test configuration, similar to the process for ant-based systems. In the case of maven, we provide test execution listener that hooks into the surefire testing plugin — developers simply modify their testing configuration to include our jars in the test classpath, and to register our listener:

```

1 <configuration>
2   <additionalClasspathElements>
3     <additionalClasspathElement>vmvm.jar</additionalClasspathElement>
4     <additionalClasspathElement>ant-mvn-formatter.jar</additionalClasspathElement>
5   </additionalClasspathElements>
6   <properties>
7     <property>
8       <name>listener</name>
9       <value>edu.columbia.cs.psl.vmvm.MvnVMVMListener</value>
10    </property>
11  </properties>
12</configuration>

```

2.4 Experimental Results

To evaluate the performance of VMVM we pose and answer the following three research questions (RQ):

RQ1: How does VMVM compare to test suite minimization in terms of performance and fault-finding ability?

RQ2: In general, what performance gains are possible when using VMVM compared to creating a new process for each test?

RQ3: How does VMVM impact fault-finding ability compared to using traditional isolation?

We performed two studies to address these research questions. Both studies were performed in the same environment as our study from §2.1 — on our commodity server running Ubuntu 12.04.1 LTS and Java 1.7.0_25 with a 4-core 2.66Ghz Xeon processor and 8GB of RAM.

2.4.1 Study 1: Comparison to Minimization

We address **RQ1**, comparing VMVM to Test Suite Minimization (TSM), by turning to a study performed by Zhang et al. [161]. Zhang et al. applied TSM to Java programs in the largest study that we could find comparing TSM algorithms using Java subjects. In particular, they implemented four minimization techniques (each implemented four different ways, for a total of 16 implementations): a greedy technique [39], Harrold et al’s heuristic [79], the GRE heuristic [38, 39], and an ILP model [25]. Zhang et al. studied the reduction of test suite size and reduction of fault-finding ability of these TSM implementations using four real-world Java programs as subjects, comparing across several versions of each. The programs were selected from the Software-artifact Infrastructure Repository (SIR) [52]. The SIR is widely used for measuring the performance of TSM techniques, and includes test suites written by the original developers as well as seeded faults for each program.

We downloaded the same 19 versions of the same four applications evaluated in [161] from the SIR and instrumented them with VMVM. We executed each test suite twice: once with each test case running in its own process, and once with all test cases running in the same process but with VMVM providing isolation. The test scripts included by SIR with each application isolate each test case in its own process, so to execute them with VMVM we replaced the SIR-provided scripts with our own, running each in the same process and calling VMVM to reset the environment between each test. For each version of each application, we calculated the reduction in execution time (RT) for both VMVM and TSM as $RT = 100 \times \frac{|T_n| - |T_{new}|}{|T_n|}$ where T_n is the absolute time to execute each test in its own process, and T_{new} is the absolute time to execute all of the tests in the same process using VMVM, or the absolute time to execute the minimized test suite. For each version of the application with seeded tests we calculated the reduction in fault-finding ability (RF) as

$RF = 100 \times \frac{|F_n| - |F_{vmvm}|}{|F_n|}$ where F_n is the number of faults detected by executing each test in its own process and F_{vmvm} is the number of faults detected by executing all tests in the same process using VMVM. Zhang et al. similarly calculated RS as the reduction in total suite size (number of tests) and RF .

Application	LOC (in k)	Test Classes	TSM		VMVM Combined	
			RS	RT	RT	RT
Ant v1	25.83k	34	3%	4%	39%	40%
Ant v2	39.72k	52	0%	0%	36%	37%
Ant v3	39.80k	52	0%	1%	36%	37%
Ant v4	61.85k	101	7%	4%	34%	37%
Ant v5	63.48k	104	6%	11%	25%	26%
Ant v6	63.55k	105	6%	11%	26%	27%
Ant v7	80.36k	150	11%	21%	28%	38%
Ant v8	80.42k	150	10%	18%	27%	37%
JMeter v1	35.54k	23	8%	2%	42%	42%
JMeter v2	35.17k	25	4%	1%	41%	42%
JMeter v3	39.29k	28	11%	5%	44%	48%
JMeter v4	40.38k	28	11%	5%	42%	47%
JMeter v5	43.12k	32	16%	8%	50%	52%
jtopas v1	1.90k	10	13%	34%	75%	77%
jtopas v2	2.03k	11	11%	31%	70%	76%
jtopas v3	5.36k	18	17%	27%	48%	68%
xml-sec v1	18.30k	15	33%	22%	69%	73%
xml-sec v2	18.96k	15	33%	26%	79%	80%
xml-sec v3	16.86k	13	38%	19%	54%	55%
Average	37.47k	51	12%	13%	46%	49%

Table 2.4: Test suite optimization with VMVM and with Harrold et al’s Test Suite Minimization (TSM) technique [79]. We show reduction in test suite size (RS , calculated by [161]) for TSM as well as reduction in test execution time (RT) for TSM, VMVM, and the combination of VMVM with TSM.

Table 2.4 shows the results of this study (RF is not shown in the table, as it is 0 in all cases).

Note that for each subject, Zhang et al. compared 16 minimization approaches, yet we display here only one value per subject. Specifically, Zhang et al. concluded that using Harrold et al.’s heuristic [79] applied at the test case level using statement level coverage (one of the 16 approaches evaluated in their work) yielded the best overall reduction in test suite size with the minimal cost to fault-finding ability. Therefore, in this experiment, we compared VMVM to this recommended technique.

To answer **RQ1**, we found that in almost all cases the reduction in testing time was greater from VMVM than from the TSM technique. On average, VMVM performed quite favorably, reducing the testing time by 46%, while the TSM technique reduced the testing time by only 13%. We also investigated the combination of the two approaches: using VMVM to isolate a minimized test suite, with results shown in the last column of Table 2.4. We found that in some cases, combining the two approaches yielded greater reductions in testing time than either approach alone. However, the speedup is not purely additive, since for every test case removed by TSM, the ability for VMVM to provide a net improvement is lowered (as it reduces the time between tests).

The RF values observed for VMVM are constant at zero, and every test case is still executed in the VMVM configuration. Although the TSM technique also had $RF = 0$ on all seeded faults, such a technique always risks a potential loss of fault finding ability. In fact, studies using the same algorithm on other subjects have found RF values up to 100% [121] (i.e., finding no faults). In general, our expectation is that VMVM results in no loss of fault-finding ability because it still executes all tests in a suite (unlike TSM). Our concerns for the impact of VMVM on fault-finding ability are instead related to its correctness of isolation: does VMVM properly isolate applications? We evaluate the correctness of VMVM further from this perspective in the following study of 20, large, real-world applications.

2.4.2 Study 2: More Applications

To further study the overhead and fault-finding implications of VMVM we applied it to the same 20 open source Java applications used for our motivating study. Most of the applications are well-established, averaging approximately 452,660 lines of code and having an average lifetime of 7 years. These applications are significantly larger than the SIR applications used in Study 1, for which the average application had only 25,830 lines of code. Additional information about each

Project	Revisions	LOC (in k)	Age (Years)	# of Tests		Overhead			False Positives	
				Classes	Methods	VMVM	Forking	RT	VMVM	No Isolation
Apache Ivy	1233	305.99	5.77	119	988	48%	342%	67%	0	52
Apache Nutch	1481	100.91	11.02	27	73	1%	18%	14%	0	0
Apache River	264	365.72	6.36	22	83	1%	102%	50%	0	0
Apache Tomcat	8537	5,692.45	12.36	292	1,734	2%	42%	28%	0	16
betterFORM	1940	1,114.14	3.68	127	680	40%	377%	71%	0	0
Bristlecone	149	16.52	5.94	4	39	6%	3%	-3%	0	0
btrace	326	14.15	5.52	3	16	3%	123%	54%	0	0
Closure Compiler	2296	467.57	3.85	223	7,949	174%	888%	72%	0	0
Commons Codec	1260	17.99	10.44	46	613	34%	407%	74%	0	0
Commons IO	961	29.16	6.19	84	1,022	1%	89%	47%	0	0
Commons Validator	269	17.46	6.19	21	202	81%	914%	82%	0	0
FreeRapid Downloader	1388	257.70	5.10	7	30	8%	631%	85%	0	0
gedcom4j	279	18.22	4.44	57	286	141%	464%	57%	0	0
JAXX	44	91.13	7.44	6	36	42%	832%	85%	0	0
Jetty	2349	621.53	15.11	6	24	3%	50%	31%	0	0
JTor	445	15.07	3.94	7	26	18%	1,133%	90%	0	0
mkgmap	1663	58.54	6.85	43	293	26%	231%	62%	0	0
Openfire	1726	250.79	6.44	12	33	14%	762%	87%	0	0
Trove for Java	193	45.31	11.86	12	179	27%	801%	86%	0	0
upm	323	5.62	7.94	10	34	16%	4,153%	97%	0	0
Average	1356.3	475.30	7.32	56.4	717	34%	618%	62%	0	3.4
Average (Isolated)	1739.3	743.16	8.86	58.7	419	12%	648%	56%	0	6.8
Average (Not Isolated)	973.3	207.43	5.79	54.1	1,015	57%	588%	68%	0	0

Table 2.5: Reduction in testing time (RT) and number of false positives for VMVM over 20 subjects. Here, false positives refer to tests that failed but should have passed. There were no cases of tests passing when intended to fail. We also include the overhead of isolation from both VMVM and creating a new process for each test, as compared to using no isolation at all. Bolded projects isolated their tests by default. The average is segregated into projects that isolate their tests by default, and those that did not isolate their tests.

project appears in our accompanying technical report [18].

For each subject in this study we executed the test suite three times, each time recording the duration of the execution and the number of failed tests. First, we executed the test suite isolating each test case in its own process (what we will refer to as “traditional isolation”). Second, we executed the test suite with no isolation, with all test cases executed in the same process (which we will refer to as “not isolated”). Finally, we instrumented the subject with VMVM and executed all tests cases in the same process but with VMVM providing isolation. We then calculated the reduction in execution time RT as in Study 1 to address **RQ2**. Half of these subjects isolate test cases by default (i.e., half do not normally isolate their tests), yet we include these subjects in this study to show the potential speedup available if the subject did indeed isolate its test cases.

To answer **RQ3** (beyond the evidence found in the first study) we wanted to exercise VMVM in scenarios where we knew that the test cases being executed had side-effects. When tests have side-effects on each other they can lead to false positives (e.g., a test case that fails despite the code being tested being correct) and false negatives (e.g., a test case that passes despite the code being tested being faulty). In practice, we were unable to identify known false negatives, and therefore studied the effect of VMVM on false positives, identifiable easily as instances where a test case passes in isolation but fails without isolation. We evaluated the effectiveness of VMVM’s isolation by observing the false positives that occur for each subject when executed without isolation, comparing this to the false positives that occur for each subject when executed with VMVM isolation. We use the test failures for each subject in traditional isolation as a baseline. In all cases, the same tests passed (or failed) when using VMVM and when using traditional isolation.

The results of this study are shown in Table 2.5. Note that for each application we executed our study on the most recent (at time of writing) development version, identified by its revision number shown in Table 2.5.

On average, the reduction in test suite execution time RT was slightly higher than in Study 1: 62% (56% when considering only the subjects that isolate their tests by default), providing strong support for **RQ2** that VMVM yields significant reductions in test suite execution time. We identified the “Bristlecone” subject as a worst case style scenario that occurred in our study. In our original motivating study (described previously in Table 2.3), we found that there was almost no overhead (3%) to isolating the tests in this subject, due to the relatively long amount of time spent executing

each individual test, and the very few number of tests. Therefore, we were unsurprised to see VMVM provide no reduction in testing time for this subject (and in fact, a slight overhead). On the other hand, we identified the “upm” subject as a near best case: with fast tests, the overhead of creating a new process for each test was very high (4,153%), providing much room for VMVM to provide improvement.

In no cases did we observe any false positives when isolating tests with VMVM, despite observing false positives in several instances when using no isolation at all. That is, no test cases failed when isolated with VMVM that did not fail when executed with traditional isolation. This finding further supports our previous finding for **RQ3** from Study 1, that VMVM does not decrease fault finding ability.

2.4.3 Limitations and Threats to Validity

The first key potential threat to the validity of our studies is the selection of subjects used. However, we believe that by using the standard SIR artifact repository (which is used by other authors as well, e.g., [78,83,137] and more) we can partially address this concern. The applications that we selected for Study 2 were larger on average, a deliberate attempt to broaden the scope of the study beyond the SIR subjects. It is possible that they are not representative of some class of applications, but we believe that they show both the worst and best case performance of VMVM: when there are very few, long running tests and when there are very many, fast running tests.

Our initial claim that these subjects represent the largest Java projects is based on two assumptions: first that number of contributing developers is an indicator of project size, and second that the projects in the Ohloh repository are a representative sample of all Java projects. We believe that we have captured all of the “largest” Java projects in our dataset regardless of the metric, given the very large number of projects retrieved. Additionally, given the overall size of Ohloh’s data (which includes all repositories from, among other sources, GitHub and SourceForge) we believe that our study is at least as broad as previous work by other authors that utilized primarily test subjects from the SIR.

Unit Test Virtualization is primarily useful in cases where the time between tests is a large factor in the overall test suite execution time. Consider an extreme example: if some tests require human interaction, and others are fully automated, then the reduction in total cost of execution by removing

the interaction-based tests from the suite may be significantly higher than what VMVM can provide by speeding up the automated component. If such a scenario arises, then it may be efficient to combine VMVM with Test Suite Minimization in order to realize the benefits of both approaches. However, in the programs studied, this is not the case: no test cases require tester input, and the setup time for each test was significant enough for VMVM to provide a (sometimes quite sizable) speedup.

Although we provide a high level approach to Unit Test Virtualization that is language agnostic (particularly among memory managed languages), we implemented it in Java. The performance benefits that we revealed could be biased to the language features of Java. For instance, it may be that Java programmers more frequently isolate their unit tests in separate processes than other developers, in which case this approach may not provide such large performance benefits to test suites in other languages.

The final limitation that we discuss is the level of isolation provided by VMVM. VMVM is designed to be a drop-in replacement for “traditional” isolation where only in-memory state is isolated between test cases. It would be interesting to extend VMVM beyond this “traditional” isolation to also isolate state on disk or in databases. Such isolation would need to be integrated with current developer best practices, and we consider it to be outside of the scope of this thesis.

There is room for further research in the implementation of VMVM that may be interesting to pursue: for instance, it may be possible to use program slicing to identify initializers for individual fields, hence relieving the need to reinitialize entire classes at a time. Alternatively, we could use the precise information about exactly what dependencies matter to guide our resetting. VMVM was published at ICSE 2014, where it received an ACM SIGSOFT Distinguished Paper Award [20], is currently available publicly on GitHub [17], and has been the basis for an industrial collaboration with the bay area build acceleration company, ElectricCloud.

2.5 Related Work

Unit Test Virtualization can be seen as complementary to Test Suite Minimization (TSM), an approach where test cases that do not increase coverage metrics for the overall suite are removed, as redundant [79]. This optimization problem is NP-complete, and there have been many heuristics

developed to approximate the minimization [38, 39, 78, 79, 87, 88, 137, 152]. TSM can be limited not only by imprecision of minimization approximations but also by the strength of optimization criteria (e.g., statement or branch coverage), a problem potentially abated by optimizing over multiple criteria simultaneously (e.g., [83]). We have shown that it is feasible to combine TSM with Unit Test Virtualization, minimizing both the number of tests executed and the amount of time spent executing those tests.

The effect of TSM on fault finding ability can vary greatly with the structure of the application being optimized and the structure of its test suite. Wong et al. found an average reduction of fault finding ability of less than 7.28% in two separate studies [152, 153]. On larger applications, Rothermel et al. reported a reduction in fault finding ability of over 50% for more than half of the suites considered [121]. Rothermel et al. suggested that this dramatic difference in results could be best attributed to the difference in the size of test suites studied, suggesting that Wong et al.'s [152] selection of small test suites (on average, less than 7 test cases) reduced the opportunities for loss of fault finding effectiveness [121]. The test suites studied in our first study averaged 51 test classes, and the suites in the second study averaged 56 test classes and over 700 individual test methods.

Similar to TSM is Test Suite Prioritization, where test cases are ordered to maximize the speed at which faults are detected, particularly in regression testing [53, 55, 119, 134, 151]. In this way, large test suites can still run in their entirety, with the hopes that faults are detected earlier in the process. We see Test Suite Prioritization and Unit Test Virtualization as complementary (and perhaps, able to be used simultaneously): Unit Test Virtualization increases the rate at which test suites execute, while prioritization increases the rate at which faults are detected by a test suite. To safely perform any of these test selection techniques (in the presence of test order dependencies), it is necessary to either isolate the tests (e.g. with VMVM), or to precisely identify those dependencies so that they can be respected by downstream techniques (a technique described further in 3).

Muşlu et al. studied the effect of isolating unit tests on several software packages, finding isolation to be helpful in finding faults, but computationally expensive [100]. Holmes and Notkin created an approach to identify program dependencies using a hybrid static-dynamic analysis [82], which could be used to detect hidden dependencies between tests. Pinto et al. studied the evolution of test suites throughout several versions of seven real-world Java programs, measuring the sort of changes made to the test suites [116]. It would be interesting to study specifically the kinds of

modifications made to test suites in order to support isolation of unit tests.

Unit Test Virtualization can be seen as similar in overall goal to sandboxing systems [8, 86, 93, 112]. However, while sandbox systems restrict all access from an application (or a subcomponent thereof) to a limited partition of memory, our goal is to allow that application normal access to resources, while recording such accesses so that they can be reverted, more similar to checkpoint-restart systems (e.g., [28, 36, 43, 56, 62]). Most relevant are several checkpointing systems that directly target Java. Nikolov et al. presented recoverable class loaders, allowing for more efficient reinitialization of classes, but requiring a customized JVM [106], whereas VMVM functions on any commodity JVM. Xu et al. created a generic language-level technique for snapshotting Java programs [155], however our approach eliminates the need for explicit checkpoints, instead always reinitializing the system to its starting state.

Unit Test Virtualization may be more similar to microrebooting, a system-level approach to reinitializing small components of applications [33], although microrebooting requires developers to specifically decouple components to enable microrebooting, while Unit Test Virtualization requires no changes to the application under test.

2.6 Conclusions

Unit Test Virtualization is a powerful new approach to reduce the time necessary to execute long test suites by reducing the overhead of isolating individual tests. We have shown the applicability of such an approach by studying 1,200 of the largest Java applications, showing that of the largest, over 80% isolate their test cases, and in general, 40% do. We implemented Unit Test Virtualization for Java, creating our tool VMVM (pronounced “vroom-vroom”), and showed that in our sample of applications, it reduced testing time by up to 97% (on average, 62%), while still executing all test cases and without any loss of fault finding ability. We are interested in exploring further the research challenges of implementing Unit Test Virtualization for non-memory managed languages such as C, as well as the technical challenges in extending VMVM to other languages that target Java byte code (such as Scala). There is also further room for research in the implementation of VMVM: for instance, it may be possible to use program slicing to identify initializers for individual fields, hence relieving the need to reinitialize entire classes at a time.

Chapter 3

Detecting Data Dependencies Between Tests

In our outreach efforts after creating VMVM, we came across several companies with test suites that took over 10 hours to run. While they were initially excited by the possibility of speeding up their testing process with VMVM, it quickly became clear that VMVM would be unable to help them: VMVM provides efficient isolation, but the tests were already un-isolated. We found that in extreme cases of very long running test suites, developers had already abandoned all test case isolation, in search of faster tests. Even in some of the open source software we studied, we found plenty of cases of test suites that did not employ any isolation.

So, perhaps, to make testing faster, these developers may turn to techniques such as Test Suite Minimization (which reduce the size of a test suite, for instance by removing tests that duplicate others) [38, 39, 78, 79, 87, 88, 137, 152], Test Suite Prioritization (which reorders tests to run those most relevant to recent changes first) [53, 55, 119, 134, 151], or Test Selection [64, 80, 109] (which selects tests to execute that are impacted by recent changes). Alternatively, given a sufficient quantity of cheap computational resources (e.g. Amazon’s EC2), we might hope that we could reduce the amount of wall time needed to run a given test suite even further by parallelizing it.

All of these techniques involve executing tests out of order (compared to their typical execution — which may be random but is almost always alphabetically), making the assumption that individual test cases are *independent*. If some test case t_1 writes to some persistent state, and t_2 depends on

that state to execute properly, we would be unable to safely apply previous work in test parallelization, selection, minimization, or prioritization without knowledge of this dependency. Previous work by Zhang et al. has found that these dependencies often come as a surprise and can cause unpredictable results when using common test prioritization algorithms [163].

This assumption is part of the *controlled regression testing assumption*: given a program P and new version P' , when P' is tested with test case t , all factors that may influence the outcome of this test (except for the modified code in P') remain constant [120]. This assumption is key to maintaining the soundness of techniques that reorder or remove tests from a suite. In the case of test dependence, we specifically assume that by executing only some tests, or executing them in a different order, we are not effecting their outcome (i.e., that they are independent).

One simple approach to accelerating these test suites is to ignore these dependencies, or hope that developers specify them manually. However, previous work has shown that inadvertently dependent tests exist in real projects, can take significant time to identify, and pose a threat to test suite correctness when applying test acceleration techniques [95, 163]. Zhang et al. show that dependent tests are a serious problem, finding in a study of five open source applications 96 tests that depend on other tests [163]. In our own study we found many test suites in popular open source software do not isolate their tests, and hence, may potentially have dependencies [21].

While a technique exists for detecting tests that are dependent on each other, its runtime is not favorable (requiring $O(n!)$ test executions for n tests to detect all dependencies or $O(n^2)$ test executions with an unsound heuristic that ignores dependencies between more than two tests) [163], making it impractical to execute. Moreover, the existing technique does not point developers to the specific code causing dependencies, making inspection and analysis of these dependencies costly.

Our new approach and tool, ELECTRICTEST, detects dependencies between test cases in both small and large, real-world test suites. ELECTRICTEST monitors test execution, detecting dependencies between tests, adding on average a 20x slowdown to test execution when soundly detecting dependencies. In comparison, we found that the previous state of the art approach applied to these same projects showed an average slowdown of 2,276x (using an unsound heuristic not guaranteed to find all dependencies), often requiring more than 10^{308} times the amount of time needed to run the test suite normally in order to exhaustively find all dependencies. Moreover, the existing technique does not point developers to the specific code causing dependencies, making inspection and

analysis of these dependencies costly.

With ELECTRICTEST, it becomes feasible to soundly perform test parallelization and selection on large test suites. Rather than detect *manifest dependencies* (i.e., a dependency that changes the outcome of a test case, the definition in previous work by Zhang et al, DTDetector [163]), ELECTRICTEST detects simple data dependencies and anti-dependencies (i.e., read-over-write and write-over-read). Since not all data dependencies will result in manifest dependencies, our approach is inherently less precise than DTDetector at reporting “true” dependencies between tests, though it will never miss a dependency that DTDetector would have detected. However, in the case of long running test suites (e.g. over one hour), the DTDetector approach is not feasible. On popular open source software, we found that the number and type of dependencies reported by ELECTRICTEST allow for up to 16X speedups in test parallelization.

Our key insight is that, for memory-managed languages, we can efficiently detect data dependencies between tests by leveraging existing efficient heap traversal mechanisms like those used by garbage collectors, combined with filesystem and network monitoring. For ELECTRICTEST, test T_2 depends on test T_1 if T_2 reads some data that was last written by T_1 . A system that logs all data dependencies will always report at least as many dependencies as a system that searches for manifest dependencies. Our approach also provides additional benefits to developers: it can report the exact line of code (with stack trace) that causes a dependency between tests, greatly simplifying test debugging and analysis.

3.1 Motivation

To motivate our work, we set out to answer three motivating questions to ground our approach:

MQ1: For those projects that take a long time to build, what component of the build dominates that time?

MQ2: Are existing test acceleration approaches safe to apply to real world, long running test suites?

MQ3: Can the state of the art in test dependency detection be practically used to safely apply test acceleration to these long running test suites?

3.1.1 A Study of Java Build Times

In our previous work [23], we studied 20 open source Java applications to determine the relative amount of build time spent testing, finding testing to consume on average 78% of build time. The longest of these projects took approximately 40 minutes to build, while the shortest completed in under one minute. Given a desire to target projects with very long build times, we wanted to make sure that those very long running builds were also spending most of their time in tests. If we are sure that most of the time spent building these projects is in the testing phase, then we can be confident that a reduction in testing time will have a strong impact in reducing overall build time.

For this study, we downloaded the 1,966 largest and most popular Java projects from the open source repository site, GitHub (those 1,000 with the most forks and stars overall, and those 1,000 with the most forks over 300 MB, as of December 23rd, 2014). From these projects, we searched for only those with tests (i.e., had files that had the word “test” in their name), bringing our list to 921 projects.

Next, we looked at the different build management systems used by each project: there are several popular build systems for Java, such as ant, maven, and gradle. To measure the per-step timing of building each of these projects, we had to instrument the build system, and hence, we selected the most commonly used system in this dataset. We looked for build files for five build systems: ant, maven, gradle, set, and regular Makefiles. Of these 921 projects, the majority (599) used maven, and hence, we focused our study on only those projects using maven due to resource limitations creating and running experiments.

We utilized Amazon’s EC2 “m3.medium” instances, each running Ubuntu 14.04.1 and Maven 3.2.5 with 3.75GB of RAM, 14 GB of SSD disk space, and a one-core 2.5Ghz Xeon processor. We tried to build each project first with Java 1.8.0_40, and then fell back to Java 1.7.0_60 if the newer version did not work (some projects required the latest version while others didn’t support it). For each project, we first built it in its entirety without any instrumentation, and then we built it again from a clean checkout with our instrumented version of Maven in “offline” mode (with external dependencies already downloaded and cached locally).

If a project contained multiple maven build files, we executed maven on the build file nearest the root of the repository, and we did not perform any per-project configuration. Of these 599 projects, we could successfully build 351.

Table 3.1: Typical distribution of time among the various phases of Java builds, showing the top 3 phases only for each category.

Phase	All	Only projects building in:	
	Projects	>10 min	>1 hour
Test	41.22%	59.64%	90.04%
Compile	38.33%	26.25%	8.46%
Package	15.49%		1.05%
Pre-Test		13.51%	

Table 3.1 shows the three longest build phases, first for all of these projects, and then filtering to only those projects that took more than 10 minutes to build (69 projects), and those that took more than one hour to build (8 projects). When looking across all projects, 41% of the build time (per project) was spent testing, and testing was the single most time consuming build step. When eliminating the cases of projects with particularly short build times (those taking less than 10 minutes to execute all phases of the build), the average testing time increased significantly to nearly 60%. In the eight cases of projects that took more than an hour to build, nearly all time (90%) is spent testing. Therefore, to answer **MQ1**, we find that testing dominates build times, especially in long running builds. This conclusion underscores the importance of accelerating testing.

3.1.2 Danger of Dependent Tests

Any test acceleration technique that executes only a subset of tests, or executes them out of order (e.g., test parallelization or test selection) is unsound in the presence of test dependencies. If the result of one test depends on the execution of a previous test, then these techniques may cause false positives (tests that should fail but pass) or false negatives (tests that should pass but fail).

Zhang et al. studied the issue trackers of five popular open source applications to determine if dependent tests truly exist and cause problems for developers [163]. They found a total of 96 dependent tests, 95 of which would result in a false negative when executed out of order (causing a test to fail although it should pass), and one which produced a false positive when executed out of order (causing a test to pass when it should fail). Given that test dependencies exist and can cause tests to behave incorrectly when executed out of order, we conclude that yes: dependent tests pose

a risk to existing test acceleration techniques.

If we isolate the execution of each of our test cases, then dependencies would not be possible. In practice, tests are typically written as single test methods, which are grouped into test classes, which are batched together into modules. Typically each test method represents an atomic test, while test classes represent groups of tests that test the same component. The module separation occurs when a project is split into modules, with a test suite for each module.

Since they are typically testing the same component, individual test methods are never isolated, although sometimes test classes are isolated. Since they represent different modules of code (that must compile separately), test modules are always isolated in our experience. We are interested in detecting dependencies both at the level of individual test methods, and also test classes, which also are the same granularity used by test selection and parallelization techniques. For the remainder of this thesis, when we refer to individual tests, we will refer to test classes and test modules.

One approach to solving the dependent test problem is to simply isolate each test to ensure that no dependencies could occur (e.g., by executing each test in its own process, or by using our efficient isolation system VMVM [20]). However, if the application does not isolate its tests, and tests currently depend on each other, then tests may present false negatives or false positives (albeit deterministically between executions) when isolated.

We examined the 351 Java projects that we built, finding that 18 (or 5%) isolated all of their test classes, and 41 (or 12%) isolated at least some of their test classes (i.e., some classes were isolated and others were grouped together and executed without isolation). The majority of projects did not isolate their tests at all, and therefore are prone to test dependencies occurring, posing a risk to test acceleration.

This result differs from our 2013 study, which showed 41% of 591 Java projects isolated their tests [20]. This study examined only projects that built with maven, while our previous study (which was performed through a static analysis of build scripts) examined both maven and ant-building projects. In our previous study, we found that of our 591 projects, only approximately 10% of those that used maven to build and run their tests isolated some or all of their tests, a number much more similar to what we found here.

Due to the risks that they impose and ability to occur (when tests aren't isolated), our goal is to detect dependencies between test classes so that we can (1) inform existing test acceleration

techniques of the dependencies to ensure sound acceleration, and (2) provide feedback to developers so that they are aware of dependencies that exist.

3.1.3 Feasibility of Existing Approaches

Finally, we study the existing state-of-the-art approach for detecting dependencies between test cases to determine if it is feasible to apply to long-running test suites.

If we define a test dependence as the case where executing some set of tests T in a different order changes the result of the test(s), then identifying test dependencies is NP-Complete [163]. This definition for dependence (henceforth referred to as a *manifest dependence*) is more narrow than ours (a distinction described later in §3.2), but is the definition used in the state-of-the-art work by Zhang et al. [163].

To identify all manifest test dependencies in a suite of n tests we would have to execute every permutation of those n tests, requiring $O(n!)$ test executions, clearly infeasible for any reasonably large test suite. Moreover, such a technique would only identify that tests are dependent, and not the specific resource or lines of code causing the dependence, making it difficult for developers who wish to examine or remove the dependency. In our study that follows, we estimated that this exhaustive process often would take more than 1×10^{308} times longer than running the test suite normally. Zhang et al. propose two techniques to reduce the number of test executions needed to detect manifest dependent tests, both of which they acknowledge may not scale to large test suites [163].

In one approach, they reduce the search space to $O(n^2)$ by suggesting that most dependencies manifest between only two tests, with no need to consider every possible n size permutation. However, this is incomplete: there may be dependencies that only manifest when more than two tests interact. They further reduce the search space by a constant factor (it is still an $O(n^2)$ algorithm) by only checking test combinations that share common resources (defined to be static fields and files). If two tests access (read or write) the same file or static field, then they are marked as sharing a common resource, regardless of whether a true data dependency exists or not. Since this very coarse dependency detection will likely result in many false positives, Zhang et al. manually inspect each resource to determine if it is likely to cause a dependence, and if not, ignore it in this process. This heuristic can limit the search space but it still can remain large, and requires manual effort to rule

Table 3.2: Testing time and statistics for the 10 longest-running test suites studied with unisolated tests, plus the 4 projects studied in previous work by Zhang et al. [163]. In addition to the normal testing time, we estimate the time that needed to run all pairwise combinations of tests, and the time needed to exhaustively run all combinations.* indicates a slowdown greater than 1×10^{308} .

Project	Test Classes	Test Methods	Testing	Pairwise Test		Exhaustive Test	
			Time	Slowdown		Slowdown	
			(mins)	Class	Method	Class	Method
Projects selected in §3.1.3	camel	5,919	13,562	109.70	1,865X	8,045X	*1E+308X
	crunch	62	243	17.58	65X	298X	54E+82X
	hazelcast	297	2,623	47.37	147X	2,536X	*1E+308X
	jetty.project	554	5,603	20.08	35X	2,555X	2E+60X
	mongo-java-driver	58	576	74.25	58X	649X	4E+76X
	mule	2,047	10,476	117.45	250X	3,438X	*1E+308X
	netty	289	4,601	62.95	11X	2,725X	62E+82X
	spring-data-mongodb	141	1,453	121.38	136X	1,715X	3E+230X
	tachyon	53	362	34.47	47X	397X	56E+56X
	titan	177	1,191	81.82	181X	398X	*1E+308X
Average		960	4,069	68.71	279X	2,276X	*1E+308X
Zhang [163]	joda-time	122	3,875	0.27	627X	418,016X	3E+204X
	xml security	19	108	0.37	59X	1,316X	47E+16X
	crystal	11	75	0.07	37X	763X	3E+8X
	synoptic	27	118	0.03	183X	3,497X	5E+28X
	Average	45	1,044	0.18	226X	105,898X	70E+202X

out some resource accesses that will not cause manifest dependencies.

Table 3.2 shows the estimated CPU time needed to detect the dependent tests in each of the ten longest building projects from our dataset from §3.1.1 with unisolated tests, along with the four projects studied by Zhang et al. previously [163]. This experiment was performed on Amazon EC2 “r3.xlarge” instances, each running Ubuntu 14.04.1 and Maven 3.2.5 with 4 virtualized Intel Xeon X5-2670 v2 2.5Ghz CPUs, 30.5 GB of RAM and 80 GB of SSD storage. Subjects ‘jetty’, ‘titan’ and ‘crunch’ were evaluated on OpenJDK Java 1.7.0_60 (the most recent version supported by the projects) while the others were evaluated on OpenJDK Java 1.8.0_40.

In the case of the four small projects previously studied by Zhang et al, we used their publicly

available tool to calculate the pairwise testing time, and estimated the exhaustive testing time. In the case of our ten projects, we estimate all times, due to scaling limitations of the DTDetector tool. We estimated all times using the following approach: first we measured the time to run each test normally and then we calculated the permutations of tests to run for each module of each project (most of these projects had many modules with tests, and since tests from different modules were isolated, there was no need to include permutations cross-module). We added a constant time of 1 second to each combination of tests executed to account for the time needed to start and stop the JVM and system under test (a conservative estimate based on our prior results [20]).

We compare this projected time to the actual time needed to run the test suite in its normal configuration, presenting the slowdown as $T_{DTDetector}/T_{normal}$. Even the pairwise heuristic (examining only every 2-pair of tests, rather than all possible permutations) can be cost prohibitive: adding an overhead of up to 418,016X (minimum 298X for test methods), even though there is no guarantee of its correctness. A large slowdown appears in both long-building and fast building projects.

For the four projects previously studied by Zhang et al., the dependence-aware approach showed approximately one order of magnitude less overhead. However, we were unable to evaluate the dependence-aware technique on our ten projects due to technical limitations of the DTDetector implementation: running it requires manual enumeration and configuration of each test to run in the DTDetector test runner. Given the manual effort required and that this heuristic is unsound, we chose not to implement it for our ten projects.

As expected, there is no situation in the projects that we studied where the fully exhaustive method (testing all possible permutations) is feasible. Even in the cases of the more modest length test suites, the overhead of DTDetector is very high. We answer MQ3 and conclude that the existing, state of the art approach for detecting dependencies between tests can not scale to detect dependencies in the wild, except when using unsound heuristics on the very smallest of test suites that took less than a minute to execute normally.

3.2 Detecting Test Dependencies

While previous work in the area has focused on detecting *manifest dependencies* between tests [163], we focus instead on a more general definition of dependence. For our purposes, if T_2 reads some value that was last written by T_1 , then we say that T_2 depends on T_1 (i.e., there is a data dependence). If some later test, T_3 writes over that same data, then we say that there is an anti-dependence between tests T_2 and T_3 : T_3 must never run between T_1 and T_2 . Note that any two tests that are *manifest dependent* will also be dependent by our definition, but two tests that have a data dependence may not have a manifest dependence.

Consider the case of a simple utility function that caches the current formatted timestamp at the resolution of seconds so that multiple invocations of the method in the same second returns the same formatted string. If the date formatter has no side-effects we can surmise that if several tests call this method, while there is a data dependency between them (since the cache is reused), this dependence won't in and of itself influence the outcome of any tests. Hence, there will be no manifest dependence between these tests even though there is a data dependence.

While detecting *manifest dependencies* between tests may require executing every possible permutation of all tests, detecting data dependencies (that may or may not result in manifest dependencies) requires that each test is executed only once. ELECTRICTEST detects dependencies by observing global resources read and written by each test, and reports any test T_j that reads a value last written by test T_i as dependent. ELECTRICTEST also reports anti-dependencies, that is, other tests T_k that write that same data after T_j , to ensure that T_k is not executed between T_i and T_j .

ELECTRICTEST consists of a static analyzer/instrumenter and a runtime library. Before tests are run with ELECTRICTEST, all classes in the system under test (including its libraries) are instrumented with heap tracking code (at the bytecode level — no access to source code is required). In principle, this instrumentation could occur on-the-fly during testing as classes are loaded into JVM, however, we perform the instrumentation offline for increased performance, as many external library classes may remain constant between different versions of the same project. This process is fairly fast though: analyzing and instrumenting the 67,893 classes in the Java 1.8 JDK took approximately 4 minutes on our commodity server. ELECTRICTEST detects dynamically generated classes that are loaded when testing (which were not statically instrumented) and instruments them on the fly. During test execution, the ELECTRICTEST runtime monitors heap accesses to detect

dependencies between tests.

Dependencies between tests can arise due to shared memory, shared files on a filesystem, or shared external resources (e.g. on a network). ELECTRICTEST's approach for file and network dependency detection is simple: it maintains a list of files and network socket addresses that are read and written during each test. ELECTRICTEST leverages Java's built in *IOTrace* support to track file and network access. Efficiently detecting in-memory dependencies is much more complex, and we focus our discussion to this technique next.

3.2.1 Detecting In-Memory Dependencies

To detect dependencies in memory between test cases, ELECTRICTEST carefully examines reads and writes to heap memory. Recall that Java is a memory managed language, where it is impossible to directly address memory. Simply put, the heap can be accessed through pointers to it that already exist on the stack, or via `static` fields (which reside in the heap and can be directly referenced).

At the start of each test, we'll assume that the test runner (which is creating these tests) does not pass (on the stack) references to heap objects, or at least not the same reference to multiple tests. We easily verified this safe assumption, as there is typically only a single test runner that's shared between all projects using that framework (e.g. JUnit, which creates a new instance of each test class for each test).

Therefore, our possible leakage points for dependencies between tests will arise through `static` fields. `static` fields are heap roots: they are directly accessed, and therefore the level of granularity at which we detect dependencies.

Unfortunately, to soundly detect all possible dependencies, it is insufficient to simply record accesses to `static` fields, since each `static` field may in turn point to some object which has other `instance` fields. If we simply recorded only accesses to `static` fields (as our previous work, VMVM did [20]), we wouldn't be able to detect all data dependencies, since we wouldn't be able to follow all pointers. With VMVM, we were forced to treat all `static` field reads as writes, since a test might read a `static` field to get a pointer to some other part of the heap, then write that other part (indirectly writing the area referenced by the `static` field).

Our key insight is that we can efficiently detect these dependencies by leveraging several powerful features that already exist in the JVM: garbage collection and profiling. The high level ap-

proach that ELECTRICTEST uses to efficiently detect in-memory dependencies between test cases is twofold. At the end of each test execution, we force a garbage collection and mark any reachable objects (that weren't marked as written yet) as written in this test case. During the following test executions, we monitor all accesses to the marked objects, and if a test reads an object that was written during a previous test, we tag it as having a dependence on the last test that wrote that object. This method is similarly used to detect anti-dependencies (write after read).

ELECTRICTEST heavily leverages the JVM Tooling Interface (JVMTI), which provides support for internal monitoring and is used for implementing profilers and debuggers that interact with the JVM [108]. While it would be possible (and likely more performant) to implement ELECTRICTEST by modifying certain aspects of the JVM directly (e.g. to piggy-back generation counters already used for garbage collection to track which test wrote an object), we chose instead to use the standard JVMTI interface so that ELECTRICTEST will not require a specialized JVM: it functions on commodity JVMs such as Oracle's HotSpot or OpenJDK's IcedTea.

Aside from bytecode instrumentation, ELECTRICTEST utilizes three key functions of the JVMTI API: heap walking, heap tagging, and heap access notifications. The heap walking mechanism provides a fairly efficient means whereby we can visit every object on the heap, descending from root nodes down to leaves. Heap tagging allows us to associate objects with arbitrary 64-bit tags, useful for storing information about the status of each object (i.e., which test last read and wrote it) and each static field. Finally, heap access notifications allows us to register callbacks for the JVM to notify ELECTRICTEST when specific objects or their primitive fields are written or read (when instance fields are accessed).

Observing New Heap Writes. For each test execution, ELECTRICTEST needs to be able to efficiently determine what part of the heap was written by that test. We have optimized this process for cases where the majority of data created on the heap is not shared between tests (which we have found to be a common case). As objects are created on the heap during test execution, ELECTRICTEST does nothing. At the end of each test, after performing a garbage collection, ELECTRICTEST uses JVMTI to scan for all objects that have no tag associated with them (i.e., those not yet tagged by ELECTRICTEST). Each untagged object is tagged with a counter indicating that it was created in the current test case. Objects are also tagged with the list of static fields from which they are accessible. Since the only objects that still exist after the test completes are those that can

be shared between tests, this method avoids unnecessarily tagging and tracking objects that can't be part of dependencies.

Observing Heap Reads and Writes of Old Data. Aside from references on the stack, data on the JVM's heap is accessed through fields of objects, static fields of classes, or array elements. The easiest type of heap access to observe is to the fields of objects, which ELECTRICTEST accomplishes through JVMTI's field tracking system. For each class of object created in a previous test but still reachable in the current test, ELECTRICTEST registers a callback through JVMTI to be notified whenever any fields of those objects are read or written.

When an object is read or written, ELECTRICTEST checks its tag to see if it was last written or read in a previous test: if so, then it is marked as causing a dependency on the last test that wrote that object, and we note this dependence to report at the end of the test. This technique will detect both data dependencies (read after write) and anti-dependencies (write after read), reporting them independently.

Detecting reads and writes of static fields and array elements is more complicated, as there is no similar callback to use. Instead, ELECTRICTEST relies on bytecode instrumentation, modifying the bytecode of every class that executes to directly notify ELECTRICTEST of reads and writes. In its instrumentation phase, ELECTRICTEST employs an intraprocedural data flow analysis to reduce the number of redundant calls that it makes to record reads and writes on the same value by inferring which arrays and fields have already been read or written before each instruction. ELECTRICTEST also dynamically detects reads and writes through Java's reflection interface by intercepting all calls to the reflection API and adding a call to the ELECTRICTEST runtime library to record the access.

Detecting Dependencies at Static Fields. At the end of each test, we perform a heap walk, rooted at every static field, visiting all objects that are reachable from each static field. We maintain a simple stop-list of common static fields within the Java API that are manually-verified as deterministically written and hence may be ignored in this process. These fields include fields such as `System.out`, which is the stream handler for standard output. While it is possible to modify these fields to point to a different object, their default value is always deterministically created. Therefore, a dependence on the default value of one of these fields can safely be ignored (a dependence on the *non-default* value is *not* ignored), since we can assume that this field would have the same value independent of which test first accessed it. This mechanism also allows developers to easily filter

specific fields that are known to be data-dependent between executions, but benign (e.g. internals of logging mechanisms). A simple configuration file maintains a stop-list of fields for ELECTRICTEST to ignore. ELECTRICTEST marks all objects at the end of each test with the list of static fields that point to it.

3.2.2 Detecting External Dependencies

ELECTRICTEST leverages the JVM's built-in *IOTrace* features to track access to external resources. When code attempts to access a file or socket, ELECTRICTEST gets a callback identifying which file or network address is being accessed. All tests that access the same file or socket are marked as dependent. This relatively coarse approach is based on our observation that tests infrequently share access to the same file or network resource — or that if they do, they are dependent. While it would be possible to have a finer grained approach to detecting these dependencies (e.g. by tracing the exact data read and written), as our evaluation shows in the following section, the coarse grained approach is sufficient to allow for reasonable test suite acceleration in the projects we studied.

3.2.3 Reporting and Debugging Dependencies

Once all dependencies have been detected, ELECTRICTEST can be used to help developers analyze and inspect them. While manual inspection is not required for sound test acceleration (the following section will describe how ELECTRICTEST does this automatically), we imagine that in some cases developers will want to understand the dependencies between tests in their projects. For instance, perhaps some dependencies may be indicative of incorrectly written tests. We expect that in some cases developers may want to investigate dependencies to make sure that they are intentional. Alternatively, developers may want to mark some dependencies as benign: perhaps multiple tests intentionally share resources, but do so in a way that doesn't create a functional dependence. For instance, we have seen many test suites that intentionally share state between tests to reduce setup time: each test checks if the shared state is established and if not, initializes it, and resets it to this initial state when done.

ELECTRICTEST supports developers analyzing dependencies by providing a complete stack-trace showing how a dependency occurred. Stack traces are trivially collected when a test reads data previously written since ELECTRICTEST is detecting data dependencies in real-time, within the

Table 3.3: Dependency detection times for DTDetector and ELECTRICTEST using the same subjects evaluated in [163]. We show the baseline runtime of the test suite as well as the running time for three configurations of DTDetector: the 2-pair algorithm, the dependence-aware 2-pair algorithm, and the exhaustive algorithm. Execution times for DTDetector on Joda and with the Exhaustive algorithm (marked with *) are estimations based on the same methodology used by the authors of DTDetector [163].

Project	# of Tests		Testing Time (Seconds)					ELECTRICTEST
			DTDetector				Speedup vs	
	Classes	Methods	Baseline	All 2-pair	Dep-Aware Pairs	Exhaustive	ELECTRICTEST	Dep-Aware
Joda	122	3875	16	*6,688,250	*657,144	*1E+308	2122	310X
XMLSecurity	19	108	22	28,958	5,500	*3E+174	57	96X
Crystal	11	75	4	3,050	874	*14E+108	22	40X
Synoptic	27	118	2	6,993	2,070	*2E+194	34	61X

executing program and JVM. In this way, ELECTRICTEST provides significantly more information than previous work in test dependency detection [163], which could only report that two tests were dependent.

3.2.4 Sound Test Acceleration

Given the list of dependencies between tests, we can soundly apply existing test acceleration techniques such as parallelization and prioritization. Naive approaches to both are straightforward, but may be sub-optimal. For instance, a naive approach for running a given test is to ensure that all tests that must run before it have just run (in order).

Haidry and Miller proposed several techniques for efficiently and effectively prioritizing test suites in light of dependencies [74]. Rather than consider prioritization metrics (e.g. line coverage) for a single test, entire dependency graphs are examined at once. Their techniques are agnostic to the dependency detection method (relying on manual specification in their work), and would be easily adapted to consume the dependencies output by ELECTRICTEST.

We propose a simple technique to improve parallelization of dependent tests based on historical test timing information. Our optimistic greedy round-robin scheduler observes how long each test

Table 3.4: Dependencies detected by DTDetector (DTD) and ELECTRICTEST (ET). For ELECTRICTEST, we group dependencies, into tests that write a value which others read (**W**) and tests that read a value written by a previous test (**R**).

Project	Dependencies			ET Shared Resource	
	DTD	ET		Locations	
		W	R	App Code	JRE Code
Joda	2	15	121	39	12
XMLSecurity	4	3	103	3	15
Crystal	18	15	39	4	19
Syntopic	1	10	117	3	14

takes to execute and combines this data with the dependency tree to opportunistically achieve parallelism. Consider the simple case of ten tests, each of which take 10 minutes to run, all dependent on a single other test that takes only 30 seconds to run (but not dependent on each other). If we have 10 CPUs to utilize, we can safely utilize all resources by first running the single test that the others are dependent on on each CPU (causing it to be executed 10 times total), and then run one of the remaining 10 tests on each of the 10 CPUs. The testing infrastructure can then filter the unnecessary executions from reports.

ELECTRICTEST generates schedules for parallel execution of tests using a greedy version of this algorithm, re-executing a single test multiple times on multiple CPUs when doing so would decrease wall time for execution. ELECTRICTEST also can speculatively parallelize test methods by breaking simple dependencies. When one test depends on a simple (i.e., primitive) value from another test, ELECTRICTEST will allow the dependent test to run separately from the test it depends on, simulating the dependent value. If ELECTRICTEST runs the test writing that value and finds that it writes a different value than was replayed, the pair of tests are re-executed serially. In our evaluation that follows, we show that most dependencies are on a small number of tests, allowing this simple algorithm to greatly reduce the longest serial chain of tests to execute in parallel.

3.3 Evaluation

We evaluated ELECTRICTEST across three dimensions: accuracy, runtime performance, and impact on test acceleration techniques. For accuracy, we compare the dependencies detected by ELECTRICTEST to the state-of-the-art tool, DTDetector [163]. In terms of performance, we measured the overhead of running ELECTRICTEST on Java test suites compared to the normal running time of the test suite. Given that ELECTRICTEST may report non-manifest dependencies (that is, those that need not be respected in order to maintain the integrity of the test suite), we are particularly interested in the impact of ELECTRICTEST’s detected dependencies on test acceleration. To determine the impact of ELECTRICTEST on test acceleration, we measured the longest chain of data dependencies and number of anti-dependencies in each of these large test suites to identify how effective test parallelization and selection could be when respecting the dependencies automatically detected by ELECTRICTEST.

All of these experiments were performed in the same environment as our previous experiment in §3.1.3: Amazon EC2 r3.xlarge instances with 4 2.5Ghz CPUs and 30.5 GB of RAM (more details on this environment are in §3.1.3).

3.3.1 Accuracy

We evaluated the accuracy of ELECTRICTEST by comparing the dependencies detected between test methods with those detected by Zhang et al.’s tool, DTDetector [163]. Table 3.4 shows the dependencies detected by each tool. ELECTRICTEST detected all of the same dependencies identified by DTDetector, plus some additional dependencies. We therefore conclude that ELECTRICTEST’s recall is at least as good as the existing tool, DTDetector.

We can directly attribute the additional dependencies to the different definition of dependencies employed by the two systems: ELECTRICTEST detects all data dependencies, whereas DTDetector detects tests that have different outcomes when executed in a different order (manifest dependencies). Since not all data dependencies will result in manifest dependencies, we expect that ELECTRICTEST reports more dependencies than DTDetector.

For the purposes of test acceleration, given the computational ability to execute DTDetector on the test suite under scrutiny, it may still be preferable to use it over ELECTRICTEST. However, as

Table 3.5: Dependencies found by ELECTRICTEST on 10 large test suites. We show the number of tests in each suite, the time necessary to run the suite in dependency detection mode, the relative overhead of running the dependency detector (compared to running the test suite normally), the number of dependent tests, and the number of resources involved in dependencies. For dependencies, we report the number of tests that write a resource that is later read (W), the number of tests that read a previously written resource (W), and the longest serial chain of dependencies (C). For dependencies and resources in dependencies, we report our findings at the granularity of test classes and test methods.

			Analysis	Analysis	Test Dependencies						# Resources in-	
	Number of Tests		Time	Relative	Classes			Methods			volved at level:	
Project	Classes	Methods	(Min)	Slowdown	W	R	C	W	R	C	Classes	Methods
camel	5,919	13,562	2,449	22.3X	1,977	3,465	1,356	4,790	8,399	1,695	4,944	5,490
crunch	62	243	165	9.4X	9	20	6	18	43	18	190	207
hazelcast	297	2,623	1,780	37.6X	174	200	186	1,163	1,261	1,482	941	1,020
jetty.project	554	5,603	184	9.2X	223	261	54	4,016	4,079	424	713	828
mongo-java-driver	58	576	103	1.4X	36	36	34	342	362	357	32	33
mule	2,047	10,476	9,698	82.6X	185	859	119	2,049	6,279	1,400	11,844	12,387
netty	289	4,601	338	5.4X	128	120	63	2,928	3,297	2,926	640	1,104
spring-data-mongodb	141	1,453	364	3.0X	114	130	110	1,407	1,404	1,401	1,469	1,489
tachyon	53	362	89	2.6X	9	13	9	55	93	13	125	157
titan	177	1,191	2,262	27.7X	118	126	46	429	877	40	1,433	1,562
Average	960	4,069	1,743	20.0X	297	523	198	1,720	2,609	976	2,233	2,428

discussed in §3.1.3, it is often infeasible to use DTDetector on projects of reasonable size. Moreover, in cases where developers want to debug a dependency, ELECTRICTEST would still be preferable over DTDetector (which would only tell the developer that two tests had a dependency).

Interestingly, all dependencies detected by ELECTRICTEST were caused by shared accesses to a very small number of resources (static fields in this case). Most of the data dependences were caused by references to static fields within the JRE made by JRE code (not by application code), and all such references were to fields of primitive types, allowing for the opportunistic parallelism described in §3.2.4. This is also interesting in that it may make manual investigation of dependencies by developers easier: even if many tests are dependent, the number of actual resources shared is small.

3.3.2 Overhead

We evaluated the overhead of ELECTRICTEST on the same four subjects studied by Zhang et al. [163], in addition to the ten large Java projects described earlier in §3.1.3.

We reproduced Zhang et al.’s experiments [163] in our environment to provide a direct performance comparison between the two tools. Table 3.3 shows the runtime of the same four test suites evaluated by Zhang et al., presenting the baseline test execution time, the DTDetector execution time and the ELECTRICTEST execution time. None of the DTDetector algorithms we studied provided reasonable performance on the *Joda* test suite, and the exhaustive technique was infeasible in all cases. Even in the case of the dependence-aware optimized heuristics (which is not guaranteed to detect all dependencies), ELECTRICTEST still ran significantly faster than DTDetector.

However, these test suites were all very small, with the longest taking only 22 seconds to run. We applied ELECTRICTEST to the ten large open source projects with unisolated tests previously discussed in §3.1.3, recording the number of dependencies detected and the time needed to run the tool.

Table 3.5 shows the results of this study, showing the number of test classes and test methods in each project, along with the time needed to detect dependencies, the relative slowdown of dependency detection compared to normal test execution, and the number of dependent tests detected. For dependencies, we report dependence at both the level of test classes and test methods (in the case of test classes, we report the dependencies between entire test classes, and not the dependencies

between the methods in the same test class). We report the number of tests writing values (W) that are later read by dependent tests (R), as well as the size of the longest serial chain (C). Finally, we report the distinct number of resources involved in dependencies between tests, both at the test class and test method level.

In general, far more tests caused dependencies (i.e., wrote a shared value) than were dependent (i.e., read a shared value). The longest critical path was fairly short (relative to the total number of tests) in almost all cases, indicating that test parallelization or selection may remain fairly effective. Given infinite CPUs to parallelize test execution across, the maximum speedup possible is restricted by this measure.

ELECTRICTEST imposed on average a 20X slowdown compared to running the test suite normally to detect all dependencies between test methods or classes. In comparison, we calculated that on the same projects, DTDetector (using the pairwise testing heuristic) would impose on average a 2,276X slowdown when considering dependencies between test methods, or 279X between entire test classes (Table 3.3). ELECTRICTEST’s overhead fluctuates with heap access patterns — in test suites that share large amounts of heap data between tests, ELECTRICTEST is slower. The overhead also fluctuated somewhat with the average test method duration: since a complete garbage collection and heap walk had to occur after each test finishes, test suites consisting of a lot of very fast-executing tests (like in ‘mule’) had a greater slowdown.

We believe that ELECTRICTEST’s overhead makes it feasible to use in practice, and note that it is still much less than DTDetector’s, the previous system for detecting test dependencies [163].

3.3.3 Impact on Acceleration

Our approach may detect dependencies between tests that do not effect the outcome of tests. That is, two tests may have a data dependency, but this dependency may be completely benign to the control flow of the test.

Therefore, we take special care to evaluate the impact of dependencies detected by ELECTRICTEST on test acceleration techniques, notably, test parallelization. In the extreme, if ELECTRICTEST found data dependencies between every single test, techniques like test parallelization or test selection yield no benefits, since it would be impossible to change the order that tests ran in while preserving the dependencies.

We first evaluate the impact of ELECTRICTEST on test acceleration techniques by examining the longest dependency chain detected in each project, shown under the heading ‘C’ in Table 3.5. In almost all projects, even if there were many dependent tests, the longest critical path was very short compared to the total number of tests. For example, while 4,079 of the 5,603 test methods in the jetty test suite depended on some value from a previous test, the longest dependency chain was 424 methods long. Across all of the projects, the average maximum dependency chain between test methods was 976 of an average 4,069 test methods and 198 between an average of 960 test classes. We find this result encouraging, as it indicates that test selection techniques can still operate with some sensitivity while preserving detected dependencies.

To quantify the impact of ELECTRICTEST’s automatically detected dependencies on test parallelization we simulated the execution of each test suite running in parallel on a 32-core machine, distributing tests in a round-robin fashion in the same order they would typically run in. In this environment, there are 32 processes each running tests on the same machine, with each process persisting for the entire execution.

We simulated the parallelization of each test suite following three different configurations: without respecting dependencies (“unsound”), with a naive dependency-aware scheduler (“naive”), and the optimistic greedy scheduler described in §3.2.4 (“greedy”). The naive scheduler groups tests into chains to represent dependencies, such that each test is in exactly one group, and each group contains all dependencies for each test — this approach soundly respects dependencies but may not be optimal in execution time. Table 3.6 shows the results of this simulation, parallelizing at the granularities of test classes and test methods. We show the theoretical speedups for each schedule provided relative to the serial execution, where speedup is calculated as $T_{serial}/T_{parallel}$. Overall, the greedy optimistic scheduler outperformed the naive scheduler in some cases, and at times provided a speedup close to that of the unsound parallelization. In some cases, the dependency-preserving parallelization was faster when parallelizing at the coarser level of test classes. In these cases, there were so many dependencies at the test method level that the schedulers were generating incredibly inefficient schedules, requiring that some tests were re-executed many times. Also, this may have occurred in some cases because we assumed that the shortest amount of time that a single test could take was one millisecond: in the case of a single test class that took one millisecond that had several test methods, if we parallelized the test methods, we may assume a total time to execute

Table 3.6: **Relative speedups from parallelizing each app’s tests.** Shown at the test class (**C**) and test method (**M**) while respecting ELECTRICTEST-reported dependencies (with the naive scheduler and the greedy scheduler) in comparison to unsound parallelization without respecting dependencies.

Project	Naive ET		Greedy ET		Unsound	
	C	M	C	M	C	M
camel	4.6	6.5	6.9	9.8	16.1	18.0
crunch	4.8	3.0	7.8	3.1	12.4	15.5
hazelcast	1.2	2.1	1.2	2.6	12.9	20.0
jetty.project	6.1	6.9	6.1	6.9	9.5	17.0
mongo-java-driver	1.8	1.6	1.8	1.6	8.2	27.8
mule	9.6	7.9	9.6	13.8	17.0	18.0
netty	2.8	6.7	2.8	6.7	2.9	7.0
spring-data-mongodb	1.2	0.8	1.2	0.8	3.5	26.5
tachyon	3.9	4.3	3.9	15.5	5.3	26.9
titan	3.8	6.3	3.8	6.3	8.0	18.2
Average	4.0	5.0	5.0	7.0	10.0	19.0

longer than just running a test class at once.

We investigated the cases where ELECTRICTEST didn't do as well, 'spring-data-mongodb' and 'mongo-java-driver' — both projects had very long dependency chains. Upon inspection, we found that most tests in each project *purposely* shared state between test cases for performance reasons. For instance, the mongo driver created a single connection to a database and reused that connection between tests to save on setup. The spring based project had a similar pattern.

These cases bring up an interesting point: sometimes tests may be intentionally data-dependent on each other. Especially in the case of short unit tests all testing the same large functional component, it is reasonable to expect that developers would intentionally re-use state to reduce the overall testing time. Thanks to its integration with the JVM, ELECTRICTEST can easily be configured by developers to ignore particular dependencies at the level of static or instance fields.

3.3.4 Discussion and Limitations

There are several limitations to our approach and implementation. Because we detect dependencies of code running in the JVM, we may miss some dependencies that occur due to native code that is invoked by the test. While ELECTRICTEST can detect Java field accesses from native code (through the use of field watches), it can not detect file accesses or array accesses from native code. However, none of the applications that we studied contained native libraries. It would be possible to expand our implementation to detect and record these accesses by performing load-time patching for calls to JNI functions for array accesses and system calls for file access to record the event.

When we detect external dependencies (i.e., files or network hosts), we assume that there is no collusion between externalities. For example, we assume that if one test communicates with network host *A*, and another test communicates with network host *B*, hosts *A* and *B* have no backchannel that may cause a dependency between the two tests. We have not built our tool to handle specialized hardware devices (other than those accessed via files or network sockets) that may be involved in dependencies. However, ELECTRICTEST could easily be extended to handle such devices in the same manner as files and network hosts. Since ELECTRICTEST is a dynamic tool, it will only detect dependencies that occur during the specific execution that it is used for: if tests exhibit different dependencies due to nondeterminism in the test, the dependency may not be detected. ELECTRICTEST could be expanded to include a deterministic replay tool such as [24, 84]

to ensure that dependencies don't vary.

There are also several threats to the validity of our experiments. We studied the ten longest building open source projects that we could find, in conjunction with four relatively short-building projects used by other researchers. These projects may not necessarily have the same characteristics as those projects found in industry. However, we believe that they are sufficiently diverse to show a cross-section of software, and show that ELECTRICTEST works well with both long and short building software.

We simulated the speedups afforded by various parallel schedules of test suites. Due to resource limitations, we did not actually run the test suites in parallel. We assume that the running time of a test is constant, regardless of the order in which it is executed. Therefore we may expect that the speedup of the unsound parallelization is an over-estimate: if multiple tests share the same state to save on time running setup code, then it may actually take longer to run the tests in parallel since the setup must run multiple times. However, we are confident that the various speedups predicted for dependency-preserving schedules are sound, as we do not believe that other external factors are likely to impact the running time of each test.

Similarly, we did not directly study the impact of the dependencies we detected on test selection or prioritization techniques, instead using the maximum dependency size as a proxy for selectivity. A more thorough study may have instead downloaded many different versions of each program and performed test selection or prioritization on each version (based on results from the previous version) and then measured the impact of detected dependencies on these tools. Such a study also would show the practicality of caching ELECTRICTEST results throughout the development cycle, so it need not be executed for each build. This caching might significantly reduce the performance burden of checking for test dependencies with each successive change to the program. Again, we were limited in resources to perform such a study, and believe that our use of maximum dependency size as an indicator for selectivity is sufficient.

3.4 Related Work

Test dependencies are one root cause of the general problem of flaky tests, a term used to refer to tests whose outcome is non-deterministic with regards to the software under test [59, 95, 97]. Luo,

et al. analyzed bug reports and fixes in 51 projects, studying the causes and fixes of 161 flaky tests, categorizing 19 (12%) of these to be caused by test dependencies [95]. ELECTRICTEST could be used to automatically detect and avoid these dependency problems before they result in flaky tests.

ELECTRICTEST is most similar to Zhang et al.’s DTDetector system, which detected *manifest dependencies* between tests by running the tests in various orderings [163]. A manifest dependency is indicated by a test having a different outcome when it is executed in a different order relative to the entire test suite. This approach required $O(n!)$ test executions for n tests, with best-case approximation scenarios at $O(n^2)$. ELECTRICTEST instead detects data dependencies, where one test reads data that was last written by a previous test, and does not require running each test more than once, in a much more scalable approach.

Unlike ELECTRICTEST, which observes and reports actual data dependencies between tests, Gyori et al.’s PolDet tool detects potential data sharing between tests by searching for data “pollution” — data left behind by a test that a later test may read (which may or may not ever occur) [73]. PolDet captures the JVM heap to an XML file using Java reflection and compares these XML files offline, while ELECTRICTEST performs all analysis on live heaps, greatly simplifying detection of leaked data.

ELECTRICTEST’s technique for dependency detection is more related to work in Makefile (build) parallelization, such as EMake [110] or Metamorphosis [66]. These systems observe filesystem reads and writes for each step of the build process to detect dependencies between steps and infer which steps can be paralleled. In addition to filesystem accesses, ELECTRICTEST monitors memory and network accesses.

While ELECTRICTEST detects hidden dependencies between tests, there has also been work to efficiently isolate tests to ensure that dependencies do not occur. Popular Java testing platforms (e.g., JUnit [3] or TestNG [4] running with Ant [9] or Maven [11]) support optional test isolation by executing each test class in its own process, resulting in isolation at the expense of a high runtime overhead. Our previous work, VMVM (Chapter 2), eliminates in-memory dependencies between tests without requiring running each test in its own process, greatly reducing the overhead for isolation. While VMVM preserves the exact same semantics for isolation and initialization that would come by executing each test in its own process, other systems such as JCrasher [49] also isolate tests efficiently, although without reproducing the same exact semantics. If tests are already dependent

on each other, but the goal is to isolate them, then ELECTRICTEST could be used to identify which tests are currently dependent (and how), allowing a programmer to manually fix the tests so that they can run in isolation.

Other tools support test execution in the presence of test dependencies. However, all of these tools require developers to manually specify dependencies, a tedious and difficult process which is automated by ELECTRICTEST. For instance, both the depunit [2] and TestNG [4] framework allow developers to specify dependencies between tests, while JUnit [3] allows developers to specify the order to run tests.

Test Suite Minimization identifies tests cases that may be redundant in terms of coverage metrics and removes them from the suite. Many heuristics and coverage metrics have been proposed to minimize test suites, although most approaches are limited by the strength of the coverage criteria used [38,39,78,79,87,88,137,152]. Test selection approaches the same problem of having too many tests to run from a different angle by instead selecting only tests to run that have been impacted by changes in the application code since the last time that tests were executed [15, 30, 65, 70, 80]. Since test selection can be dangerous if additional tests are impacted by changes but not selected (i.e. due to imprecision in the coverage metrics used to determine impact), some may turn to test prioritization, where entire test suites are still executed, but tests most likely to be impacted by recent changes are executed first [53,55,119,134,151]. Haidry and Miller propose several test prioritization techniques that consider dependencies between tests when performing the minimization, but require developers to manually specify dependencies [74]. ELECTRICTEST could be combined with each of these techniques to efficiently and automatically detect dependencies between tests, then safely accelerate them using test selection or prioritization.

3.5 Conclusions

While testing dominates long build times, accelerating testing is tricky, since test dependencies pose a threat to test acceleration tools. Test dependencies can be difficult to detect by hand, and prior to ELECTRICTEST, there was no tool to practically detect them in all but the very smallest test suites (those which took less than several minutes to run normally). We have presented ELECTRICTEST, a tool for detecting data dependencies between Java tests with an average slowdown of only 20X,

where previous approaches would have been completely infeasible taking up to 10^{308} times longer to find all dependencies. We evaluated the accuracy of ELECTRICTEST, finding it to have perfect recall compared to the previous approach in our study. Because not all data dependencies will influence the control flow of the data-dependent tests, we evaluated the impact of ELECTRICTEST on test parallelization and selection, finding its dependency chains small enough to still allow for acceleration. The dependencies detected by ELECTRICTEST can further be used by developers to gain insight into how their tests interact and fix unintentional dependencies.

Chapter 4

Dynamic Data-flow Analysis in the JVM

Dynamic taint tracking is a form of information flow analysis that identifies relationships between data during program execution. Inputs to the application being studied are labeled with a marker (are “tainted”), and these markers propagated through data flow. Dynamic taint tracking can be used for detecting brittle tests [85], end user privacy testing [57, 132] and debugging [60, 92].

While the exact semantics for how labels are propagated may vary with the problem being solved, many parts of the analysis can be reused. Dytan [46] provides a generalized framework for implementing taint tracking analyses for x86 binaries, but can’t be easily leveraged in higher level languages, like those that run within the JVM. By operating within the JVM, taint tracking systems can leverage language semantics that greatly simplify memory organization (such as variables). However, in Java, associating metadata (such as tags) with arbitrary variables is very difficult: previous techniques have relied on customized JVMs or symbolic execution environments to maintain this mapping [34, 85, 103], limiting their portability and restricting their application to large and complex real-world software.

Without a performant, portable, and accurate tool for performing dynamic taint tracking in Java, testing research can be restricted. For instance, Huo and Clause’s *OraclePolish* tool uses the Java PathFinder (JPF) symbolic evaluation runtime to implement taint tracking to detect overly brittle test cases, and due to limitations in JPF, could only be used on 35% of the test cases studied. Other previous general purpose taint tracking systems for the JVM [34, 103] were implemented as modifications to research-oriented JVMs that do not support the full Java specification and are not practical for executing production code. While some portable taint tracking systems exist for the

JVM, they support tracking tags through Strings only [42, 76, 77], and can not be used to implement general taint tracking analyses, as they are unable to track data in any other form.

Our dynamic taint tracking system for Java, PHOSPHOR, efficiently tracks and propagates taint tags between all types of variables in off-the-shelf production JVMs such as Oracle’s HotSpot and OpenJDK’s IcedTea [19]. PHOSPHOR provides taint tracking within the Java Virtual Machine (JVM) without requiring any modifications to the language interpreter, VM, or operating system, and without requiring any access to source code. Moreover, PHOSPHOR can be applied to any commodity JVM, and functions with code written in any language targeting the JVM, such as Java and Scala.

PHOSPHOR’s approach to tracking variable level taint tags (without modifying the JVM) seems simple at first: we essentially need only instrument all code such that every variable maps to a “shadow” variable, which stores the taint tag for that variable. However, such changes are actually quite invasive, and become complicated as our modified Java code begins to interact with (non-modified) native libraries. In fact, we are unaware of any previous work that makes such invasive changes to the bytecode executed by the JVM: most previous taint tracking systems for the JVM use slower mechanisms to maintain this shadow data [149].

We evaluated PHOSPHOR on a variety of macro and micro benchmarks on several widely-used JVMs from Oracle and the OpenJDK project, finding its overhead to be impressively low: as low as 3.32%, on average 53.31% (and up to 220%) in macro benchmarks. We also compared PHOSPHOR to the popular, state of the art Android-only taint tracking system, TaintDroid [57], finding that our approach is far more portable, is more precise, and is comparable in performance.

The contributions of this chapter are:

- A general purpose approach to efficiently storing meta-data for variables in the JVM, without requiring any modifications to the JVM.
- A general purpose approach to propagating this shadow information in the form of taint tracking, again, without requiring any modifications to the JVM.
- A description of our open source implementation of this technique: PHOSPHOR (released on GitHub [16]).

4.1 Motivation

Although several existing systems target Java applications (e.g. [42, 76, 77]) by modifying application or library bytecode, these are not general purpose: they can only track data flow of Java Strings (and not of any other type), and therefore are unable to continue tracking those Strings in the event that they are converted by the application to another representation (such as a character array). Moreover, these systems can not track inputs that are not Strings (e.g. integers, or a language-specific version of String in another, non-Java JVM language).

Several existing systems can perform taint tracking on all data types in Java, but are highly restricted in portability, functioning only on research JVMs. The JVMs targeted by [103] (Kaffe [142]) and [34] (Jikes RVM [141]) support only a subset of Java version 6, severely limiting applicability. We will refer to both of these incomplete JVMs as “research JVMs,” as they do not implement the complete Java specification, and are principally used within the research community (rather than in production environments).

We also note that while we focus on dynamic taint tracking, static taint analysis is also a topic of interest. However, while static taint analysis for Java [12, 133, 144] can determine a priori where data might leak from a system, it may report false positives from code which can not execute in practice, and as with all static analysis tools for Java, it must model reflective calls, possibly further increasing the likelihood of false positives.

There is a need for a general purpose taint tracking system that is sufficiently decoupled from specific data types to support a wide range of precise and sound analyses (i.e. with no false positives or false negatives) for applications running on any production JVM. We briefly describe work in three broad areas that could benefit from PHOSPHOR.

4.1.1 Detecting injection attacks

Taint tracking has been widely studied as a mechanism for improving application security. Taint tracking can be used to ensure that untrusted inputs from external sources (such as an end-user) are not used as inputs to critical functions [77, 130, 135]. For instance, consider an application that takes an input string from the user, and then reads a file based on that input, returning the file to the user. An attacker could perhaps craft an input to coerce the application to read and return an arbitrary file,

including sensitive files such as `/etc/passwd`. Similar injection attacks can occur when calling external processes, or performing SQL queries. SQL injection attacks are the fifth most prevalent type of attack reported by CVE [50].

Taint tracking has been shown to be effective in detecting these sorts of attacks: all user input is tagged with a taint, and any function that may be an injection point is instrumented to first check its arguments to ensure that there are no taint tags. Trusted input sanitizers that sit between the user's input and the injection point can be used to allow sanitized inputs to flow to possible injection points (with the assumption that they will correctly sanitize the input).

4.1.2 Privacy testing and fine grained-access control

Taint tracking has also been successfully applied to fine-grained information access control [14, 37, 99, 122], and to end-user privacy testing [57]. In both cases, taint tracking is used to improve the granularity of existing mechanisms for enforcing rules about information flow. For access control, taint tracking is useful as it allows developers or system administrators to specify access rules based on data. For instance, administrators may wish to restrict the operations that users may perform on certain data, without a priori knowledge of where in the application's control flow that data may appear from. As another example, an application may include untrusted libraries during run time, and want to restrict those libraries from accessing sensitive data.

For end-user privacy testing, users specify system-wide taint sources (e.g. on a mobile device, GPS location, personal contacts, etc.), and destinations, where tainted data must never flow to (e.g. system-level functions that send data over the network). In this way, users can determine if their private information is being transferred to remote servers.

Note that both of these applications of taint tracking demand a system that is both performant and portable. For example, an end-user may wish to observe the privacy violations of an application, without the prior planning of the application developers to support taint-tracking, and without requiring specialized hardware or a specialized operating system. Both systems would be challenging to implement in the JVM without a taint tracking system.

4.1.3 Testing and Debugging

Taint tracking has also been employed to improve the testing and debugging process. For instance, taint tracking can be used to increase test coverage when using automated input generators [92]. In this application, the taint tracking system labels each input, and at each conditional branch, records what label (or set of labels) the jump condition had. This information is then fed back to the input generator to focus input generation on those that are known to be restricting control flow. This approach can also be useful for debugging program failure by using taint tracking to identify which inputs were relevant to the crash [60].

4.2 Approach

In designing PHOSPHOR, our primary goal was to enable studies and analyses of dynamic data flow in languages that target the JVM, such as Java, Scala and Clojure. While some of these analyses may be targeted towards researchers running experiments in closed environments (in which case, run time overhead and portability are unlikely to be significant concerns), others may target actual use by end-users (e.g. the privacy study performed in [57]). Hence, a key goal for PHOSPHOR was to ensure that it has both relatively low run time overhead and was portable (i.e. could be used on a variety of JVMs and platforms).

In general, common challenges to building taint tracking systems in support of such analyses include:

1. **Soundness:** When working with native binaries, it can be difficult or impossible to determine the correct level of granularity to assign distinct taint tags. Should each byte be distinctly tagged? Each word? These questions are difficult if not impossible to answer in the general case, and can directly impact the soundness of the tool. If a tool is not sound, then it may incorrectly drop taint information from variables.
2. **Precision:** In the process of improving soundness of a taint tracking system, systems often trade higher accuracy for lower precision, leading to over tainting, where taint tags are propagated between values even when there is no actual connection between them. In some cases, over tainting can lead to significant decreases in precision, with values marked by the wrong

tag. If a tool is not precise, it may incorrectly add additional taint information to variables.

3. **Portability:** Most taint tracking systems require access to application source code [91, 156], require modified operating systems [146, 160] or modified language interpreters [14, 34, 72, 103, 105].
4. **Performance:** Taint tracking often adds a very high performance overhead (commonly showing slowdowns of 1x-30x depending on the tool and benchmark), limiting its use in deployment environments.

Our approach to taint tracking uses *variable-level* tracking, inspired by previous work that modified the interpreter to support taint-tracking in Java [34, 57, 103]. A key observation is that when operating within the JVM (e.g. in Java, Scala and others), we can bypass the common challenges related to accuracy and precision: variables are clear units of data, and because code can not access arbitrary memory addresses, we can be certain that if we associate a taint tag with a variable, any access to that variable can be mapped to the taint tag. Therefore, this design choice can eliminate some difficulties associated with maintaining precision in taint tracking that typically affect systems operating at a lower level (e.g. at the OS level [146, 160], or via binary instrumentation [40, 46]).

Most taint tracking systems for other memory managed languages (e.g. targeting JavaScript [72], php [130, 157], Dalvik [57], Java [34, 103] and others), rely on modifications or extensions to the interpreter, which allows taint tracking code access to significantly lower level memory operations than taint tracking code running within a managed environment like the JVM. However, in order to ensure portability, we designed PHOSPHOR to run entirely within the confines of an unmodified JVM.

The decision to run within the confines of code executing in the JVM (and not inside of the JVM's interpreter) raises several unique challenges because our taint tracking instrumentation is subject to the same memory management restrictions that any other code is. The prime challenge in creating PHOSPHOR (and our key contribution), therefore, is to efficiently maintain a mapping from values to taint tags within the confines of a memory-managed environment.

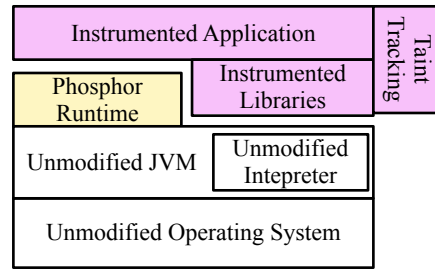


Fig. 4.1: The high level architecture of PHOSPHOR

4.2.1 JVM Background

Before describing how PHOSPHOR works, we first provide a brief background on data organization within the JVM (based on the JVM specification, version 7 [94]).

There are eight “primitive” types supported by the JVM, all of which are stored and passed by value: boolean, byte, character, integer, short, long, float, and double. In addition to primitive types, the JVM supports two reference types: objects and arrays. Objects are instances of classes, which may contain fields (which are members of each instance) and static fields (which are members of each class). Arrays can be declared to store either reference types (which would include other arrays) or primitive types. Reference types can be cast to a super type, which affects what operations are available on that instance of that type, and are all sub-types of the root type, `java/lang/Object`.

The JVM is a stack machine, with stack memory split into two components: the operand stack and the local variable area. The operand stack is used for passing operands to instructions and can only be manipulated with stack operators, while the local variable area is indexed. Method arguments are passed by placing them on the operand stack, and are accessed by the receiver as local variables. The combination of the operand stack and local variable area make up a JVM frame. When a method is invoked, a new frame is created for that method, and when it returns, the frame is destroyed. It is impossible for code can to access any frame other than the current frame.

4.2.2 High Level Design

Figure 4.1 shows a high level overview of our approach to portable taint tracking with PHOSPHOR: we modify all bytecode running within the JVM, and then run that code in a completely unmodified JVM, running on an unmodified operating system, with commodity hardware.

```

1 // Original Code
2 int foo(int in){
3   int ret = in + val;
4
5   return ret;
6 }

```

(a) The original class

```

1 //With Int Tag Tainting
2 TaintedIntWithIntTag doMath$PHOSPHOR(
    int in_tag, int in){
3   int ret = in + val;
4   int ret_tag = in_tag | val_tag;
5   return TaintedIntWithIntTag.
6     valueOf(ret_tag, ret);
7 }

```

(b) The modified class, ready to track taint tags

Fig. 4.2: A basic example of the sort of transformations that PHOSPHOR applies at the bytecode level to support taint tracking. Underlined lines call out to changes made by PHOSPHOR. Example shown at the source level, for easier reading.

PHOSPHOR’s taint tracking is based on variable-level tracking, storing a tag for every variable. When operations are performed on these variables, PHOSPHOR combines their taint tags to create the new tag for the resulting combination.

PHOSPHOR modifies bytecode to include storage for taint tags and to include instructions to propagate these tags. We use the ASM [31] bytecode manipulation library to insert our instrumentation and support all recent versions of the Java bytecode specification (up to version 8). This instrumentation normally occurs offline (before execution) but in the event that a class is defined at run time (and hence, wasn’t instrumented), PHOSPHOR intercepts all classes as they are loaded, ensuring that every single class is instrumented. The instrumentation process is performed only once per class and is relatively quick, requiring only 1.4 minutes to instrument the entire Java 7 JRE (approximately 19,000 classes). Figure 4.2 shows an example of the sorts of transformations that are applied to bytecode. Note that our example is shown as Java source code for ease of understanding, but in reality, all transformations occur at the level of Java bytecode.

At the high level, PHOSPHOR adds a field to every Class to track the tag of instances of that Class, and adds a shadow variable for every variable (be they local variables, method arguments, or fields) that is not an instance of a Class to track that variable’s tag. When it’s impossible to add such a shadow variable (e.g. to pass the tag of a primitive return value from a method), PHOSPHOR combines the taint tag with the value into a *container class*, which encapsulates both the tag and

the value into one reference (which is then the return value). Formally, PHOSPHOR consults the following five properties to determine how to store or retrieve the taint tag for a variable:

Property 4.2.1. Let R be a reference to an instance of an Object. Then the taint tag of R is stored as a component of the object to which R points.

Property 4.2.2. Let A be a reference to an array of references. Then the taint tag of array element $A[i]$ is stored as a component of the object to which $A[i]$ points.

Property 4.2.3. Let V be a primitive value. Then the taint tag of V is stored as a shadow value next to V .

Property 4.2.4. Let A be a primitive array reference. Then a shadow array A_s is stored next to A , and the taint tag of primitive value $A[i]$ is $A_s[i]$.

Property 4.2.5. Let A be a primitive array reference and A_s be the reference to its shadow array. If A is stored as the type `Object`, then A and A_s are first boxed into a container, as $C(A, A_s)$.

Note that by these properties, every single variable has its own distinct taint tag: each element in an array is tracked distinctly (unlike in other taint tracking systems, such as [34,57], which sacrifice this precision for added performance by storing only a single taint tag for all of the elements in an array). The implementation and rationale behind each of these properties is described in much greater detail in §4.3.1.

These properties are also enforced when programs dynamically access fields and invoke methods via Java’s reflection interface, which we patch to propagate taint tags.

PHOSPHOR can automatically apply a taint tag to variables that are returned from pre-defined taint “source” methods (for instance, methods that take user input). When applying taint-tracking transformations to bytecode, PHOSPHOR consults a configuration file for a list of methods that should result in their return value (or arguments) being tainted. PHOSPHOR also consults the same configuration file for a list of methods that should check their arguments to determine if any of them are tainted (a “taint sink,” for instance, a method that executes a SQL command), logging the occurrence or raising an exception in the case that they are tainted.

For more complicated semantics to mark variables with taint tags and respond to variables that are marked, PHOSPHOR provides a simple API, exposing the simple functions `setTaint` and

`getTaint`, which respectively set the taint tag of a variable and retrieve the taint tag of a variable. These functions are useful for implementers of analyses that build upon PHOSPHOR, and are not intended to need to be inserted into any target application code directly.

PHOSPHOR represents the taint of a variable as a 32-bit long bit vector, allowing for a total of 32 distinct taints (similar to other systems, such as TaintDroid [57]). When taint tags are combined, they are bit-wise OR'ed. Alternatively, a developer could specify more complex logic for generating and combining taint tags, allowing for 2^{32} possible taint tags, although with perhaps greater overhead (an evaluation which we leave for future work and consider out of scope). We also have anecdotal evidence showing that PHOSPHOR can use any arbitrary type (e.g., objects) to represent the taint tag of a variable — not just a primitive number.

4.2.3 Approach Limitations

There are several limitations to our approach. As PHOSPHOR functions within the confines of the JVM, it is unable to track data flow through native code executing outside of but interacting with the JVM. We have implemented the current best-practices for handling such flows (i.e., assuming that all native code propagates taints from all inputs to all outputs), discussed further in §4.3.3. Next, since our approach requires modifying the bytecode of applications, this could modify the behavior of applications that somehow use that bytecode as an input, since the bytecode will have been modified by PHOSPHOR to include taint propagation instructions. Typically in Java, such inspection is done using the *Reflection* interface, which our implementation patches to hide all traces of PHOSPHOR. PHOSPHOR works with applications that use Java's *Reflection* to read their bytecode, but does not work with applications that use other, non-standard approaches to read their code.

4.3 Implementation

PHOSPHOR consists of an instrumenter that modifies each Java class (either offline, or dynamically at load-time by intercepting classes as they are loaded) to add additional variables and instructions to perform taint tracking, and a small runtime library. The runtime library is very small, and consists only of several helper methods used for ensuring that taint tags are tracked through calls to Java's

reflection interface. There are no central data structures that store taint tags: as shown in Figure 4.2(b), taint tags are stored in variables adjacent to the variables that they are tracking. This lack of centralized structure allows PHOSPHOR to be both performant and thread-safe.

4.3.1 Taint Tag Storage

Based on the discussion above of memory organization within the JVM, we consider shadow variable storage (for taint tags) in four different areas: as fields, as local variables, on the operand stack, and as method return values. Moreover, based on the discussion of types in the JVM, we consider five broad categories of variables for which we may need different taint tag representations: primitives, primitive arrays, multi-dimensional primitive arrays, arrays of other references, and general references. For each of these types, we will enumerate rules for their taint tag storage.

4.3.1.1 Reference Types

PHOSPHOR stores one taint tag per-variable, so there is no tag stored for each reference to a variable: the taint tag of a reference is simply the tag of the value that it points to. Storing the taint tag for references that point to instances of classes (i.e. objects) is straightforward: PHOSPHOR adds a new field to that type, such that each instance of the class has an extra field in which we can store the taint tag. This model extends to support arrays of reference types, since the taint tag of each reference type in the array is stored directly as part of the reference type. From these two observations, we can derive Properties 4.2.1 and 4.2.2.

However, there are reference types for which PHOSPHOR can not add an extra field to track the taint tag of that type: notably, primitive multi-dimensional arrays. Recall that primitive arrays are reference types, so a multi-dimensional primitive array must be an array of reference types. Since arrays are not objects, we can not simply add a field to that type: instead, we create a new class to *box* the primitive array and its taint tag into a single type. For example, an N -dimension array `char[][][]`, will be mapped to an $(N - 1)$ -dimension array of `MultiDimensionCharArray[]`, where `MultiDimensionCharArray` is a class that has two fields: a `char[]` field to store the value of the final dimension of the array, and an `int[]` field to store its taint tags. All references to multi-dimension primitive arrays are remapped to access the array through the container, ensuring that Property 4.2.2 continues to hold.

4.3.1.2 Primitives and Primitive Arrays

For variables that are primitives (or primitive arrays), we cannot simply add an extra field to the type to store the tag, since there is no structure exposed within the JVM that represents these types that we could modify. Instead, PHOSPHOR stores the taint tag (or a reference to the taint tag) in a shadow, alongside the actual value (Properties 4.2.3 and 4.2.4). This subsection will describe exactly where that shadow is stored.

For variables that are stored as fields in a class, PHOSPHOR creates a shadow field to store the taint tag for that element. For instance, if a class has a member `private int val`, then PHOSPHOR adds another field: `private int val.tag`.

To support primitive values and primitive arrays as local variables, PHOSPHOR creates an additional local variable to store the taint tag, for each local variable that represents a primitive or primitive array. Primitive and primitive array method arguments are supported similarly to local variables: we create shadow arguments to track the taint tag for each primitive and primitive array argument.

Primitive and primitive array return types are supported by boxing the value and its taint tag into a container just before return. PHOSPHOR changes the return type of all such methods to be the appropriate container, and modifies the return instruction to first construct the container, and then return it (instead of just returning the primitive value or primitive array reference). Just after the call site to a method that returns a container type, the container is unboxed, leaving the primitive return value on the stack, with the taint tag just below it. To reduce overhead, each method pre-allocates containers at its entry point for the methods that it will call, passing these containers to each method called. In this way, if a method makes several calls to another method which returns a primitive value, only one container is allocated, and is re-used for each call.

To support primitive values and primitive arrays on the operand stack, PHOSPHOR instruments every stack operator to ensure that before any primitive value or primitive array reference is pushed onto the stack its taint tag is pushed as well, and just after a primitive value or primitive array reference is popped, its taint tag is as well.

PHOSPHOR creates these extra fields and variables as necessary based on the type information for the field or variable. However, note that because primitive arrays are reference types, they are assignable to fields and variables with the generic type `Object` (for which PHOSPHOR would not

```
1 public String leakString(String in){
2     String r = "";
3     for(int i = 0; i < in.length; i++)
4     {
5         switch(in.charAt(i)){
6             case 'a':
7                 r+="a";
8                 break;
9             ...
10            case 'z':
11                r+="z";
12                break;
13        }
14    }
15    return r;
16 }
```

Fig. 4.3: Simple code showing the inadequacy of data flow tag propagation: the output will have no taint tag, even if the input did. Control flow propagation, however, will propagate these tags.

have a priori created a shadow variable). PHOSPHOR accounts for this situation by automatically boxing primitive arrays with their taint tags before assigning them to the generic type `Object`, and by automatically unboxing them when casting from the generic type `Object` back to a primitive array.

4.3.2 Propagating Taint Tags

The remainder of this section will describe the specific changes made to application and library bytecode to propagate taint tags. A complete listing of all bytecodes available and the modifications that PHOSPHOR makes is available in the appendix to this thesis, in Table 1.

PHOSPHOR can combine taint tags in one of two different ways. In traditional tainting mode, taint tags are 32-bit integers which are combined through bit-wise OR'ing, allowing for a maximum of 32 distinct tags, with fast propagation. In multi-taint mode, taint tags are objects which contain a list of all other tags from which that tag was derived, allowing for an arbitrary number of objects and relationships.

Like most taint tracking systems, PHOSPHOR propagates taint tags through data flow operations

(e.g. assignment, arithmetic operators, etc.). However, depending on the goals of the analysis, data flow tracking may be insufficient to capture all relationships between variables. Figure 4.3 shows an example of a short code snippet will return a string identical to the input, but without a taint tag (if tags are tracked only through data flow operations), since there is no data flow relationship between the input and output.

PHOSPHOR optionally propagates taint tags through control flow dependencies as well (“implicit flow”), which would be necessary in the case of the code in Figure 4.3 to propagate tags through the method. Our implementation of control flow dependency tracking mirrors that of prior work from Clause et al [46], and leverages a static post-dominator analysis (performed as part of PHOSPHOR’s instrumentation) to identify which regions of each method are effected by each branch. Each method is modified to pass and accept an additional parameter that represents the control flow dependencies of the program to the point of that method. Within the method execution, PHOSPHOR tracks a stack of dependencies, with one entry for each branch condition that is currently influencing execution. When a given branch no longer controls execution (e.g. at the point where both sides of the branch merge), that taint tag is popped from the control flow stack. Before any assignment, PHOSPHOR inserts code to generate a new tag for that variable by merging the current control flow tags with any existing tags on the variable.

Method and Field Declarations: PHOSPHOR rewrites all method declarations to include taint tags for each primitive or primitive array, and to change all primitive and primitive array return types to be container types, which include the taint tag on the primitive value in addition to the actual value. All references to multi-dimension primitive arrays (in both fields and method descriptors) are replaced with container types. PHOSPHOR adds a new instance field to every class, used to track the taint tag of that instance. Finally, for every field that is a primitive or primitive array, PHOSPHOR adds an additional field that stores the taint of that primitive or primitive array.

Array Instructions: For all array load or store instructions, PHOSPHOR must remove the taint tag of the array index from the operand stack before the instruction is executed. For stores to primitive arrays, PHOSPHOR inserts instructions to also store the taint tag of the value being stored into the taint array. For loads from primitive arrays, PHOSPHOR similarly inserts instructions to load the taint tag from the taint array. For stores to reference type arrays, if the item being stored is a primitive array, PHOSPHOR inserts code to box the array and tag into a container before storing it.

PHOSPHOR instruments instructions that create new one-dimension primitive arrays with additional instructions to also create a taint tag array with the same length. For instructions that create multi-dimension primitive arrays, PHOSPHOR modifies them to instead create arrays of our containers (as discussed in §4.3.1.1).

The last array instruction that PHOSPHOR instruments is `ARRAYLENGTH`, which pops an array off of the operand stack and pushes onto the stack the length of that array. For this instruction, PHOSPHOR adds instructions to pop the taint array from the stack (if the array is a primitive array), and to add an empty taint (i.e. 0) to the returned value (we consider array length to be a control flow operation, and do not propagate any array taints into the taint of the length of each array).

Local Variable Instructions: PHOSPHOR adds an instruction to store a variable's taint tag immediately after each instruction that stores a primitive or primitive array variable. Similarly, for instructions that store object references to local variables, if the variable type is a primitive array, PHOSPHOR also stores the taint tag array for that variable. If the variable type is not a primitive array (i.e. *Object*), but the item being stored is a primitive array, then PHOSPHOR inserts instructions to first box the array into a container, before storing the array. For instructions that load local variables onto the operand stack, if the variable is a primitive or primitive array, then just before the variable is loaded, PHOSPHOR loads the pre-existing shadow variable (containing the taint tag) onto the stack.

Method Calls: PHOSPHOR instruments every method call, first modifying the method descriptor (i.e. the arguments and return type) to pass taint tags. Next, PHOSPHOR ensures that for every parameter of the generic type `Object`, if the parameter being passed is a primitive array, its taint array is boxed with it into a container. If the method is an instance method (i.e. has a receiver instance), PHOSPHOR ensures that if the receiver is a primitive array, its taint tag is dropped from the operand stack before the call. Immediately after the method call, if its return type had been changed to a container type, instructions are inserted to unbox the container, placing on the top of the stack the return value followed by the taint tag.

Method Returns: PHOSPHOR ensures that all return instructions that would otherwise return a primitive value or reference to a primitive array first box the primitive or primitive array with its taint tag(s) before returning.

Arithmetic Instructions: For arithmetic operators that take two operands (e.g. addition, subtraction, multiplication, etc), each operator expects that the top two values on the stack are the

operands, yet with PHOSPHOR, the top value will be the first operand, while the second will be the taint tag of the first operand, and the third the second operand, with the fourth its taint tag (as shown in Figure 4.4). PHOSPHOR prepends each arithmetic operator with instructions to combine the two taint tags (by bitwise ORing them), placing the new taint tag under the two (intended) operands, allowing the arithmetic to complete successfully.

Type Instructions: The JVM provides the `instanceof` instruction, which pops an object reference off of the stack and returns an integer indicating if that reference is an instance of a specified type. For this instruction, PHOSPHOR inserts a null taint tag (i.e. “0”) under the return value of the instruction (similar to array length, we consider this to be a control flow operation). Additionally, if the reference type on the operand stack is a primitive array, then its taint tag array is dropped from the stack. If the type argument to `instanceof` is a multidimensional primitive array, then PHOSPHOR changes the argument to instead refer to the appropriate container type (since again, we have eliminated multidimensional primitive arrays).

The other type instruction that PHOSPHOR instruments is the `checkcast` instruction, which ensures that the object reference at the top of the stack is an instance of a specified type, throwing an exception if not. PHOSPHOR rewrites this instruction to be aware of our boxed container types: if the cast is to a one-dimension primitive array type and the operand is a container, PHOSPHOR first unboxes the array and its taint tag array. If the cast is to a multi-dimension primitive array, then PHOSPHOR changes the type cast to be to the appropriate container type (since PHOSPHOR eliminates multi-dimensional primitive arrays), leaving it boxed.

Stack Manipulators: There are several instructions that directly manipulate the order of elements on the operand stack, for instance, swapping the top two values. In all cases, PHOSPHOR modifies each instruction based on the contents of the operand stack just before execution. For

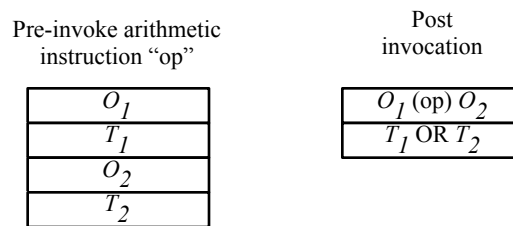


Fig. 4.4: Operand stack before and after performing two-operand arithmetic. The actual operands are shown as O , and their taint tags as T .

instance, if an instruction will swap the top two elements on the stack, and the top element is a primitive value (with a taint tag stored beneath it), but the element below that is an object reference (hence, with no taint tag beneath it on the stack), PHOSPHOR removes the swap instruction, replacing it with instructions to place the top two elements beneath the third.

Locking Instructions: There are two instructions in Java bytecode related to locking, one to procure a lock on an object reference, and one to release a lock already held on an object reference. In both cases, PHOSPHOR checks the top stack value, and if it is a one dimensional primitive array (which implies that there is a taint tag array on the stack beneath it), PHOSPHOR pops the taint tag array after the lock is acquired or released.

Jump Instructions: The JVM provides several jump instructions, jumping on either one or two object references or primitive values. For those that jump based on primitive values, in all cases PHOSPHOR first removes the taint tag from the value(s) being checked before the jump. For those that jump based on object references, PHOSPHOR removes the taint array tag, if the value(s) being checked before the jump are references to one dimensional primitive arrays.

4.3.3 Native Code and Reflection

As PHOSPHOR is implemented within the JVM, it is restricted from propagating taint tags in code that executes outside of the JVM. The JVM allows for “native” methods, which are implemented in native machine code, and can be called by normal code running inside of the JVM. We follow the same approach used by TaintDroid [57] for patching taint flow through these methods: we surround each with a wrapper that can propagate taint tags from the arguments of the method into the return value. As with TaintDroid, our implementation currently assigns the taint tag of the return type to be the union of the taint tags of all primitive, primitive array and String parameters. The wrapper is also necessary to wrap and unwrap values from their container types. For example, if a native method returns a primitive integer, the calling code will expect that the return value will actually be a `BoxedTaintedInteger` (rather than the primitive integer that it would normally return).

Java supports reflection, a feature that allows code to dynamically access and invoke classes and methods. PHOSPHOR patches all reflective calls to propagate taint tags as necessary, following the exact same semantics used for regular method calls and field accesses. PHOSPHOR also patches calls that inspect the fields and methods that exist in classes to hide any artifacts of the taint tracking

```

1 public static Integer valueOf(int i) {
2     assert IntegerCache.high >= 127;
3     if (i >= IntegerCache.low && i <= IntegerCache.high)
4         return IntegerCache.cache[i + (-IntegerCache.low)];
5     return new Integer(i);
6 }

```

Fig. 4.5: Java’s `Integer.valueOf` method, a very commonly used method with an indirect data flow caused by caching. If the input is between `IntegerCache.low` and `IntegerCache.high`, the output will have no taint tag, even if the input did. PHOSPHOR uses a special case to patch it.

process, removing additional fields and arguments as applicable.

4.3.4 Java-Specific Features

While our taint tracking process is generic to any language running in the JVM, we found that its support of Java could be significantly enhanced with several optimizations and modifications. For instance, both JVMs that we evaluated (OpenJDK and Oracle’s HotSpot JVM) make implicit assumptions about the internal structure of several classes (notably the super-type: `java.lang.Object`, and several of the classes internally used as containers for primitive types: `java.lang.Character`, `java.lang.Byte`, `java.lang.Boolean`, and `java.lang.Short`), which would prevent PHOSPHOR from adding taint storage fields to these classes. PHOSPHOR does not track taint tags on raw instances of the class `java.lang.Object`, which has no fields itself, and therefore, we do not believe is relevant in data flow analyses. For the four restricted primitive container types, PHOSPHOR instead stores the taint tag for instances of these types in a `HashMap` (similar to the technique used by [149]), hence avoiding the need to modify the internal structure of the class. Storing taint tags in a `HashMap` is much slower than as individual variables (when using it to store all taint tags, [149] showed a slowdown of up to 526x).

We also make a small modification to support a very commonly used indirect data flow in Java. Primitive container types can be very frequently used in Java, and are used within the JVM when necessary to represent a primitive value as an instance of a reference type. For efficiency, for each primitive type there is a cache of instances of the container class for all low values of that type. Listing 4.5 reproduces the code used to fetch an instance of class `Integer`. Due to the implicit

flow in lines 3-4, if an integer is found in the cache, then its taint tag is dropped. If the integer does not exist in the cache, then the taint tag will be propagated into the new instance of `Integer` in line 5. PHOSPHOR modifies the code that calls the `valueOf` method for each of the primitive container types to ensure that if the primitive argument has a non-zero taint tag, a new instance of the container is created with the tag, hence continuing to propagate taints.

4.3.5 Optimizations

The entire instrumentation process is implemented in a stream-processing manner: for each byte-code instruction, PHOSPHOR outputs new instructions, without context of instructions that previously were output, or those that will be output next. After the instrumentation process, we add several short optimization passes to provide a small amount of context to PHOSPHOR, greatly reducing the size of outputted methods.

First, PHOSPHOR detects instances where taint tags may be loaded to the stack, then immediately popped: for instance, variables loaded to the operand stack and used as operands for jump conditions. PHOSPHOR simply ignores loading the taint tags in these places.

Next, PHOSPHOR detects large methods that perform no instructions other than to load constants into arrays. Rather than initialize the taint tag for each constant as each constant is loaded, PHOSPHOR instead reasons that all tags will be 0, and can instead rapidly initialize them all at once, rather than initializing them one-by-one. This optimization was necessary in several cases in order to ensure that the generated methods remained within the maximum method size (64 kilobytes; this limitation is based on the size of the JVM's internal program counter).

Finally, after all instrumentation has been completed, PHOSPHOR scans each generated method for simplifications. For example, given our rules outlined in the previous section, for any method that returns a primitive value, instructions are inserted after its call site to unbox the taint tag and return value from the return container. However, if both of those values will be immediately discarded from the stack (i.e. `pop`'ed), then we can simplify the instructions that load and then discard the return value and return taint tag to simply not load the value or tag.

To some extent, these optimizations can also be achieved by the JIT compiler as it compiles the bytecode, but we have found that performing them in advance still improves run time (and in some cases, is necessary to ensure that the generated code fits within the maximum method size).

4.3.6 Application to Android and Dalvik

Although we designed PHOSPHOR for the JVM, we recognized that it could also be applicable to the language virtual machine used by Android, the Dalvik Virtual Machine (DVM). Nearly all applications for Android devices are written in Java, which is then compiled to Java bytecode and translated into the DVM's form of bytecode, called dex.

Because it executes a translated form of Java bytecode, and PHOSPHOR operates at the bytecode level, we can apply PHOSPHOR to Android and the DVM by inserting taint propagation logic in the intermediate Java bytecode before it is translated to dex. PHOSPHOR could even be applied without needing this intermediate Java bytecode, by using a tool such as [51], which translates dex bytecode back into Java bytecode. Note that although it runs a translated form of Java bytecode, the DVM should not be confused with a JVM; our primary target remains the JVM, and any modifications to the DVM or access to intermediate compiled code described in this subsection are unnecessary for JVM taint tracking.

There are many optimizations that the DVM performs beyond those of the JVM, perhaps due to the tight vertical integration of Android devices (from operating system to interpreter to language to APIs and applications). Several of these optimizations pose significant challenges for PHOSPHOR, as they significantly increase coupling between the interpreter and other classes, beyond those discussed in §4.3.4. Notably, the DVM provides very efficient native implementations of the `java.lang.String` methods `charAt`, `compareTo`, `equals`, `fastIndexOf`, `isEmpty` and `length`. These implementations rely on compile-time knowledge of the run-time organization of the class `java.lang.String` (i.e. the byte-level offsets of each field). Further, the DVM assumes in several cases that all internal primitive container types (not just the several assumed by the JVMs evaluated) contain only a single field containing the primitive value, and no other fields. While we could in principle support taint tracking instances of these classes by storing their taint tag in a `HashMap` (as for the several classes similarly restricted in the JVMs evaluated), doing so for all of the tightly coupled classes would have posed a prohibitive overhead.

Instead, we made several very small modifications to the Dalvik VM to decouple the VM from the implementation of these classes. Note that although we chose to modify the DVM in this case, the number of changes is significantly smaller than those necessary for TaintDroid, as we are not modifying the interpreter to perform taint tracking, but only to decouple it. These changes required

modifying seven constants defined in header files, and modifying six lines of native code that handle reflection. In comparison, the most recent version of TaintDroid (4.3.1) contains a total of over 32,000 lines of new code in the Dalvik VM (as reported by executing a diff of the repository), of which over 18,000 are in assembly code files, and 10,661 in C source code files.

4.3.7 General Usage

PHOSPHOR is available for download (both source code and pre-compiled binaries) on GitHub, at <https://github.com/Programming-Systems-Lab/phosphor>. The GitHub page contains further links to the artifact that passed the OOPSLA 2014 artifact evaluation process, which consists of a VirtualBox VM image that contains all of the java experiments performed in the original OOPSLA 2014 paper on PHOSPHOR.

We describe here a brief getting started guide (which is also available on the PHOSPHOR website), as well as a listing of the options available when using PHOSPHOR and the key API methods exposed by PHOSPHOR.

4.3.7.1 Getting Started

PHOSPHOR works by modifying your application's bytecode to perform data and control flow tracking. To be complete, PHOSPHOR also modifies the bytecode of JRE-provided classes, too. The first step to using PHOSPHOR is generating an instrumented version of your runtime environment. We have tested PHOSPHOR with versions 7 and 8 of both Oracle's HotSpot JVM and OpenJDK's IcedTea JVM.

We'll assume that in all of the code examples below, we're in the same directory (which has a copy of `phosphor.jar`), and that the JRE is located here: `UNINST_JAVA` (modify this path in the commands below to match your environment).

Then, to instrument the JRE we'll run: `java -jar phosphor.jar UNINST_JAVA jre-inst.`

The instrumenter takes two primary arguments: first a path containing the classes to instrument, and then a destination for the instrumented classes. Full use including all options is detailed in Figure 4.6.

After instrumenting the JRE, make sure to `chmod +x` the binaries in the new folder, e.g. `chmod +x jre-inst/bin/*`.

The next step is to instrument the code which you would like to track. We'll start off by instrumenting the demo suite provided under the PhosphorTests project. This suite includes a slightly modified version of DroidBench, a test suite that simulates application data leaks (modified to remove Android-specific tests that are not applicable to a desktop JVM). We'll instrument the phosphortests.jar file: `java -jar phosphor.jar phosphortests.jar inst`.

This will create the folder `inst`, and place in it the instrumented version of the demo suite jar.

We can now run the instrumented demo suite using our instrumented JRE using the command:

```
jre-inst/bin/java -Xbootclasspath/a:phosphor.jar  
-cp inst/phosphortests.jar -ea phosphor.test.Droid  
BenchTest. The result should be a list of test cases, with assertion errors for each "testImplicit-  
Flow" test case (assuming you did not enable control flow tracking).
```

4.3.7.2 Interacting with Phosphor

PHOSPHOR exposes a simple API to allow marking data with tags, and to retrieve those tags, shown in Figure 4.7. Key functionality is implemented in two different classes, one for interacting with integer taint tags, and one for interacting with object tags (used for the multi-taint mode). To get or set the taint tag of a primitive type, developers call the `taintedX` or `getTaint(X)` method (replacing `X` with each of the primitive types). To get or set the taint tag of an object, developers first cast that object to the interface `TaintedWithIntTag` or `TaintedWithObjTag` (PHOSPHOR changes all classes to implement this interface), and use the `get` and `set` methods.

In the case of integer tags, developers can determine if a variable is derived from a particular tainted source by checking the bit mask of that variable's tag (since tags are combined by bitwise OR'ing them). In the case of multi-tainting, developers can determine if a variable is derived from a particular tainted source by examining the dependencies of that variable's tag.

4.3.7.3 Extending Phosphor

We have released PHOSPHOR under an MIT license, and encourage its use and extensions of it. We would very much welcome any feedback regarding PHOSPHOR.

```
[frame=single]
Usage: java -jar phosphor.jar [OPTIONS] [in] [out]
  -controlTrack          Enable taint tracking
                        through control flow
  -help                  print this message
  -multiTaint            Support 2^32 tags
                        instead of just 32
  -taintSinks <taintsinks> File with listing of
                        taint sinks to use to
                        check for auto-taints
  -taintSources <taintSources> File with listing of
                        sources to auto-taint
  -withoutDataTrack      Disable taint
                        tracking through data
                        flow (on by default)
```

Fig. 4.6: Arguments accepted by PHOSPHOR

```
//Integer-taint related API
//Class: edu.columbia.cs.psl.phosphor.runtime.Tainter
int getTaint(<primitive type>);
<primitive type> taintedPrimitiveType(<primitive type> val , int tag);
//Interface: edu.columbia.cs.psl.phosphor.struct.TaintedWithIntTag
int getPHOSPHOR_TAG();
void setPHOSPHOR_TAG(int tag);

//Multi-taint related API
//Class: edu.columbia.cs.psl.phosphor.runtime.MultiTainter
Taint getTaint(<primitive type>);
<primitive type> taintedPrimitiveType(<primitive type> val , Object tag);
//Interface: edu.columbia.cs.psl.phosphor.struct.TaintedWithObjTag
Taint getPHOSPHOR_TAG();
void setPHOSPHOR_TAG(Taint tag);
//Class: edu.columbia.cs.psl.phosphor.runtime.Taint
Taint(Object label);
LinkedList<Taint> getDependencies();
Object getLabel();
```

Fig. 4.7: Key API methods, classes and interfaces exposed by PHOSPHOR

4.4 Evaluation

We evaluated PHOSPHOR in the dimensions of performance (as measured by runtime overhead and memory overhead) and in soundness and precision. We have also compared the performance of PHOSPHOR with that of TaintDroid, when running within the Dalvik VM on an Android device. We were restricted from comparing against other taint tracking systems, as many were unavailable for download and did not utilize standardized benchmarks in their evaluations. All of our JVM experiments were performed on an Apple Macbook Pro (2013) running Mac OS 10.9.1 with a 2.6Ghz Intel Core i7 processor and 16 GB of RAM. We used four JVMs: Oracle’s “HotSpot” JVM, version 1.7.0_45 and 1.8.0 and the OpenJDK “IcedTea” JVM, of the same two versions. All instrumentation was performed ahead of time and the dynamic instrumenter therefore only needed to instrument classes that were dynamically generated (for example, by the Tomcat benchmark, which compiles JSP code into Java and runs it).

For all experiments, no other applications were running and the system was otherwise at rest. All of our Android experiments were performed on a Nexus 10, running Android version 4.3.1, built from the Android Open Source Project repository. No other applications were running on the Android device during our experiments.

4.4.1 Performance: Macro benchmarks

Our first performance evaluation focused on macro benchmarks, from the DaCapo [27] benchmark suite (9.12 “bach”), and the Scalabench [129] benchmark suite (0.1.0-20120216). The DaCapo benchmark suite contains 14 benchmarks that exercise popular open source applications with workloads designed to be representative of real-world usage. Several of these workloads are highly relevant to taint tracking applications, as they benchmark web servers: the “tomcat,” “tradebeans” and “tradesoap” workloads. The Scalabench suite contains 12 benchmarks written in Scala that are also broad in scope. In all cases, we used the “default” size workload.

First, we ran the benchmarks using both the Oracle “HotSpot” JVM and the OpenJDK “IcedTea” JVM in our test environment to measure baseline execution time. Then, we instrumented both JVMs and all of the benchmarks to perform taint tracking, and measured the resulting execution time and the maximum heap usage reported by the JVM. To control for JIT and other factors, we executed

Benchmark	Oracle Hotspot 7						Other JVMs			
	Runtime (ms)			Heap Size (MB)			Runtime Overhead			
	T_b	T_p	Overhead	M_b	M_p	Overhead	HotSpot 8	IcedTea 7	IcedTea 8	
DaCapo 9.12-bach [27]	avroa	2333 ± 53	2410 ± 27	3.3%	75	223	198.8%	.7%	3.8%	3.6%
	batik	903 ± 15	1024 ± 15	13.5%	105	211	100.2%	12.1%	N/A*	N/A*
	eclipse	15305 ± 702	48907 ± 1885	219.6%	1026	2901	182.7%	138.8%	209.8%	124.0%
	fop	203 ± 6	320 ± 7	57.7%	100	261	162.0%	63.3%	57.4%	49.8%
	h2	3718 ± 136	5137 ± 138	38.2%	739	2738	270.5%	34.0%	34.7%	35.2%
	jython	1343 ± 19	2107 ± 47	56.9%	412	805	95.1%	25.7%	59.4%	26.8%
	luindex	454 ± 50	642 ± 44	41.6%	39	157	303.6%	52.9%	44.4%	53.2%
	lusearch	584 ± 65	1126 ± 73	92.8%	619	2750	344.2%	86.6%	102.0%	92.6%
	pmd	1336 ± 20	1705 ± 56	27.6%	172	583	239.5%	26.8%	29.8%	23.5%
	sunflow	1616 ± 76	2182 ± 231	35.0%	532	1086	104.3%	28.8%	28.2%	29.1%
	tomcat	1364 ± 35	1885 ± 41	38.2%	173	881	410.7%	33.4%	30.0%	36.8%
	tradebeans	3175 ± 94	4189 ± 136	31.9%	1093	2225	103.6%	33.3%	41.4%	34.3%
	tradesoap	12159 ± 2416	14657 ± 2470	20.6%	1910	3058	60.1%	17.5%	14.1%	3.6%
	xalan	498 ± 40	748 ± 102	50.2%	91	790	771.9%	49.2%	38.5%	75.7%
Average	3214	6217	51.9%	506	1334	239.1%	43.1%	53.4%	45.2%	
Scalabench 0.1.0-20120116 [129]	actors	2523 ± 103	2663 ± 130	5.5%	90	716	692.0%	4.0%	.6%	3.5%
	apparat	7874 ± 640	13516 ± 1102	71.7%	509	2430	377.0%	102.6%	66.7%	92.8%
	factorie	19262 ± 1812	25063 ± 781	30.1%	2769	2791	.8%	38.7%	32.1%	35.5%
	kiana	238 ± 5	381 ± 11	60.3%	151	529	250.7%	51.1%	52.9%	59.7%
	scaladoc	1092 ± 25	2206 ± 85	102.1%	174	1225	602.7%	98.0%	94.0%	94.0%
	scalap	136 ± 6	227 ± 7	67.3%	86	298	248.8%	82.1%	61.6%	82.3%
	scalariform	419 ± 15	523 ± 8	24.6%	88	304	246.1%	28.9%	21.8%	23.5%
	scalatest	840 ± 55	1133 ± 73	34.9%	153	599	292.1%	45.4%	32.8%	41.9%
	scalaxb	288 ± 6	540 ± 38	87.8%	87	413	373.6%	218.8%	79.9%	219.2%
	specs	1268 ± 44	1770 ± 21	39.6%	162	714	340.8%	24.6%	36.5%	40.6%
	tmt	3755 ± 65	6834 ± 33	82.0%	2733	2777	1.6%	95.0%	81.5%	93.5%
Average	3427	4987	55.1%	637	1163	311.5%	71.7%	50.9%	71.5%	
All Average	3307	5676	53.3%	563	1259	270.9%	55.7%	52.2%	57.3%	

Table 4.1: Runtime duration for macro benchmarks, showing baseline time (T_b), PHOSPHOR time (T_p) and relative overhead for Oracle’s HotSpot JVM version 1.7.0.45, indicating standard deviation of measurements with \pm . We also show heap size measurements for the baseline execution (M_b) and PHOSPHOR execution (M_p), as well as the percent overhead for heap size. For HotSpot 8, IcedTea 7 and IcedTea 8, we show only runtime overhead. *The “batik” benchmark depends on Oracle-proprietary classes, and therefore does not execute on the OpenJDK IcedTea JVM.

Benchmark	Oracle - HotSpot 7			Rel. Overhead (Other JVMs)			Rel. Overhead (DVM)	
	T_b (ns)	T_p (ns)	Rel. Overhead	Hotspot 8	IcedTea 7	IcedTea 8	PHOSPHOR	TaintDroid
Float	5620 \pm 25	7765 \pm 47	38.2%	108%	37.7%	114.2%	131.2%	63.9%
Logic	1338 \pm 4	1341 \pm 5	0.2%	-0.3%	-0.6%	0.2%	1.5%	11.2%
Loop	3283 \pm 59	4060 \pm 38	23.7%	23.4%	22.2%	23.3%	43.9%	64%
Method	266 \pm 5	642 \pm 4	141.3%	140.1%	132.4%	137.6%	9.8%	25.3%
Sieve	6128 \pm 42	7062 \pm 87	15.2%	12.8%	14.3%	13.6%	27.7%	3.2%
String Buffer	1081 \pm 7	3396 \pm 43	214.2%	212.9%	215.9%	208.4%	183.3%	30.8%
Average	2953	4044	72.1%	82.8%	70.3%	82.9%	66.2%	33.1%

Table 4.2: Runtime duration (in nanoseconds) and overhead for micro benchmarks, showing base-line time (T_b), PHOSPHOR time (T_p) with standard deviation as \pm , and relative overhead for Oracle’s HotSpot JVM version 1.7.0_45 and 1.8.0, OpenJDK’s IcedTea JVM version 1.7.0_45 and 1.8.0, and Android’s DVM version 4.3.1. For the DVM, we also show TaintDroid’s overhead (relative to the same baseline Android configuration).

each benchmark multiple times in the same JVM until the coefficient of variation (a normalized measure of deviation: the ratio of the standard deviation of a sample to its mean) dropped to at most 3 over a window of the 3 last runs (a technique recommended in [63]). Our measurements were then taken in the next execution of the benchmark in that JVM. This process was repeated 10 times, starting a new JVM to run each experiment, and we then averaged these results.

We include results for all benchmarks except for the “scalac” benchmark from the scalabench workloads, a benchmark that exercises the Scala compiler. The Scala compiler has certain expectations about the structure and contents of class files that it compiles, so injecting taint tracking code into the compiler itself (plus the intermediate code that is being compiled) causes runtime errors. A general limitation of our approach is that applications that inspect their own bytecode directly (rather than that code being read and interpreted by the JVM, and rather than using Java’s reflection interface to inspect it) may not function correctly, as we have changed that bytecode (a limitation discussed in §4.2.3).

Table 4.1 presents the results of this study, showing detailed results for Oracle’s HotSpot JVM (version 7), and summary results for HotSpot 8, and OpenJDK’s IcedTea JVMs (versions 7 and 8). We focus on the results for HotSpot 7, as it is far more widely adopted than version 8 (at time of

submission, Java 7 was approximately three years old, and Java 8 was approximately one week old). Using Oracle’s HotSpot JVM 7, for the DaCapo suite, the average runtime overhead was 51.9%, and across the Scalabench suite, the average runtime overhead was 55.1% (runtime overhead for other JVMs is shown in Table 4.1). The average heap overhead was 239.1% for DaCapo, and 311.5% for Scalabench (heap usage in the other JVMs was similar). This heap overhead is unsurprising: in addition to requiring additional memory to store the taint tags, PHOSPHOR also increases memory usage by its need to allocate containers to box and unbox primitives and primitive arrays for return values, and primitive arrays when casting them to the generic type `java.lang.Object` (as discussed in §4.3.1.2).

There are several interesting factors that can contribute to the heap overhead growing to be more than twice as large. First, note that a Java `integer` is four bytes, while a `byte` is 1 byte, and `chars` and `shorts` are both two bytes. Therefore, the space overhead to store the taint tag for a variable can be as high as 4x.

The second factor that can adversely impact heap overhead comes from our container types. For every method that returns a primitive type, we replace its primitive return type with an object that wraps the primitive value with its taint tag. Although we pre-allocate these return types and attempt to reuse them, our implementation will only allow for reuse when (1) a method calls multiple other methods that return the same primitive type, or (2) a method calls other methods that return the same primitive type as the caller. These allocations are relatively cheap in terms of execution time (and are represented in our overall execution overhead measures), but can put significant pressure on the garbage collector that wouldn’t exist without PHOSPHOR, as primitive values are not reference-tracked. We saw a particularly heavy allocation pattern in the *xalan* benchmark, where approximately 36 million instances of `TaintedInt` and 35 million instances of `TaintedBoolean` were allocated to encapsulate return types.

In terms of runtime overhead, we saw the best performance from PHOSPHOR in the “avrora” benchmark, and worst performance in the “eclipse” benchmark. The “avrora” benchmark runs a simulator of an AVR micro controller, and from our inspection, contains many primitive-value operations. We believe that it was a prime target for optimization by the JIT compiler; indeed, when disabling the JIT compiler and running the benchmark in a purely interpreted mode, we saw an 87% overhead, much more in line with the average performance of PHOSPHOR. “Eclipse” represents a

greater mix of operations that are more complicated and computationally expensive for PHOSPHOR to implement. For instance, many parts of the Eclipse JDT Java compiler (a component of the benchmark) store primitive arrays into fields declared with the generic type, `java.lang.Object`. For every access to these fields, PHOSPHOR must insert several instructions to box or unbox the array, which requires allocating a new container each time, and hence, adding significantly to the overhead.

To compare broadly to other binary-instrumentation based taint tracking systems, DyTan [46] shows a performance overhead of 30x in a macro benchmark, with a memory overhead of 240x. LibDFT [90] shows a performance overhead of 1.14-6x on macro benchmarks. PHOSPHOR showed an average overhead of 1.5x, ranging overall from 1.03x to 3.19x. Again, it is impossible to compare directly to these systems, as they target different platforms (i.e., not the JVM) and there was no standard benchmark that we could use for the purpose.

The most applicable systems to compare PHOSPHOR to are TaintDroid [57], Trishul [103] and Chandra et al’s approach [34]. Of these, we were able to obtain TaintDroid and Trishul (the authors of [34] were unable to find their implementation [35]), but were unable to use our macro benchmarks to compare to these systems as the benchmarks are not supported by Dalvik and Kaffe respectively (the VMs used by Trishul and TaintDroid). The authors of TaintDroid used the CaffeineMark [114] benchmark in their evaluation, and the authors of Trishul used the jMocha benchmark [67] in their evaluation. We compare PHOSPHOR’s performance directly to TaintDroid and Trishul in the following section.

4.4.2 Performance: Micro Benchmarks

We performed a series of micro benchmarks to further analyze PHOSPHOR’s runtime performance overhead. Our micro benchmarks are based on the CaffeineMark [114] suite of micro benchmarks, commonly used by Android developers – including by the authors of TaintDroid [57]. We modified these benchmarks to run under Google’s Caliper micro benchmark tool, so that they could benefit from the framework’s warmup, timing, and validation features (the original CaffeineMark benchmarks do not contain any warmup phase and therefore the results can be skewed by JIT compilation). The “embedded” suite (used in the TaintDroid study) consists of six benchmarks: “Float” (simulates 3D rotation of objects around a point; uses arrays), “Logic” (contains many simple branch

conditions), “Loop” (contains sorting and sequence generation; uses arrays), “Sieve” (uses the sieve of eratosthenes to find primes; uses arrays), “Method” (features many recursive method calls) and “String” (performs string concatenation; uses arrays). Each benchmark was executed several times in the same JVM over the course of 3 seconds to warm up, and then executed for a period of 1 second. For that last second, we measure the amount of time in nanoseconds that each benchmark took (by running it many times and averaging). We did this entire process 10 times, and averaged the results of each trial.

Table 4.2 shows the results of this study, showing the runtime for PHOSPHOR for Oracle’s HotSpot 7 JVM (being the most popular JVM at time of publication), and the runtime overhead for all of the subject JVMs, plus the Android DVM. We also show our measured overhead of TaintDroid, relative to the same baseline Android DVM. PHOSPHOR’s fine-grained array taint tag tracking (i.e. that it stores a taint tag per-element, rather than a single tag per-array) caused it to perform somewhat poorer than TaintDroid in the benchmarks that relied heavily on arrays. Recall that this optimization will result in a loss in precision for TaintDroid, which does not affect PHOSPHOR.

However, in the benchmarks that did not involve significant array usage (e.g. “Logic,” “Loop,” and “Method”), PHOSPHOR outperformed TaintDroid. It would be interesting to perform a followup study by modifying TaintDroid to also track taint tags per-element, to see which approach is faster in that case. Another interesting observation from the micro benchmarks is that the average overhead across these micro benchmarks for PHOSPHOR (72.13%), is somewhat higher than its average overhead across the macro benchmarks (52.06%). Perhaps these less than optimal cases occur less in practice than those cases wherein PHOSPHOR is faster. Unfortunately we are severely restricted in availability of macro benchmarks for Android (DaCapo is not easily ported to Android as many of its benchmarks rely on Java APIs that are not included in the Android Dalvik VM), and therefore could not perform a macro benchmark study comparing TaintDroid with PHOSPHOR.

To compare to Trishul [103], we hoped to use the same suite of micro-benchmarks (the suite used by the authors of Trishul, jMocha, is no longer available) used above. However, we found that the benchmark framework that we used to collect timing information (Google Caliper version 0.5) was incompatible with Kaffe, the JVM that Trishul is built upon. Therefore, we selected another suite of micro benchmarks to run for this purpose: JavaGrande [32], a micro benchmark suite from 2000, which was popular at the time (and worked with Kaffe). We performed these experiments in

Benchmark Group	Relative Overhead to HotSpot 7		
	Kaffe	Trishul	PHOSPHOR
Arithmetic	64.4%	74.5%	10.7%
Assign	56.5%	86.6%	50.2%
Cast	87.1%	86.7%	13%
Create	98.5%	98.8%	24.4%
Exception	90.0%	69.9%	1.7%
Loop	2.3%	89.0%	7.6%
Math	89.1%	96.5%	96.0%
Method	42.2%	76.0%	6.3%
Serial	90.04%	N/A*	21.14%
Average	68.9%	84.7%	25.7%

Table 4.3: Runtime overhead of PHOSPHOR, Kaffe 1.1.7 [142] and Trishul [103] compared to Oracle HotSpot 1.7.0_55 on the JavaGrande benchmark [32]. *Threw exception

an Ubuntu 6.10 VirtualBox VM with 3.5GB of RAM (running on the same MacBook Pro 2.6Ghz Intel Core i7, 16 GB of RAM) that was provided by the Trishul authors. We measured the performance of Oracle’s HotSpot 7 running within this VM as a baseline, and then also measured the performance of Kaffe 1.1.7 (which Trishul is based on), Trishul, and a PHOSPHOR-instrumented HotSpot 7. Again, we executed each benchmark 10 times (each time in a separate JVM), but here, with no warmup phase, as JavaGrande’s benchmark runner does not support a warmup phase (and we were unable to use Google Caliper to control for warmup as it was not supported by Kaffe VM).

Table 4.3 presents the results of this evaluation (overheads presented are relative to HotSpot 7). Note that in most cases, Kaffe itself (the VM that Trishul is built on) is significantly slower relative to HotSpot, and hence, perhaps some large amount of the overhead imposed by Trishul can be attributed to the underlying VM. In all cases, PHOSPHOR had significantly lower overhead than Trishul (in the case of the “Serial” benchmark, Trishul threw an exception and was unable to execute the benchmark). Trishul’s performance on the loop and method benchmarks was particularly poor (even relative to an unmodified Kaffe VM), likely due to the fact that it performs control flow tainting, and not just data flow tainting. Adding control flow tainting to PHOSPHOR would likely also increase its overhead in these two benchmarks.

4.4.3 Soundness and Precision

We evaluated the soundness and precision of PHOSPHOR using two benchmark suites. First, we wrote our own suite of unit tests, testing that each of our taint tracking properties (as described in §4.3.1) are not violated, for each primitive and primitive array type, as well as for reference types. PHOSPHOR passed all of these tests. These unit tests are included in our GitHub repository [16].

To add additional validity to our claim that PHOSPHOR is sound and precise, we also implemented the DroidBench [12] taint tracking benchmark, removing the components that were Android specific so that it would run on a desktop JVM. DroidBench consists of 64 test cases for taint tracking systems, of which, we found 35 to be Android-specific (testing taint propagation through Android-specific callbacks and life-cycle events), leaving 29 tests. These tests are designed to test both soundness (that variables that should be tainted are indeed tainted with the correct taint) and precision (that variables that should not be tainted are not tainted) of taint tracking. Four of the tests are designed to test taint tracking through implicit flows. PHOSPHOR passed all data flow tests and failed on the four implicit flow tests as expected.

4.4.4 Portability

We further studied the portability of PHOSPHOR by attempting to apply it to three completely different JVMs (in addition to the two versions of Oracle’s HotSpot and OpenJDK’s IcedTea, plus the Dalvik DVM). We downloaded the most recent versions of the Apache Harmony JVM (version 6.0M3) [10], Kaffe VM (version 1.1.9) [142] and Jikes RVM (version 3.1.3) [141]. For each VM, we attempted to execute our soundness and precision tests as a basic indicator of whether PHOSPHOR would work.

While PHOSPHOR did not work immediately with Harmony or Kaffe, after approximately 30 minutes of debugging, we identified several additional classes that were tightly coupled between the class library and the interpreter. For instance, no JVM that we tested allowed for unrestricted modifications of the class `java.lang.Object`; Harmony and Kaffe similarly would not allow for modifications of the class `java.lang.VMObject` (which does not exist in Oracle or OpenJDK’s class library). We patched around these classes, and can confirm that PHOSPHOR works with Harmony and Kaffe.

However, we were unable to successfully apply PHOSPHOR to the Jikes RVM, which is a JVM

implemented in Java. We believe that this is due to our inherent design limitation (discussed further in the following section), that should an application try to read its own bytecode, it will see unexpected entries (namely, everything added by PHOSPHOR). Jikes uses its own internal implementation of Java's reflection library for configuring its bootstrap class image, and PHOSPHOR does not currently patch this to hide its modifications, causing it to fail. We believe that it would be possible to modify PHOSPHOR to be compatible with Jikes, but have not investigated this further.

4.4.5 Threats to Validity

The main threats to validity to our experiments are related to our claims of portability. We claim that PHOSPHOR is portable to any JVM that fulfills the official JVM specifications versions 7 and 8, as it only requires modifications to application bytecode and library bytecode. We evaluated this claim on four JVMs, including two versions of two very widely used JVMs (Oracle HotSpot and OpenJDK IcedTea), and two much less frequently used JVMs (Kaffe and Harmony). Just as these JVMs had tight coupling for several classes, preventing PHOSPHOR from adding fields to them to track taint tags, it is certainly possible that other JVMs have even more constraints on more classes (such coupling between class libraries and interpreter are not discussed in the JVM specification). However, we are confident that if such cases arose, PHOSPHOR would still be applicable, falling back to storing taint tags for instances of such classes with a HashMap, an approach that would still work, though perhaps with somewhat higher overhead (such changes would need to be manually implemented). We believe that PHOSPHOR's incompatibility with the Jikes Research Virtual Machine is an exceptional case in that (1) it is intended specifically for research purposes and not production purposes, (2) it is written in Java itself and is self-hosted (i.e. its Java code runs on itself). Moreover, although we were unable to find any usage statistics, we believe that Oracle HotSpot and OpenJDK IcedTea dominate the JVM market by far.

Although we selected popular, well-accepted macro benchmarks for evaluating PHOSPHOR, it is possible that the selected benchmarks are not representative of the sorts of workloads that would normally be targets for taint tracking. However, because three of these benchmarks involve workloads on web servers, and taint tracking has been shown to be highly applicable to detecting and preventing code injection attacks in web servers, we believe that the benchmarks are sufficient.

There are several key limitations to our approach, as discussed previously in §4.2.3, most no-

tably that PHOSPHOR only tracks data flows, and not control flows (“implicit flows”), much like other well known taint tracking systems [37, 57, 90]. Note that implicit flow tracking primarily requires static analysis, and its implementation should be unaffected by PHOSPHOR’s approach to data flow tracking. Support for implicit flows would be interesting to add as an optional feature to PHOSPHOR (e.g. DyTan [46] supports both sorts of tracking), but we consider this to be future work, outside of the scope of this thesis.

Java provides a simple reflection API (also used by many Scala applications) to access information about class files, such as the list of methods available in a class. PHOSPHOR patches this API to hide all of its changes from applications, however, if an application directly reads in the byte stream of a Class file (without using this API) and parses its structure, that application will find potentially unexpected artifacts of PHOSPHOR in the Class. This scenario arose in our macro benchmark study exactly once: in the case of the Scala compiler (“scalac”), which does not use the reflection API. We do not believe that this is a common occurrence outside of the scope of compilers, as Java’s reflection API is widely used for this purpose.

4.5 Related Work

Dynamic taint analysis is a problem widely studied, with many different systems tailored to specific purposes and languages. For instance, there are several system-wide tainting approaches based on modifications to the operating system ([135] and others). However, PHOSPHOR tracks taint tags by instrumenting application byte code. This general approach is most similar to other approaches that track taint tags by instrumenting application binaries. When available, we compared the Java-based systems directly to PHOSPHOR (an evaluation presented in Section 4.4), but please note that the performance overheads reported in this section are to provide ballpark information only — the selection of benchmarks used varies greatly from system-to-system (the slowdowns reported here are provided by the original authors).

DyTan is a general purpose taint tracking system targeting x86 binaries that supports implicit (control) flow tainting, in addition to data flow tainting, with runtime slowdown ranging from 30x-50x [46] (where a slowdown of 1x means that the system now takes twice as much time to run). TaintTrace only performs data flow tainting, and achieves an average slowdown of 5.53x [40].

Libdft, another binary taint tracking tool, shows overheads between 1.14x-6x, thanks to optimizations largely based on assumptions that data (overall) will be infrequently tainted [90]. In contrast, PHOSPHOR does not assume that variables are mostly not tainted (and hence does not make such optimizations, although they mostly are still applicable to the JVM), and therefore its performance will remain constant regardless of the frequency of tainting.

Another general class of taint tracking systems target interpreted languages and make modifications to the language interpreter, targeting, for example, JavaScript [150], Python [157], PHP [105, 130, 157], Dalvik [57] and the JVM [34, 103]. In general, interpreter level approaches can benefit from additional information available in the context of the language that defines the exact boundary of each object in memory (so soundness and precision can be improved over binary-level approaches). The portability of these systems is often restricted, as they require modifications to the language interpreter and/or modifications to application source code.

Of these interpreter-based taint tracking systems, the most relevant to PHOSPHOR are Trishul [103], an approach by Chandra et al. [34], and TaintDroid [57]. Trishul performs data and control flow taint tracking by modifying the Kaffe interpreted JVM, an open source JVM implementation (in a purely interpreted mode, with no JIT compilation — adding an inherent slowdown of several orders of magnitude). Chandra et al. modifies the Jikes Research Virtual Machine to perform data and control flow taint tracking, showing slowdowns of up to 2x on micro-benchmarks, but its implementation depends on the usage of the research VM, rather than a more popularly deployed JVM [34]. Neither the Jikes nor the Kaffe JVM support the complete Java language specification. TaintDroid is a popular taint tracking system for Android’s Dalvik Virtual Machine (DVM), implemented by modifying the Dalvik interpreter [57]. TaintDroid only maintains a single taint tag for every element in an array (unlike PHOSPHOR, which maintains a tag for each element), allowing TaintDroid to perform more favorably on array-based benchmarks, but at the cost of precision.

While all of these approaches employ variable-level tracking, like PHOSPHOR, the key difference that sets PHOSPHOR apart is its portability: each of the above systems requires modifications to the language interpreter. For example, TaintDroid’s most recent version (version 4.3 at time of publication) adds over 32,000 lines of code to the VM (as measured by lines of code in the TaintDroid patch to Android 4.3.1). For any new release of the VM, the changes must be ported into the new version and if a researcher or user wished to use a different VM (or perhaps a different architecture),

they would need to port the tracking code to that VM. PHOSPHOR, on the other hand, is designed with portability in mind: PHOSPHOR runs within the JVM without requiring any modifications to the interpreter (and we show its applicability to the popular Oracle HotSpot and OpenJDK IcedTea JVMs). This design choice also allows us to support Android’s Dalvik Virtual Machine with only minor modifications, as discussed in §4.3.6.

There have been several recent works in dynamic taint tracking for Java that operate by modifying core Java libraries to track taint tags. Without requiring interpreter modification, WASP detects and prevents SQL injection attacks in Java by using taint tracking with low overhead (1-19%), but is restricted to only track taint tags on Strings [77], much like the earlier Java tainting system by Haldar et al. [76], and Chin et al’s optimized version of the same technique [42]. These systems simply provide a replacement for the class, `java.lang.String` that is manually modified to perform taint tracking for those objects, and the approach is therefore unsuited to general purpose taint tracking (aside from Strings). PHOSPHOR differs from all of these approaches in that it tracks taints on all forms of data within the JVM: not just Strings.

Vitasek et al. propose a solution to a problem related to taint tracking: in addition to assigning labels to each object in the JVM, their ShadowData system can also enumerate all such labels [149]. Vitasek et al. evaluated several approaches to this, finding the most efficient to be storing the mapping from object to label in a HashMap, showing slowdown ranging from 4.8x-185.5x, largely due to contention in accessing that HashMap, a drawback that PHOSPHOR’s decentralized taint tag storage avoids (but note that PHOSPHOR does not provide the ability to enumerate all data that is tagged).

4.6 Conclusions

Due to difficulties simultaneously achieving precision, soundness, and performance, all previous implementations of dynamic taint analysis for JVM based languages have been restricted, functioning only within specialized research-oriented JVMs, making their deployment difficult. We presented PHOSPHOR, our approach to providing accurate, precise, and performant taint tracking within the JVM without requiring any modifications to it, demonstrating its applicability to two very popular JVMs: Oracle’s HotSpot and OpenJDK’s IcedTea, each for the two most recent versions: 1.7 and

1.8. Moreover, PHOSPHOR does not require any specialized operating system or specialized hardware or access to application source code. PHOSPHOR appeared originally at OOPSLA 2014 [19], and additional information on its control flow tracking appeared at ISSTA 2015 [21]. PHOSPHOR is available on GitHub [16], and passed the OOPSLA 2014 artifact evaluation process.

Chapter 5

Detecting Hidden Object Structure

One particularly application of taint tracking and dynamic data flow analysis to software reliability is data protection analysis and testing. Despite recent high-profile failures in applications' management of our data [7], in the absence of system-level support for fine-grained data organization, we are forced to entrust them with our data. When users perform day-to-day data management activities – deleting individual emails, identifying specific data that was viewed, or sharing pictures – they are forced to rely on applications to behave properly. Yet, a 2010 study of 30 popular Android applications showed that 20 leaked sensitive data, such as contacts or locations [58]. Our own study of deletion practices within mobile apps, described later in this chapter, revealed that 18 of 50 popular Android applications left information behind instead of deleting it. Notably, we found that until 2011, Android's default email application left behind the attachments of deleted emails while deleting the messages themselves.

Although a plethora of system-level data management tools exist – including encrypted file systems [69,75], deniable file systems [145], auditing file systems [61], or assured delete systems [115] – these tools operate at a single level of abstraction: *files*. Without a one-to-one mapping between user-relevant objects (for example, individual email messages in a mail client or documents in a word processor) and files, such systems provide poor granularity, preventing end-users from protecting individual objects that matter to them.

Consider Android's default email application: it stores each email's contents and to/from/subject fields as several rows in a SQLite database (all emails are stored in the same DB, which is itself stored as a single file), attachments as files, and cached renderings of messages in different files.

Such complex object-to-file mappings are typical in Android, as our large-scale measurement study of Android storage patterns shows (§5.2). Moreover, others have observed complex storage layouts in other OSes, such as OSX, where researchers have concluded that “a file is not a file” but a complex structure with complex access patterns [81].

Given the complexity of these object-to-file mappings, we ask: is it possible for system-level tools to support management and protection at the granularity of user-relevant objects? Intuitively, this would require developers to specify the structure of their applications’ persisted data to the operating system. Nevertheless, we observe that the high level storage abstractions included and predominant in today’s operating systems – the SQLite relational database in Android and the CoreData object-relational mapper in iOS – bear sufficient structural information to recover these user-relevant data objects from unmodified applications.

We call these objects *logical data objects* (LDO), examples of which include an email (including its to, from, subject, body, attachments and any other related information); a mailbox including all emails in it; a bank account in a personal finance application; etc. We present *Pebbles*, a system that exposes LDOs to protection tools, without introducing any new programming models or interfaces, which can be prone to programmer error, slow adoption, or incompatibility with legacy applications.

We implemented Pebbles and several new protection tools based on it on the Android platform. Each of these tools provides protection at the LDO level, leveraging Pebbles to greatly simplify their development. Using Pebbles tools, users can mark objects from their existing applications to verify their proper deletion, protect their access from other applications, and back them up to the clouds they trust.

In a study of 50 popular Android applications, we found Pebbles to be highly effective in automatically identifying LDOs. Across these apps, object recognition recall was 91% and precision was 97%. In other words, in 91% of the cases, there was no leakage of data from user-visible objects to LDOs, and in 97% of the cases, there was no over-inclusion of extra data beyond user expectation in LDOs. Pebbles relies on several key assumptions based on common practices. Many of the cases in which Pebbles had poor accuracy, it could have been addressed had the developers followed these common practices.

Overall, this work makes the following contributions:

1. A study of over 470,000 Android apps, analyzing, for the first time at scale, the storage

abstractions in common use today (§5.2). Our results suggest major differences compared to traditional storage abstractions, which render file-level data management ineffective while creating untapped opportunities for object-level data management.

2. The first design and implementation of a persistent data object recognition system that requires no app changes (§5.3 and §5.5). Our design taps into the opportunities observed from our large-scale Android app study. We make our code available from <https://systems.cs.columbia.edu/projects/pebbles>.
3. Four protection tools implemented atop Pebbles, demonstrating the power and value of application-level objects to protection tools (§5.4).
4. An evaluation of LDO construction accuracy with Pebbles over 50 popular applications from Google Play, showing it to be effective in practice (§5.6) and underscoring its well-defined failure modes (§5.7).

5.1 Motivation and Goals

We begin by presenting a set of example scenarios that highlight the need for fine-grained data management support within modern OSes.

5.1.1 Example Scenarios

Scenario 1: Object Deletion: Ann, an investigative journalist, has received an extremely sensitive email on her phone with an attachment that identifies her sources. To protect her sources, Ann does her due diligence by deleting the email immediately after reading its contents and restarting her phone to clean up any traces left in memory. Her phone is already configured with an assured-delete file system [115] that deletes data promptly upon request. Worried that the application might have created a copy of her data without her knowledge or control, she wonders: *Is there any remnant of that email left anywhere on the phone?* She is disappointed to realize that she has zero visibility into the data stored on her device. Weeks later, she learns that her fears were well-founded: the email app she is using contains a bug that leaves attachments intact when an email is deleted.

Scenario 2: Object Access Auditing: Bob, a financial auditor, uses his phone for all interactions with client data while on field engagements. Recently, Bob’s device was stolen. Fearing that his fingerprint unlock might not withstand motivated attackers [140], Bob asked his IT admin a natural question: *Has any of my clients’ data been exposed?* The admin’s answer was mixed. Although activity on Bob’s phone was tracked by a remote auditing file system [61], the logs show that a file, `/data/data/com.android.email/cache/7dcee8`, was accessed immediately before the phone’s wipe-out. The file stores the HTML rendering of an email, but no one knows *which* email. Bob is left wondering what he should disclose to clients about the potential exposure of their data, and to *which* clients, since neither he nor the IT staff can map that file to a specific client or email.

Scenario 3: Object Access Restriction: Carla, a local politician, uses her phone to take photos for professional purposes, but she has several personal photos on it as well. She uses a cloud-based photo editor to enhance her promotional photos before posting them. Due to the coarse-grained permissions model of her Android device, she must provide this photo editor with access to all of her photos in order to use it. Carla is concerned that the photo editor may be secretly collecting all the photos from her device, including several potentially sensitive photos that could be politically compromising.

5.1.2 Goals and Assumptions

The above hypothetical users, along with millions of real-life users of mobile technology, have a mental model of application-level objects that is not matched by current protection tools. Ann wants to ensure that a particularly sensitive email is deleted in full, including attachments, to, from, any related caches, and other fields; Bob wants to know the sender or contents of a compromised email instead of a meaningless file name; Carla wants to protect a few of her most sensitive photos from prying applications. Traditional protection tools, such as file-based encryption, auditing, or secure deletion cannot fulfill these needs because the mapping between objects and files is application-specific and complex. The alternative, whole-disk encryption [6, 136], does not provide the flexibility that these users need.

To support such object-level data management needs, we developed Pebbles, a system that automatically reconstructs application-level logical data objects (LDOs) from unmodified applications. Pebbles exposes these LDOs to any system-wide protection tool that could benefit from under-

standing application-level objects. An encryption system could use LDOs to support meaningful fine-grained protection as an extra layer on top of whole-disk encryption. An auditing system could use LDOs to provide meaningful information about an accessed component. An object manager could reveal to users which parts of an object are left after deletion. And a backup system could let users choose their most sensitive objects for backup onto a trusted, self-managed server, letting the rest be backed up into the cloud.

Goals. The Pebbles design was guided by three goals:

- G1: Accurate and Precise Object Recognition:* Pebbles objects (LDOs) must closely match application-level persisted objects. This includes: (a) *avoiding data leaks* (if an item belongs to an LDO it must be included), and (b) *avoiding data over-inclusions* (if an item does not belong to an LDO it should not be included).
- G2: Meaningful Granularity:* Pebbles must recognize LDOs that are meaningful to users, such as individual emails.
- G3: No New Application APIs:* Pebbles must not require app developers to use new APIs; it can recommend developers to follow existing common practices but must work well even if they do not precisely follow.

Our first goal is accurate and precise object recognition (*G1*). We aim to achieve (1) good object recognition *recall* by avoiding leaks and (2) good object recognition *precision* by avoiding over-inclusions. We acknowledge that perfect recall or precision cannot be guaranteed in either an unsupervised approach or in a supervised API approach with imperfect developers, since a poorly written app could convolute data structure in a way that Pebbles cannot recover. However, we wish to formulate clearly all potential sources of leakage, to design mechanisms to address the leakages for most applications (§5.3.2), and to remind developers how they could avoid such leakages by following existing common practices (§5.7).

Related to *G1*, our second goal (*G2*) is to recognize relevant and meaningful LDOs. For example, in an email app, Pebbles should be able to recognize individual emails, not just coarse accounts with many emails. We note here that Pebbles identifies application-level objects that are persisted in stable storage, and we assume that those have a direct mapping onto the objects that users interact with and wish to protect.

G3 stems from our skepticism that developers will convert applications to use new security-related APIs or correctly use such APIs. However, we do expect that most developers will follow certain common practices (as evaluated in §5.2). Pebbles addresses this by leveraging application-level semantics already available within storage abstractions such as database schemas, XML structures, and the file system hierarchy. Pebbles also provides recommendations for developers which are rooted in already popular development practices (§5.7).

Threat Models and Assumptions. Pebbles is designed to support fine-grained data management – such as encryption, auditing, and deletion of individual emails, photos, or documents – within modern OSes. The specific threat model for a given protection tool depends on that tool’s goal; however, Pebbles’s mechanisms should bolster the guarantees applications can provide. In general, we assume that protection tools are *trusted* system-wide services. This is similar to assumptions made by encrypted file systems, assured-delete file systems, and other current fine-grained data management tools.

We also assume that mobile applications that create or have access to a particular object, or part thereof, will not obfuscate their data’s structure or act maliciously against Pebbles. For example, they will not create their own data formats and will not willfully interfere with analysis mechanisms involved in object discovery. An application that has not yet been given access to data of a particular object, however, need not be trusted.

The scope of Pebbles is confined to those application-level objects that are persisted into a device’s stable storage. We explicitly ignore attackers with access to either RAM or the underlying OS or hardware. If volatile memory protection is important, we recommend combining Pebbles with secure memory deallocation [44,45,71], OS buffer cleaning [54], and idle in-RAM data eviction [138] mechanisms. We also assume that secure disk scrubbing [117, 139] is deployed. In addition, while many modern applications include a cloud component, which stores or backs up data, Pebbles currently ignores that component. In the future, we plan to extend Pebbles LDOs to transcend the local and cloud environments.

While some may believe that users are incapable of dealing with fine-grained controls, we believe that there are many circumstances in which users want and are capable of handling *some* level of control, particularly for their most sensitive data. Evidence that users are capable of handling, and require, some level of control *when they feel it is important for them to do so* is available in prior

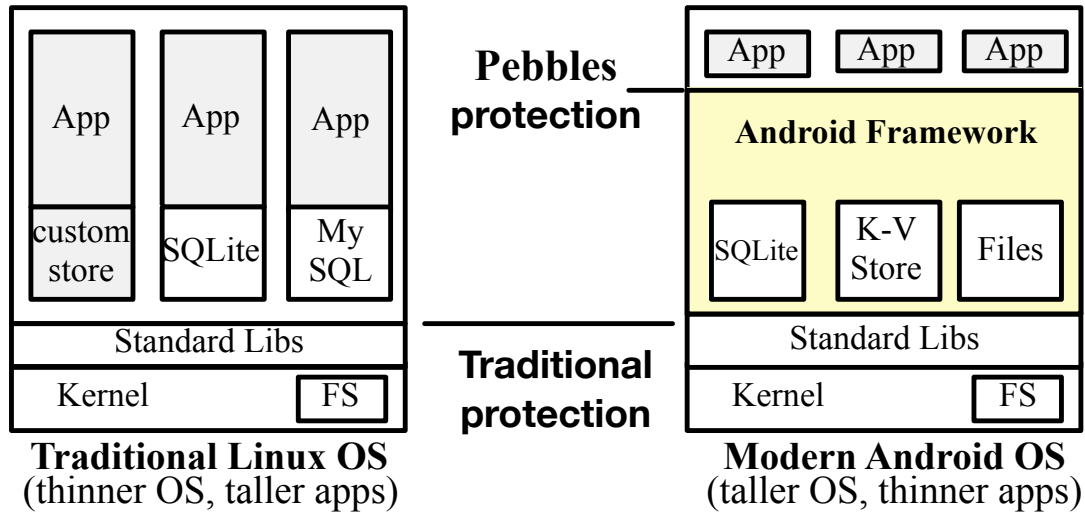


Fig. 5.1: OS Storage Abstraction Evolution. Modern OSes provide higher-level abstractions for data management, yet protection is often at the traditional file level. Pebbles, our aligns data protection with modern abstractions.

studies [41,96]. Such evidence can also be gauged from the immense popularity of data hiding apps, such as Vault-Hide [107] and KeepSafe Vault [89], which have garnered over 10 million downloads each and let users hide data, such as photos, contacts, and SMSes.

5.2 Study: Android Storage Abstractions

The Pebbles design is motivated and informed by our high-level observation that storage abstractions within modern OSes are evolving in major yet unquantified ways. Fig.5.1 shows this evolution. Specifically, we hypothesize that the inclusion of high-level storage abstractions, such as the SQLite database in Android or the CoreData abstraction in iOS, has created a new “narrow waist” for storage abstractions that largely hides the traditional hierarchical file system abstraction. These new storage abstractions should bear sufficient structure to let us reverse engineer application-level data objects from the OS’s vantage point.

In this section, we perform a simple measurement study to gauge the use of these abstractions and extract useful insights to inform our design of Pebbles. We specifically ask the following questions:

Storage Abstraction	# Apps (of 98)	Example Apps
No storage	5	Cardio Trainer
DB only	43	CWMoney, Amazon, BestBuy, Browser, Calendar, Contacts, ColorNotes, EverNote
FS only	3	Exchange Rates
KV only	5	Google Talk, Biorythms
DB+FS	24	OlNote, Angry Birds, DropBox, Gallery
DB+KV	1	Twitter
FS+KV	2	Adobe Reader, Temple Run
DB+FS+KV	15	Email, Antivirus, Amazon Kindle, Astro File Manager, Box, EBay

(a) Use of SQLite (DB), FS, and key/value (KV) store

Storage Library	# of Apps (of 476,375)
ORMLite	6,846 (1.4%)
SQLCipher	168 (0.3%)
DB4o	116 (0.2%)
H2	16 (0.0%)
Other 4 libs combined	38 (0.0%)

(b) Third-party library use

App	Object	DB/FS Use
	Email	to/from/date in one DB table; contents in another table; attachments in FS
Email (DB+FS+KV)	Mailbox	name/server/account in one DB table; includes emails; backup in kv
	Account	address/meta data in one DB table; includes mailboxes, emails
OlNote (DB+FS)	Note	title/note/tags/ in one DB table; notes exported as files in /sdcard FS
	Expense	name/amount in one DB table
CWMoney (DB only)	Category	category name in one DB table; includes expenses
	Account	name/balance in one DB table; includes categories, expenses

(c) Example object structures

Fig. 5.2: Storage API Usage in 98 Android Applications. (a) Number of apps that use the various storage abstractions in Android. Most apps use DB, but many also use FS and KV together with DB. (b) Use of eight other storage libraries among 476K free apps from Google Play. Third-party storage libraries are largely irrelevant. (c) Structure of sample objects in a few popular apps. Object structure is complex and spans multiple abstractions.

Q1 What storage abstractions do Android apps use?

Q2 How do individual apps organize their data?

Q3 How are these abstractions used?

Background. Android provides three storage abstractions [68] relevant to this thesis:

1. *SQLite Database*: Stores structured data.
2. *XML-based Key/Value Store*: Stores primitive data in key/value pairs (also known as the SharedPreferences API).
3. *Files*: Stores unstructured data on the device's flash memory.

Methodology. We ran both *static* and *dynamic* experiments. Static experiments can be run at large scale but lack precision, while dynamic experiments provide precise answers but can only be run at small scale. For static experiments, we decompiled Android applications and searched their source code for imports of the storage abstractions' packages (e.g., `android.database.sqlite`). We ran large-scale, static experiments on 476,375 apps downloaded through a February 2013 crawl of Google Play [148], the main Android app market. For the dynamic experiments (over 98 apps),

we installed Android apps on a Nexus S phone, manually interacted with them, and logged their accesses to the various APIs. These were some of the most popular apps, cutting across categories such as email clients, editors, banking, shopping, social, and gaming.

Results. *Q1 Answer: Apps primarily use SQLite, but use other abstractions as well.* Fig. 5.2(a) classifies apps according to the Android-embedded storage abstractions they use during execution. It shows that the usage of Android-provided abstractions – SQLite (denoted DB) and the key/value store (denoted KV) – eclipses the traditional file abstractions (denoted FS). Very few apps rely on the FS as their only storage abstraction (4/98). Almost half of the apps rely solely on SQLite for all of their storage needs (43/98), while almost all apps that have some local storage use SQLite (81/92). Even apps that one would consider to be primarily file-oriented (e.g., Astro File Manager, DropBox) use SQLite. A significant fraction of the apps (41/98) rely on more than one abstraction, and a notable fraction (15/98) rely on all three abstractions. This last result suggests a complex disk layout, a topic discussed further below. Overall, the most popular formations are: DB-only (43/98), DB+FS (23/98), and DB+FS+KV (15/98).

A related question is whether mobile apps use storage abstractions *other* than those provided by Android. Angry Birds, for example, stores game data and high scores in opaque binary files. We also searched the Internet for recommended Android storage options beyond those included in the OS, finding eight third-party libraries. We searched our 476K-app corpus for use of those libraries, and present the results in Fig. 5.2(b). None of these libraries are popular: only 2% of the apps use even one of them. Our dynamic experiments found that none of these libraries are used and provided no indication of additional libraries that we might have overlooked.

Q2 Answer: Data objects span multiple storage abstractions. Fig. 5.2(c) shows the structures of several logical data objects, representative of what users think and care about in various applications. It shows that objects often have complex structures that involve multiple storage abstractions. For example, Android’s default email client, an example of the DB+FS+KV formation, stores various fields of the email object in two DB tables, attachments in the FS, and account recovery information in the KV. Object structure is fairly complex even for DB-only apps, such as CWMoney, a personal finance app, where a category includes metadata in one table and all expenses in another table. It thus spans multiple tables that are not linked together through explicit foreign keys. This suggests that protecting each storage abstraction separately will not work: any

data protection abstraction at the end-user object level must span multiple storage abstractions.

Q3 Answer: SQLite is the hub for data management. Given this complexity, a natural question concerns how one can even begin to build some meaningful protection abstraction. Using a modified TaintDroid (a popular data flow taint tracking system for Android [58]) version, we tracked the flow of data between storage abstractions, confirming that at least 70/81 apps that use the DB use it as a *central hub* for managing their data. By central hub, we mean that data flows mostly from the DB into the FS/KV (when they are used) or is accessed using pointers from the DB; an observation that was true for 27 of the 38 apps that use FS or KV in addition to the DB. For example, many apps, including Email, use files to store caches of rendered versions of data stored in SQLite (such as the body of an email) or blobs of data that are indexed and managed through SQLite (such as the contents of pictures, videos, or email attachments).

Thus, SQLite is not just frequently used; it is the central abstraction in Android that originates or indexes much of the data stored in the other abstractions. This result is encouraging because, intuitively, relational databases bear more explicit structure.

Implications for the Pebbles Design. Overall, our results suggest that while the storage abstraction landscape is fairly complex in Android, there is sufficient uniformity to warrant constructing of a broadly applicable object system. Such a system must detect relationships between objects stored in different abstractions. The results suggest that SQLite, a relational database that bears significant inherent structure, is the predominant storage abstraction in Android. Raw files, which lack such structure, are just used for overflow storage of bulk data, such as images, videos, and attachments. Based on these insights, we construct Pebbles, the first system to recognize application-level objects within modern operating systems without application modifications.

5.3 The Pebbles Architecture

Pebbles aims to reconstruct application-level LDOs – emails and mailboxes in an email app, saved high scores in a game, etc. – from the bits and pieces stored across the various data storage abstractions without requiring application modifications.

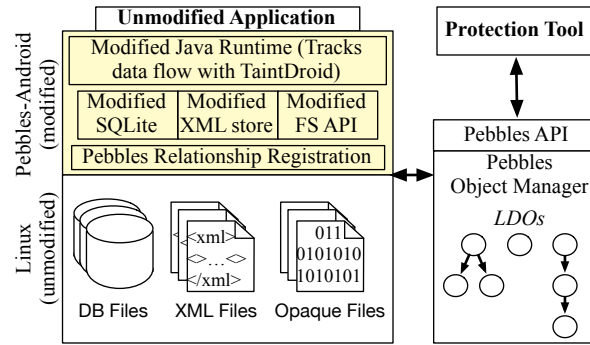


Fig. 5.3: The Pebbles Architecture. Consists of a modified Android framework and a device-wide Pebbles Object Manager. The modified framework identifies relationships between persisted data items, such as rows, XML elements, or files. The Pebbles Object Manager uses those relationships to construct an object graph; nodes map to persisted data items and edges map to relationships.

5.3.1 Overview

Fig. 5.3 shows the Pebbles architecture, which consists of two core components: (1) *Pebbles Android*, a modified Android framework that interposes on the various storage APIs, and (2) the *Pebbles Object Manager*, a separate device-wide entity for building object graphs and interacting with protection tools.

At the most basic level, the Pebbles Android framework understands units of storage (e.g., rows in DB, elements in XML, and files in FS) which become nodes in our object graph. The Pebbles Android framework then retrieves explicit relationships between these nodes and derives implicit relationships by tracking data flows between these units. The Pebbles Android framework registers these relationships with the Pebbles Object Manager using an internal registration API. The Pebbles Object Manager then stores these relationships, compiles a device-wide *object graph*, derives LDOs from the graph, and exports the LDOs to protection tools via the Pebbles API. LDOs are defined as follows: given a node in the graph (e.g., corresponding to a row in the `Email` table) an LDO is the transitive closure of the nodes connected to it. §5.6 evaluates Pebbles performance in terms of precision and recall. In the context of the graph, a failing of recall is missing nodes which should be included in a transitive closure (“leakage”); a failing of precision is including nodes which should *not* be included in a transitive closure (“over inclusion”).

To provide a concrete example of the challenges faced by Pebbles, consider Fig. 5.4, a simplified

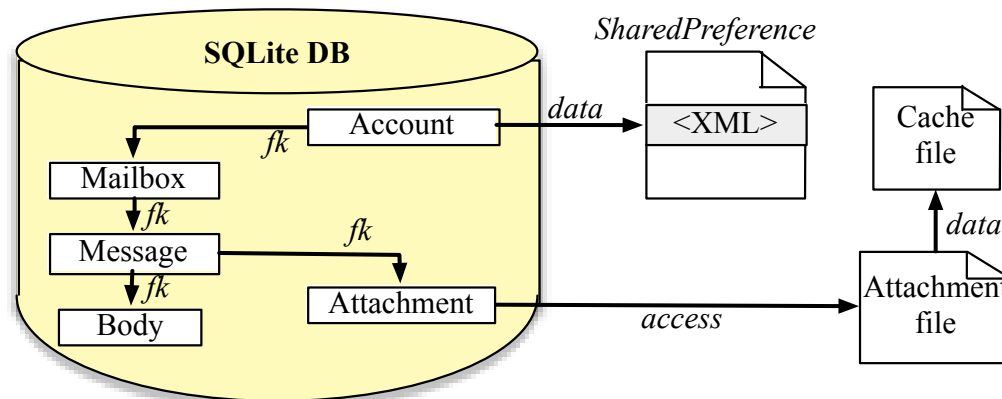


Fig. 5.4: Android Email App Object Structure. A simplified object graph for one account with one mailbox, message, and attachment. Each node represents an individual file, row, or XML element, and each edge represents a relationship. While objects can be spread across the DB, FS, and Shared Preferences, the DB remains the hub for all data.

view of how data is stored by the default Android Email application. As described previously in §5.2, this app stores its data across all three storage abstractions: SQLite database, SharedPreferences and individual files. Although a SharedPreferences is used for account recovery, and several files are used to store an attachment and a cached rendering of it, the majority of the data is stored in SQLite.

5.3.2 Building the Object Graph

The object graph is the center of innovation in Pebbles: it directly represents Pebbles’s understanding of the structure of an app’s data and lets it construct LDOs. Each file, row, and XML element is assigned a 32 bit device-wide globally-unique ID (GUID) that is stored with the data item, which are hidden from and unmodifiable by applications. For database rows, the GUID is stored as an extra column in the row’s table; for XML, it is stored as an attribute of each element; and for files, it is stored in an extended attribute. When a row, element, or file is read, the data coming from it is “tainted” with its GUID and tracked in memory using a modified version of the TaintDroid taint tracking system [58].

Pebbles builds the object graph incrementally by adding new files/rows/XML elements as nodes into the graph as they are created. It also adds directed edges (called *relationships*) between nodes in the graph as they are discovered. For example, when data tainted with one GUID is written

into a file/row/XML element with another GUID, a relationship is registered. All nodes and edges of the graph are registered by the modified Android framework with the Pebbles Object Manager, where they are persisted in a database. We next describe the mechanisms used to build this graph, formalized in Fig. 5.5.

Data flow propagation relationships: It is easy to see a strawman approach to detecting relationships between objects: when Pebbles detects that data tainted with node A 's GUID is written into node B , it adds $A \leftrightarrow B$ to the object graph. This approach can capture all data flow relationships that occur within an application, regardless of the storage abstraction used. However, without precise information about the relationship between the two nodes, Pebbles is forced to assume the “worst case” scenario: that both nodes are part of the same LDO. Left unchecked, this so called taint explosion could eventually lead to all of an app's objects being included in the same LDO. Such behavior contradicts our primary goal of accurate and precise object recognition (G1). As we will see in §5.6.1, this naïve approach leads to unacceptably low precision (70%).

Utilizing explicit relationship information: Our next relationship detection mechanism relies on *explicit relationships* that directly communicate the programmer's view of his data structure to improve the precision. In a relational database, explicit relationships are defined in the form of foreign keys (FKs), which encode the precise relationship between two tables, based on primary keys (PKs). Interestingly, we can also extract a notion of foreign keys when relating DB rows to files: in some apps, the name of the file corresponds to the PK of the row to which it refers. Foreign keys encode the directionality of relationships, specifying for instance the difference between a “has-a” relationship and an “is-part-of” relationship. If node A has an FK to node B , then Pebbles adds the edge $A \rightarrow B$ (overriding any pre-existing bi-directional edge detected from data flow propagation). In this way, foreign keys are precise but limited in coverage because they require programmers to specify them explicitly.

Increasing recall: Pebbles relies on one final relationship detection mechanism, *access relationships*. Access relationships can be seen as similar to data relationships, but while data relationships identify relationships as they are written to storage, access relationships identify relationships as they are read. Consider the case where an application has some data in memory that has not been synced to stable storage (and therefore is not yet tainted with any node's GUID). The app uses the data to generate the index for key-value object A and also writes that data into database row B . In

Property 5.3.1. Apps define explicit relationships through FKs in DBs, XML hierarchies, or FS hierarchies

Property 5.3.2. The SQLite database is the hub of all persisted data storage and access

Object Graph Construction Algorithm:

1. Data propagation: If data from A is written to B , then $A \leftrightarrow B$
2. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$ using Prop 5.3.1
3. Access propagation: If data from A is used to read B , then $A \leftrightarrow B$
4. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$, again using Prop 5.3.1
5. Utilize Prop 5.3.2, eliminating access based data propagation relationships that do not include any DB nodes.

Fig. 5.5: Object Graph Construction Rules.

the absence of explicit relationship information, we would hope that data propagation would detect the relation; however, it cannot because there is no data flow relationship when the data is written. We call this situation a *parallel write*, and resolve it by detecting data flow relationships when data is read in from storage: if data tainted with node A 's GUID is used to access (read) node B , Pebbles adds $A \leftrightarrow B$ to the object graph. Again, this process is agnostic to the storage abstraction that the data is stored in, and relies only on data flow within the app. Access relationships can become an even greater source of imprecision than data relationships. For example, one could use data from one row, such as a timestamp, to select all the rows with that timestamp. Does that imply that all those rows should be considered as one object? Probably not.

Graph Generation Algorithm: Fig. 5.5 defines the algorithm used to construct the object graph, based on the observation that the DB is the hub of all persisted data. Step (1) leverages data flow propagation to construct a base graph, while (2) refines that graph by applying explicit relationship information. Step (3) applies access based data flow propagation to increase recall, and (4) again refines that graph with explicit relationship information. §5.6.1 evaluates LDO construction accuracy

Interface	Returned Objects
<code>getLDOContent (GUID, relevantOnly)</code>	LDO rooted at GUID
<code>getParentLDOs (GUID, relevantOnly)</code>	LDOs that contain GUID

Table 5.1: The Pebbles API for Accessing LDOs.

and precision in detail.

5.3.3 LDO Construction and Semantics

After constructing the object graph using the above semantics, Pebbles extracts the LDOs. Within the graph, an LDO is defined as the set of reachable nodes starting with a given node (the root of the object). Consider the email graph (Fig. 5.4), one can define a number of LDOs: an `Account` LDO, rooted in one `Account`-table row and containing multiple instances of five other row types, two files, and one XML entry; an `Email` LDO, rooted in one `Message`-table row and containing another row and one file, and so on. Although one LDO of each type is defined in the figure, in reality, there would be as many LDOs as there are instances of that type.

It is possible and correct for a single node to be part of multiple otherwise separate LDOs, in which case we say that the LDOs *overlap*. Consider, for instance, stateful accumulators (e.g. counts or sums over objects, stored in other objects), common resources (e.g. cache files that contain information about multiple objects), or log files.

Pebbles exposes LDOs to protection tools via the Pebbles API, which consists of two functions (Table 5.1). `getLDOContent` returns the LDO rooted at the given GUID and `getParentLDOs` returns the LDOs containing the given GUID. Protection tools may specify with each call if only LDOs that may be relevant to the end-user should be returned.

5.3.4 From User-Level Objects to LDOs

Both of these API methods require an “object of interest” as a parameter. Pebbles provides a framework for protection tools to allow users to directly select an object of interest (from the user interface), and then use that object for future API calls. In this approach, a user enables a “marking

mode” from a device-wide menu item, and then touches the item that they are interested in. Through taint tracking, we can determine the internal GUID for the object that was selected, and return that GUID back to the protection tool. This feature makes designing user-centric protection tools very easy: the tool need not concern itself with determining which objects to protect.

The mechanisms described thus far are useful for building a graph of all of an application’s objects, but does not yet include a way to identify those objects that are relevant to users. For instance, in our email application there is another table, “sync_state,” that stores how recently an account was synchronized with the server. Sync_state should clearly not be considered its own LDO, as its existence is essentially hidden from the end-user – the user will likely consider whatever data is stored here as, logically, part of the account. Pebbles leverages its system-wide taint tracking to identify which nodes in the object graph are directly displayed on the screen, Pebbles marks those objects (and other LDOs of the same type) as *relevant*. If an object is not relevant, then Pebbles will not allow it to be the root node of an LDO, instead including it as a member of the nearest parent node displayed on the screen.

5.4 Pebbles-based Tools

To showcase the value of Pebbles, we built four different applications that leverage its object graph.

5.4.1 Breadcrumbs: Auditing Object Deletion

Motivated by Scenario 1 in §5.1.1, Breadcrumbs lets users audit the deletion of their objects – such as emails or documents – by their applications. It uses Pebbles’s primitives to track objects as they are being deleted and identify any breadcrumbs left behind by the application.

Users mark objects to audit for deletion (using Pebbles’s object marking functionality), and then delete the object through their unmodified applications. They then open the Breadcrumbs application, which shows any persisted data related to recently tracked objects. In this way, users are not inundated with notifications about deletions and instead are only being presented with auditing information upon request. Fig. 5.6 shows a screenshot of Breadcrumbs’s output when the user deletes an email in the Android email application. It shows the attachment file left behind and provides meaningful information about the leakage. A brief predefined interval after the user deletes a

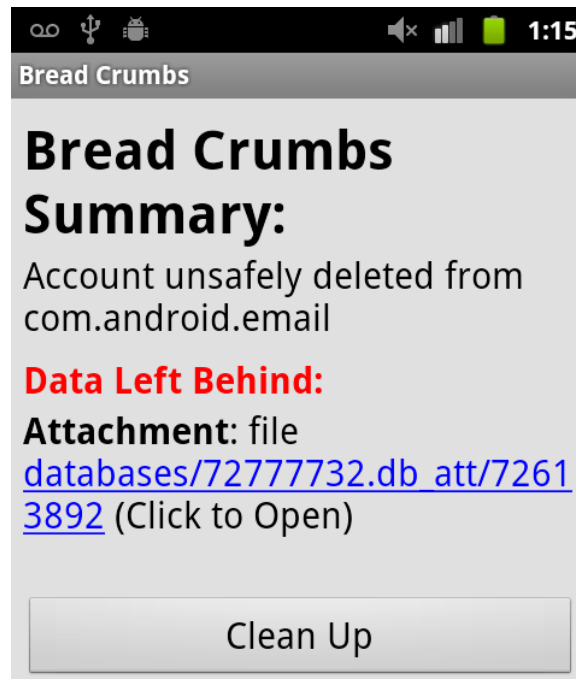


Fig. 5.6: Breadcrumbs.

tracked object, Breadcrumbs destroys all relevant auditing information to protect the confidentiality of the partially deleted object.

Algorithm 1 shows how Breadcrumbs uses Pebbles’s APIs to obtain all information necessary to identify and provide meaningful information about data left behind. Given a selected UI object, Pebbles identifies the GUID of the LDO represented by that LDO (as described in the previous section), and then Breadcrumbs calls `getLDOContent` to get all of its parts. For any part that still exists in persistent storage – the attachment file in this case – it displays meaningful metadata about that node. For example, instead of just showing the file’s path, which can be nondescript, Breadcrumbs uses Pebbles’s `getParentLDOs` function to retrieve the parent node, presumably a row. It displays the row’s table name (“Attachment” in Fig.5.6), providing more context for information left behind. While the specific user interface we chose for Breadcrumbs can be improved, this example underscores the great value protection tools like Breadcrumbs can draw from understanding application-level object structures.

Our evaluation of Breadcrumbs on 50 apps (§5.6.3), reveals that incomplete deletions are surprisingly common: 18/50 apps leave breadcrumbs or refuse to delete objects from the local device.

Algorithm 1 Breadcrumbs Pseudocode

```

function WASFULLYDELETED(LDO  $l$ )  $B \rightarrow$ 

  for all getLDOContent( $l$ ) as  $x$  do
    if  $x$  exists still then Add  $x \rightarrow B$ 
    end if
  end for

  for all  $B$  as  $x$  do
    Display  $x$  and getParentLDOs( $x$ ) to the user
  end for

end function

```

Breadcrumbs could also be a useful tool for developers. A developer could proactively use Breadcrumbs to ensure that they are responsibly handling their user’s data.

5.4.2 PebbleNotify: Tracking Object Exfiltration

Inspired by TaintDroid’s data exfiltration tool [58], we built PebbleNotify, a tool that tracks exfiltration at a more meaningful object level. TaintDroid reveals data exfiltration at a coarse granularity: it can only tell a user that *some* data from *some* provider was exfiltrated from the device, but not the specific data that was leaked. For instance, consider a cloud-based photo editing application. A user might expect this application to upload the photo being edited to a server for processing; however, he may be interested in checking that no other photos are exfiltrated. Shown in the left hand side of Fig.5.7, TaintDroid would warn the user that data related to *some* photo was uploaded, but not *which* photo or *how many* photos. PebbleNotify is a 500 line of code application built atop Pebbles that interposes on the same taint sinks as TaintDroid, but provides object-level warnings. §5.5 describes in somewhat greater detail the modifications that we made to TaintDroid to track individual objects with high precision. Shown in the right hand side of Fig.5.7, it leverages application-level data structures exposed by Pebbles to give users meaningful, fine-grained information about their leaked objects.

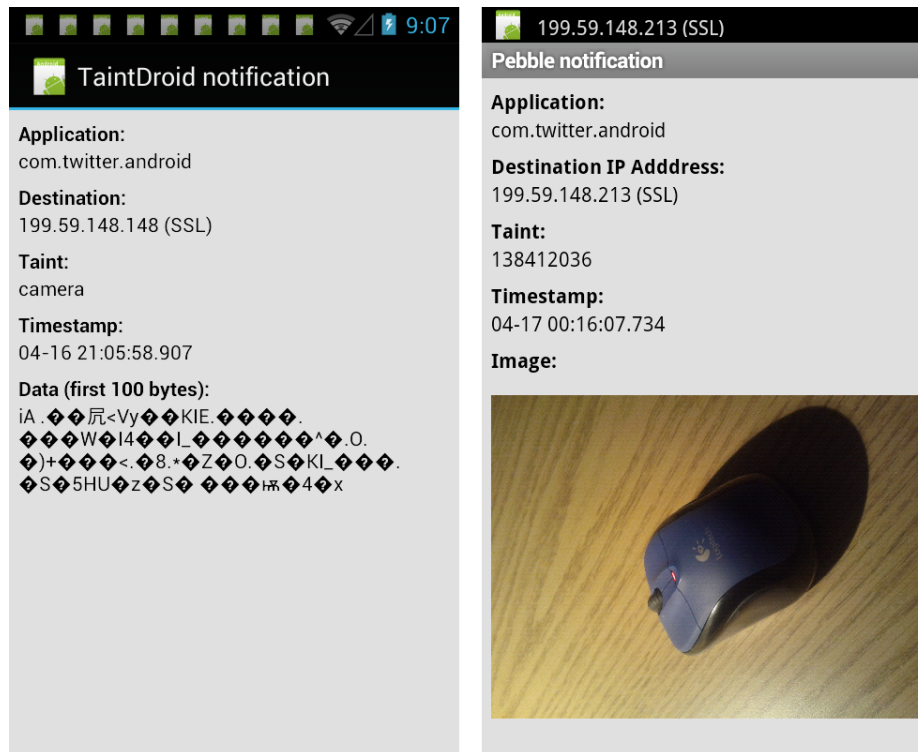


Fig. 5.7: Alert Screenshots. (L): TaintDroid, (R): PebbleNotify.

5.4.3 PebbleDIFC: Object Level Access Control

As a logical extension to PebbleNotify, consider the case where rather than monitor the exfiltration of sensitive data, users want to prevent specific apps from having access to it. For example, in our previous example of a user using a cloud-based photo editing application, perhaps the user would rather simply prevent that photo editing app from having any access whatsoever to sensitive photos. PebbleDIFC supports this use-case by interposing on Android content providers, the mechanism used to share data between apps.

PebbleDIFC allows users to select individual objects that are sensitive, and then prevent them from being shared with other applications (in this case, photos). As with the rest of our protection tools, PebbleDIFC's implementation is straightforward. Before returning an object from a content provider, PebbleDIFC checks a table that maps apps to hidden objects, and prevents access to hidden objects.

5.4.4 *HideIt*: Object Level Hiding

Whereas PebbleDIFC allows objects to be permanently hidden from specific apps, *HideIt* supports a slightly different use case: allowing objects to be selectively hidden from all apps on the device, and then redisplayed at some later point, and perhaps hidden again later on. When objects are hidden (again, using Pebbles’s marking mode), they are encrypted, and any record of their existence is filtered, by interposing on storage APIs. When objects are un-hidden, they are decrypted, and no longer filtered from API results. *HideIt* is intended for use-cases where small amounts of data need to be infrequently hidden from prying eyes, for instance, a parent lending their phone to their child.

5.4.5 Other Pebbles-based Tools

Although we designed and implemented Pebbles for Android, we believe that its object recognition mechanisms are applicable to other environments where a database is used as the hub of storage. In particular, we can imagine applying Pebbles as a software engineering tool to help developers understand either current or legacy applications where the database is the storage hub. A developer could use Pebbles to explore undocumented systems that do not make use of modern abstractions such as object relational mappers that would make the system easy to understand or to determine whether an application conforms to best practices and alert the developer if not. Understanding data structure from below the application could also enable testing tools and policy compliance auditing tools for cloud services [128]. We leave investigation of such applications for future work.

5.5 Implementation

We implemented Pebbles and each of the four above protection tools on Android 2.3.4 and TaintDroid 2.3.4. For Pebbles, we modify the SQLite, XML key/value store (a.k.a. *SharedPreferences*), and Java file system API to extract explicit structure, to intercept read/write/delete operations, and to register relationships. We also make several key changes to the TaintDroid tracking system, which we release as open source (<https://systems.cs.columbia.edu/projects/pebbles>). We next review our TaintDroid changes, after which we describe some implementation-level details of object graph creation.

TaintDroid Changes. To support Pebbles, we made three modifications to TaintDroid: (1) we increase the number of supported taints from 32 to several million, (2) we implement multi-tainting to allow objects to have an arbitrary number of taints simultaneously, and (3) we implement fine-grained tainting. The first two TaintDroid changes are necessary to track every row, file, and XML element with a separate taint and are implemented with a technique recently proposed in the context of another taint tracking system [111]. We omit the details here for space reasons.

The third TaintDroid change is motivated by massive taint explosion that we observed due to TaintDroid’s coarse-grained tracking. Specifically, TaintDroid stores a single taint tag per String and Array [58]. Deemed a performance benefit in the paper, this coarse-grained tracking is unusable in Pebbles: we observed extremely imprecise object recognition and application-wide LDOs due to this poor granularity. As one example, CWMoney, a personal finance application, has an internal array that holds selection arguments used in database queries. This causes all nodes selected by that query to be related, defeating any hopes of object precision.

To address this problem, we modify TaintDroid to add fine-grained tainting of individual Array and String elements. To implement fine-grained tainting we add a shadow buffer to the Dalvik ArrayObject that contains the taint of each element in the array. If implemented naively, the shadow arrays would likely double the memory required for each array. To minimize the memory overhead from the shadow arrays we allocate the shadow array only when a tainted element is inserted into the array. This same optimization is implemented in [47]. Intuitively, only a small fraction of arrays in an device’s memory should contain tainted elements (3-5% according to our evaluation). §5.6.2 shows that this lazy shadow array allocation significantly reduces the memory overhead of precise fine-grained tainting. We release our changes open source as a patch for TaintDroid.

Object Graph Implementation. The Pebbles graph is populated incrementally during application execution and persisted in a central database on the data partition so the graph does not need to be regenerated on each reboot. Applications interact with the Pebbles API through the Pebbles Object Manager that runs as part of the central system server process. Graph edges are generated on read and write operations to SQLite, shared preferences, and the file system. On read and write operations that generate new edges, requests for edge registration are placed on a queue within the application’s memory space. This lets Pebbles perform bulk asynchronous registrations off of the main application thread improving application interactivity even during periods of heavy edge

creation. In its current implementation the registration queue is not persisted to stable storage so it will be lost on application crashes or restarts. This is a potential attack vector that does not fall under the threat model for non-malicious applications.

5.6 Evaluation

We evaluate Pebbles over 50 popular applications downloaded from Google’s Android market on a Nexus S running our modified version of Android 2.3.4. We seek answers three key questions:

Q1 How accurate and precise is object identification in Pebbles?

Q2 What performance overhead does it introduce?

Q3 How useful are Pebbles and the tools running atop?

Application Workloads. We chose 50 test applications from the top free apps within 10 different Google Play Store categories, including Books and Reference, Finance, and Productivity. We looked at the top 30 most popular applications within each category (by number of installs) and selected those that used stable storage. We also added a few open-source applications (e.g., OINote). The resulting list included: Email (Android’s default email app), OINote (open-source note app), Browser (Android’s default), CWMoney (personal finance app), Bloomberg (stocks app), and PodcastAddict (podcast app). For each application, our workload involved exercising it in natural ways according to manual scripts. For example, in Wunderlist, a todo list app, we created multiple lists, added items to each list, and browsed through its functions.

5.6.1 Pebbles Precision and Recall (Q1)

We measure the precision and recall of our object recognition by identifying how closely LDOs match real, application-level objects as users perceive them. We manually identified 68 potentially interesting LDO types across 50 popular applications (e.g., individual emails, folders, and accounts in the default email app; individual expenses, expense categories, and accounts in the CWMoney financial app). We evaluated whether Pebbles correctly identifies those objects (no leakage or over-inclusions). Recall measures the percentage of LDOs recognized without leakage; precision measures the percentage of LDOs recognized without over-inclusion.

Application	LDO	Pebbles		File Tainting Only	
		Detected	Precise	Detected	Precise
Email	Account	Y	Y	Y	N
	Mailbox	Y	Y	Y	N
	Email	Y	Y	Y	N
OINote	Note	Y	Y	Y	N
Browser	History Item	Y	Y	Y	N
	Bookmark	Y	Y	Y	N
CWMoney	Account	Y	Y	Y	N
	Category	Y	Y	Y	N
	Expense	Y	Y	Y	N
Bloomberg	Stock	N	Y	Y	N
	Chart	Y	Y	Y	N
Podcast	Podcast	Y	Y	Y	N
	Episode	N	Y	Y	N
50 Total	68 Total	62/68 (91%)	66/68 (97%)	68/68 (100%)	0/68 (0%)

Table 5.2: LDO Precision and Recall. Sample applications and objects tested for object recognition precision and recall. “Y” indicates that an LDO was identified without leakage (column “Detected”) or without over inclusion (column “Precise”). If an LDO has “Y” in both columns, its recognition is deemed correct. As expected, Pebbles performs far better than a straw man approach of treating entire files as a single LDO.

To establish ground truth about LDO structure, we first populated the application with data and took a snapshot of the phone’s disk, S_1 , prior to creating the target object. Then, we created the object and took a second snapshot of the disk, S_2 . The ground truth is the diff between S_2 and S_1 after manually excluding differences that are unrelated to the objects (e.g., timestamps in log files that differ between the two executions). We then exercised the application as thoroughly as possible so as to capture any edges that Pebbles might detect. To measure accuracy, we compare Pebbles-recognized LDOs to the ground truth; if identical, we declare accurate recognition for that application and object.

Table 5.2 shows whether Pebbles correctly and precisely detects these LDOs. For comparison, we also evaluated the precision and recall of a basic approach, which represents perhaps the cur-

rent state of the art: detecting relationships between files using just taint tracking and not using additional file structure to refine the granularity of objects. Pebbles correctly identifies 60 of the 68 objects across these 50 apps, without requiring any program modifications. Of the eight incorrectly identified objects, six were not correctly detected and two were not precise.

In each case that Pebbles failed to properly detect all components of the object (i.e., where it failed in recall), the leakage was due to a non-standard database specification. For instance, in the case of the app “ColorfulBudget”, users can group expenses into categories, but Pebbles did not always properly detect the relationship between an expense and its category. Best practices would dictate that in such a case, all categories would be listed in a single table with a primary key (PK), and then each expense would contain a foreign key (FK) to reference the category’s PK [29]. Traditionally this PK is an integer, to significantly increase lookup speed and decrease the amount of space needed to store any references to it [29]. However, in its current implementation, this app uses the actual name of the category as a key into the category table, without declaring such a dependency. Therefore, if a new category is created simultaneously with the creation of a new expense, we will experience a parallel write: there will be no data dependence when the category is inserted and when the expense is inserted, since the category did not yet exist in storage. Moreover, since the relationship is not declared in the app schema as an FK, explicit relationship mechanism will not detect it.

While our access-based technique will largely eliminate this problem, there is still a gap when data is written but never read back. In these scenarios, such relationships could never be detected. Had these apps explicitly declared their DB relationships (e.g., in the above case by referencing each category by its PK), Pebbles would accurately recognized the objects.

As an example of Pebbles failing in precision (i.e., including additional objects as part of an LDO), consider the “Evernote” note taking app. Each time a notebook is updated, text in a Shared-Preferences node is updated to reflect the newest notebook, creating a data dependency between the SharedPreference and the notebook. In this way, each notebook can become related to each other because Pebbles currently does not break data dependencies when text is updated. The only way that relations are broken in Pebbles is if an explicit relationship exists and is removed.

Without requiring any modifications to applications, Pebbles is able to achieve up to 91% recall or 97% precision. The straw man approach of utilizing only taint tracking (without knowledge of

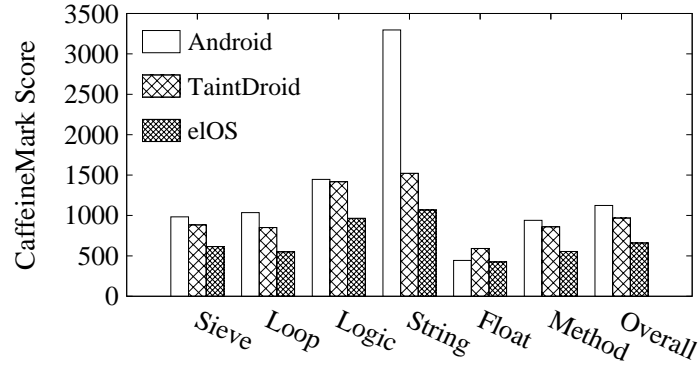


Fig. 5.8: Java Microbenchmarks. Overheads of the modified TaintDroid on the Java runtime with CaffeineMark, a standard Java benchmark. Higher values are better. Overheads on top of TaintDroid are 28-35%.

file structure) showed perfect recall (100%), and a complete failure in precision (0%). In other words, there were *no cases* of a single logical object stored in a single file. Overall, our results confirm that an unsupervised approach to application-level object recognition from within the OS works well, especially if schemas are relatively well-defined.

5.6.2 Performance Evaluation (Q2)

To evaluate Pebbles performance overheads, we ran two types of benchmarks: (1) *microbenchmarks*, which let us stress various components of our system, such as the computation and SQLite plugins; and (2) *macrobenchmarks*, which let us quantify our system’s performance impact on user-visible application latency. Pebbles is built atop the taint tracking system TaintDroid [58], with several modifications made to increase taint precision (as discussed in §5.5). Therefore, we evaluate the performance overhead of Pebbles in comparison to both TaintDroid and to a stock Android device.

Microbenchmarks. Our first experiments evaluate the overhead of Pebbles with the Java benchmark CaffeineMark 3.0 [113] and are shown in Fig. 5.8. We ran the six computational benchmarks and find that Pebbles decreases the score by 32% compared to TaintDroid, which itself decreases

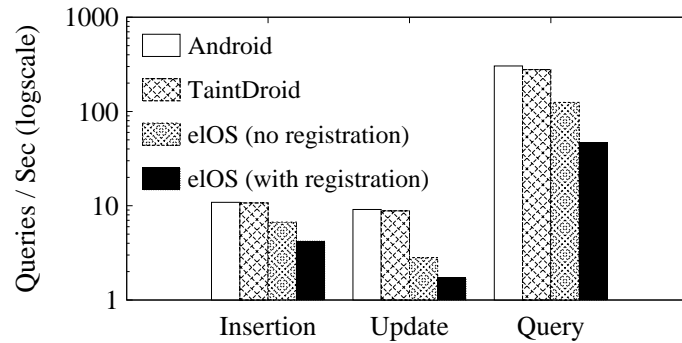


Fig. 5.9: SQLite Microbenchmarks. Overheads for various queries without and with relationship registrations.

the score by 16% compared to Android. The majority of this overhead comes from modifications to support more than 32 taints in Pebbles: TaintDroid combines tags by bitwise OR'ing, but Pebbles supports 2^{32} distinct taint markings, which are maintained in a lookup table. Pebbles also stores taint tags per individual array element, whereas TaintDroid stores only one taint tag per array, creating an additional overhead for Pebbles array-heavy benchmarks.

Pebbles also incorporates modifications to SQLite to detect and register relationships between rows with the Pebbles service. To evaluate the overhead, we compared the latency of simple, constant-size `SELECT`, `INSERT`, and `UPDATE` queries on an Pebbles-enabled Android versus Android. Fig. 5.9 shows query overheads when the query involves a relationship registration (59-168%) and when it does not (158-553%). No-registration queries – the cheapest to Pebbles – will likely be the common case for read-mostly workloads. For example, a document may be read many times, but relationship registration occurs only once. Moreover, batching and asynchronous-registration optimizations will likely help alleviate the overheads. The XML-based key/value store exhibits similar behavior, although we suppress concrete results.

Application-Level Performance. The above workloads are micro-benchmarks that stress the various components but do not necessarily relate to user-perceived performance impacts. To measure the impact of Pebbles on user-perceived interactivity, we evaluated the runtimes for various operations with three popular applications: Email, Browser and OINote. For Email, we look at app launch times and email reads; for Browser, we load the simple IANA homepage and the rich CNN and Google News pages over a local network; and for OINote we read a note. All network access

App	Activity	Base	TDroid	Pebbles	Overhead
Email	Launch	196.8	202.1	260.0	63.2 \pm 1.11
	Load Email	211.6	253.6	463.6	252.0 \pm 1.64
OINote	Launch	182.6	229.4	219.7	37.2 \pm 1.58
	Load Note	59.5	70.2	84.9	25.4 \pm 0.14
Browser	Launch	96.5	124.0	148.1	51.6 \pm 1.63
	Load (iana)	154.0	209.3	395.3	241.4 \pm 2.26
	Load (CNN)	778.9	862.7	1443.1	664.2 \pm 17.56
	Load (GNews)	951.3	1023.5	1311.2	359.9 \pm 10.75

Table 5.3: Application Performance. Operation runtimes and overheads in milliseconds. 95% confidence interval shown for overhead. Base is the Android baseline, TDroid is TaintDroid.

occurred over USB tethering to a host running a caching proxy; timing information excludes cache warmup. Table 5.3 shows the results in milliseconds. In almost all of the cases, overhead was less than 250ms. We saw more overhead and variation when rendering multimedia heavy web pages.

Memory Overheads. The modifications to TaintDroid to add fine grained tainting adds a memory overhead to the running system. We measure system wide memory usage while exercising three applications (Email, OINote, and Browser) with a similar workload as above. Without lazy memory allocation of array taint vectors (see §5.5), Pebbles’s system-wide memory overheads are high: 188MB, 70MB, and 119MB, respectively, compared to TaintDroid. With lazy memory allocation, Pebbles exhibits much lower system-wide overheads: 34MB, 16MB, and 29MB, respectively. Although still higher than TaintDroid’s own overhead of around 7MB for these applications, we believe Pebbles overheads are acceptable given devices’ increased memory trends.

5.6.3 Case Study Evaluation (Q3)

Breadcrumbs. Using our Breadcrumbs prototype we evaluated deletion practices of 68 types of LDOs across 50 applications. Of the 50 applications, 18 of them exhibited some type of deletion malpractice.

Table 5.4 shows sample deletion malpractice. There were several cases where data from one LDO was written into another another and not cleaned up later. There were also several applications

Application	Object Deletion Leakage
Email	Attachments remain after email/account deletion
ExpenseManager	Expenses remain after associated category deleted
Evernote	Notes/notebooks remain in database after deletion
On Track	Measurements remain after deleting category
14 other apps	21 LDO types unsafely deleted

Table 5.4: Breadcrumbs Findings. Shows samples of unsafe deletion in various applications.

that did not delete items at the users' request, instead simply removing them from the user interface. We observed this in applications that heavily rely on cloud storage such as Wunderlist, a popular cloud-backed todo list application.

PebbleNotify. To evaluate PebbleNotify, we compared its output to that of TaintDroid Notify. When TaintDroid Notify detects that data tainted with a value from one of the selected sources is exfiltrated, it notifies the user with the application that is responsible for the network connection, the destination, the data source, the timestamp, and the first 100 bytes of the packet. This is useful metadata but it won't help a user learn specific information about the data being exfiltrated such as which picture or specific contact is leaving the device. We found that PebbleNotify was more informative because it shows a summary of the data being exfiltrated, and not just the metadata presented by TaintDroid Notify. PebbleNotify was particularly useful in the case of image exfiltration because it displays a thumbnail of the image being sent.

PebbleDIFC. We integrated PebbleDIFC with the Android Media Provider and evaluated it by using it to mark several photographs on our device as sensitive (i.e., to prevent them from being shared). We then verified that those photos were not visible to applications other than the default Gallery application. We found that for this use case, PebbleDIFC has perfect accuracy: every photo that was marked was hidden, and no additional photos were hidden.

HideIt. We evaluated *HideIt* against many applications and largely found it to be effective. In our evaluation, we interacted with the application, populated it with data, and then marked a subset of the data as private so the application no longer had access. Interestingly, in most cases apps behaved as hoped when individual data objects were hidden and then again returned. There were however several cases where apps crashed when they expected some data to still exist, but was removed. We

are interested in performing further investigations of the applicability of *HideIt*.

5.6.4 Anecdotal User Experience

To gain experience with Pebbles, the primary author carried it on his Nexus S phone for about a week. He primarily used the Email, Browser, Gallery, Camera, and PodcastAddict apps. We report two anecdotal observations from this experience. First, applications exhibit noticeable overhead during periods of intense I/O, such as on initial launch or when applications populate or refresh local stores. During regular operation we observed overheads that are anecdotally similar to ones exhibited by running Android 4.1 (a 2012 OS) on our Nexus S (a 2010 device). Second, to check if object recognition remains accurate over time, we examined at the end of the week the structures of a sample of the objects in our applications (e.g., emails, folders, photos, browser histories, and podcasts). We saw no evidence that object recognition degraded over time due to taint explosions or other potential sources of imprecision for Pebbles. Objects grew naturally; email folders grew in size to include relevant new email objects and they remained accurate.

5.6.5 Summary

Overall, our results show that: Pebbles is quite accurate in constructing LDOs in an unsupervised manner (*Q1*), performance remains reasonable when doing so (*Q2*), and data management tools can benefit from Pebbles to provide useful, consumer-grade functions to the users (*Q3*). In our experience, Pebbles either consistently identifies objects of a particular type (e.g., all emails, all documents, etc.), or it does not. Whether it works depends largely upon the application's own adherence to some common practices (described in the next section). When Pebbles works for all object types of an application, Pebbles can provide the desired guarantees under our threat model. And even when Pebbles is incomplete, it can still support transparency applications, improving visibility into data (mis)management of applications. Our accuracy results show that Pebbles discovers all object types in 42 out of 50 applications correctly (no over-inclusions/leakages). We leave development of tools to identify whether an application matches the Pebbles assumptions for future work.

5.7 Discussion

Pebbles leverages the structure inherently present in the storage abstractions commonly used on Android to identify LDOs. More formally, Pebbles assumes the usage of the following best practices:

- R1:** *Declare database schemas in full:* Given that the database is becoming the central point of all storage in modern OSes, having a well-defined database schema is important and natural. 42/50 apps we have evaluated in §5.6.1 meet such requirements sufficiently for Pebbles to work perfectly for them.
- R2:** *Use the database to index data within other storage systems:* A common programming pattern is to create a parent object (e.g., a message) in the database, obtain an auto-generated primary key, and then write any children objects (such as message body, attachment files) using the PK as a link. 47/50 apps use this pattern. We strongly recommend it to any programmers who need to store data outside the DB.
- R3:** *Use standard storage libraries or implement Pebbles storage API:* To avoid precision lapses, we recommend that apps use standard storage abstractions. As §5.2 shows, most apps already adhere to this practice: most apps use exclusively OS-embedded abstractions.

Relative to our evaluation of 50 apps, 39/50 adhere with all three recommendations, and 50/50 adhere with at least one of them. Pebbles' performance could suffer for apps that do not follow any of these recommendations. However, we believe that each recommendation is sufficiently intuitive and rooted in best practices to not impose undue burden.

5.8 Related Work

Taint Tracking for Protection and Auditing. Taint tracking systems (such as [13, 44, 76, 104, 123, 158, 165]) implement a dynamic data flow analysis that has been applied to many different context such as privacy auditing [44, 58, 164], malware analysis [104], and more [13, 165]. TaintDroid [58] provides taint tracking of unmodified Android applications through a modified Dalvik VM, a system that Pebbles builds upon for its object graph construction. To our knowledge, Pebbles is the

first system to use taint tracking to discover data semantics of objects and provide a higher level abstraction with which to reason about and enforce such security properties.

Several systems utilize taint tracking to provide fine grained data protection and auditing. In each of these cases, however, a burden lies on the application developers to add hooks to identify relevant data structures to protection tool developers – a burden that could be lifted by Pebbles. For instance, CleanOS aims to minimize data exposure on a mobile device by automatically encrypting its “sensitive data objects” (SDOs) when not under active use [138]. The LDO abstraction is perhaps to some extent inspired by the SDO; however, SDOs must be manually specified by application developers, whereas LDOs are automatically identified and registered by Pebbles. Pebbles could be used to automatically identify SDOs, without requiring developer interaction.

Distributed information flow control (DIFC) systems such as Laminar [123], Asbestos [147], and Resin [158] let developers associate data with labels, and then allow either developers or end-users to specify security policies that apply to different labels. Taint tracking is performed during application execution to ensure that labels are propagated to derived data. Pebbles could be used to eliminate the need to statically annotate data with labels in code, instead automatically applying labels to LDOs as users request them. PebbleDIFC demonstrates the feasibility and power of such a system.

Related to taint tracking, data provenance [101, 102, 127] is close in spirit to logical data objects. It tracks the lineage of data (e.g., the user or process that created it). It has been proposed to identify the original authors of online information, to facilitate reproduction of scientific experiments [127], detect and avoid faulty data propagation in clouds [102], and others. It has to our knowledge never been used as an OS protection abstraction.

Fine-Grained Protection in Operating Systems. Many systems have been proposed in the past to support fine-grained, flexible protection in operating systems. Some of the earliest OSes, such as Hydra [154] and Multics [124], provided immense protection flexibility to applications and users. Over time, OSes removed more and more flexibility, being considered too difficult for programmers. Our goal is to eliminate the programmer from the loop by having the OS identifying objects.

More recently, OS security extension systems, such as SELinux [126] and its Android version, SEAndroid [125], extend Linux’s access control with flexible policies that determine which users and processes can access which resources, such as files, network interfaces, etc. Our work is com-

plementary to these, being concerned with external attacks, such as thieves, shoulder surfing, or spying by a user with whom the device has been willfully shared. Our abstractions, might, however, apply to SEAndroid to replace its antiquated file abstraction.

Securing and Hiding Data. Many encryption systems exist, operating largely at one of two levels of abstraction: block level [6, 98, 145] and file level [69, 75]. A drawback to such encrypted file systems is that it forces users to consider data as individual files, while logically there may be multiple objects that the user is interested in in a single file. Pebbles allows protection tool developers to provide a far finer level of control (at the object level) than these existing systems (at the file level).

Some protection tools are already operating at a higher level of data abstraction. These applications, such as Vault-Hide [107] and KeepSafe Vault [89], allow users to hide specific types of data, including photos, contacts, and SMSes. However, they only plug into a handful of supported apps and cannot provide generic protection for all apps. Pebbles aims to effect a similar level of control, but without requiring specialized work by protection tool developers to support specific applications.

Inferring Structure in Semistructured Data. Discovering data relationships is a key aspect of our work. Other have worked on inferring data relationships in various context: foreign key relationships in databases to improve querying [118, 162] and file relationships in OSes to enhance file search [131]. However, Pebbles can also infer relations among files, as well as other higher-level storage abstractions within modern operating systems. To perform such broad relationship detection, Pebbles differs significantly from other relationship detection systems in that it also leverages taint tracking.

Cozzie et al. developed the Laika system [48] which uses Bayesian analysis to infer data structures from memory images. Pebbles differs from Laika in that it does not attempt to recover programmer defined data structures but to discover application-level data relationships from stable storage that would be recognizable and useful to an end user or developer.

5.9 Conclusions

We have described *logical data objects* (LDOs), a new fine-grained protection abstraction for persistent data designed specifically to enable the development of protection tools at a new granularity.

We described our implementation of LDOs for Android with *Pebbles*, a system that automatically reverse engineers LDOs from application-level persisted data resources – such as emails, documents, or bank accounts. Pebbles leverages the structural semantics available in modern persistent storage systems, together with a number of mechanisms rooted in taint tracking, to construct and maintain an object graph that tracks these LDOs without introducing any new programming models or APIs.

We have evaluated Pebbles and four novel protection tools that use it, showing it to be accurate, and sufficiently efficient to be used in practice to identify and manage LDOs. We can envision many other useful applications of Pebbles, such as data scrubbing or malware analysis, and hope that LDOs will enable the development of these and other granular data protection systems.

Chapter 6

Future Work

6.1 Stable tests and stable builds

My thesis work in accelerating and stabilizing testing and builds led to my interest in the general problem of flaky tests. Recent studies have shown that tests may behave erratically (i.e., become flaky) for many reasons, not just due to the violated test order dependencies that I have looked at [95]. For instance, tests may have poorly managed synchronization (e.g., a test that starts a server, and uses a timed wait to wait for the server to come back), external dependencies (e.g., on outside services that do not have mocks), or general non-determinism. I believe that our software testing and building infrastructure should rise to the task to automatically determine when a test is flaky for one of these reasons, and - if possible, automatically patch around the solution. For instance, using my existing work, I can augment a testing infrastructure to detect the ordering dependencies between tests, and then ensure that they are always enforced. I am interested in exploring techniques for automatically repairing tests that are flaky. One approach would be to build on my prior work in lightweight record-and-replay systems (Chronicler [24]), which could be used to isolate external factors that cause tests to become flaky. By helping developers run their tests faster and more reliably, we can decrease the cost of test execution, making it easier to run tests more often and find bugs faster.

6.2 Supporting debugging and program understanding

I believe that software debugging tools can be greatly improved to support developers, making it easier to pinpoint faults and understand how code works. While much prior literature on debugging focus on purely automated techniques for debugging, I am excited to examine the potential for building supportive approaches: those that serve to help (but not completely automate) human debugging tasks.

For example, while taint tracking is typically applied in the context of security and privacy tools, it can be an incredibly powerful tool when used in the context of debugging and program understanding. I plan to integrate my existing taint tracking system for Java, PHOSPHOR [19, 21] with existing debugging tools (e.g., the Eclipse debugger) to allow developers to determine precisely what code statements effect any specific value. Previous work towards solving this problem is typically based on dynamic program slicing: where complete program dynamic dependency graphs are captured, and then analyzed to detect the relevant slice of the program that affects a given variable. This approach can be unwieldy on large and long running programs — the “slice” typically includes all instructions that effect a variable *plus* all of those that are depended on (e.g., through control dependencies) by those instructions. In contrast, I propose a significantly more lightweight analysis, which will answer only the question of “where was this variable set,” but will answer this question efficiently. Such a system will show only statements that the variable depends on through *data flow dependencies*, which will allow it to be more scalable to deploy than a traditional slicing system that includes control dependencies as well. I believe that by building on Phosphor, I can create a new approach to debugging, and prove its usefulness in actual deployment environments, a first step towards my broader agenda of creating better debugging and program understanding tools.

6.3 Runtime monitoring of deployed software

Another key challenge for software developers is monitoring software once it’s deployed: collecting performance and usage metrics, crashes, etc. My current interest in this area is specifically related to helping developers effectively reproduce production crashes in the lab — an area that I have touched on in my early work on record and replay for Java (Chronicler, [24]). Two main limitations for systems that record program executions in the field for replay in the lab are privacy and performance:

these systems often record sensitive data, and recording sufficient data to reproduce the failure successfully tends to be heavyweight. My first steps in this direction build in a new direction off of PHOSPHOR, using it to track path constraints on variables through a program execution.

Path constraints represent the symbolic constraints that are applied to programs during a concrete execution, for instance $x + 2 > 5$, $y \neq \text{null}$, or $\text{str} = \text{"foo"}$. While recording constraints can be relatively heavyweight, it tends to be very easy to determine if a given input satisfies a set of constraints. I have extended PHOSPHOR to track path constraints on variables, creating KNARR, which is significantly more performant and portable than previous approaches (e.g., KLEE requires sourcecode, JPF is designed for modeling small Java programs and does not support many real-world Java programs), enabling new research directions.

One potential project will use KNARR to record constraints on application inputs during a profiling phase, and then check inputs against this constraint cache at runtime to determine if the inputs will exercise new program behavior. If so, additional monitoring can be performed (e.g., security checks, logging, input recording, etc.), and, if not, execution can proceed knowing that the program is unlikely to crash or behave differently than it did previously with similar inputs. A logical extension would be to consider program optimizations that can be performed if we know that an input matches a given profile. Such an approach can work towards the overall goal of increasing confidence of deployed software, as developers can monitor when applications see similar (or dissimilar) inputs, and use this information as feedback for testing cycles.

6.4 Build Acceleration

In our previous work, we determined that build times of Java projects are often dominated by the execution of tests, and presented two approaches to significantly reduce the time necessary to run entire test suites by cutting time spent isolating tests, and safely parallelizing them.

We have discovered a new way to accelerate the testing phase of builds (specifically, Java builds performed with Maven) even further by introducing a new level of parallelism, allowing tests to begin executing before projects are fully built without introducing a risk of build failure or non-determinism. While there have been attempts to support fully parallel Java builds (e.g., Maven's `-T` feature), realizing high degrees of parallelism remains challenging due to difficult-to-break depen-

Project	Build Time (mins)	Testing Time	Modules w/ Tests
titan	380.77	94.99%	13/15
camel	359.57	84.68%	195/271
mule	198.87	92.81%	57/72
spring-data-mongodb	123.17	99.32%	3/3
cdap	110.62	97.15%	19/33
hadoop	108.03	97.78%	27/36
opennms	120.73	76.89%	122/220
ks-final-milestone	124.23	71.08%	17/46
mongo-java-driver	74.92	99.35%	1/1
netty	67.63	92.24%	16/19

Table 6.1: Testing statistics for large maven projects. Shows the build time, percent of build time running tests, and the number of maven modules of each project that have tests.

dencies. Maven is a modular build system, allowing projects to consist of multiple sub-modules, each of which may depend on other modules. When a multi-module Maven project is built, each module goes through the entire Maven build lifecycle (compile, test, package, etc.)

We measured the amount of time spent running tests in 10 large open source java projects (building on Amazon’s EC2 with m3.medium instances), finding that tests are often distributed among many modules. Table 6.1 shows the results of this preliminary study: most projects have many modules, and many of those projects have tests.

We have observed that while one module may depend on the code or other artifacts generated by a previous module, they do not rely on the execution of the tests of a prior module. However, due to Maven’s modular nature, it is impossible to specify that a single component of a module (for instance, its tests) should run in parallel with other modules. Therefore, we have built a plugin for test execution in Maven that *delays* the dependency between each module’s test phase, allowing tests to execute in parallel to the build of other modules. This way, Maven considers individual modules as fully built (for the purpose of dependency resolution) even if that module’s tests haven’t finished running yet. Since the dependence is delayed (rather than dropped), Maven still executes the tests for each module, and won’t consider the build overall complete until the tests finish executing. This

approach requires making no modifications to Maven, instead functioning solely as an additional plugin, ensuring that it remains portable across new versions of Maven.

We applied this technique to build a proprietary, internal system that had previously taken 20 minutes to build, even when utilizing Maven's provided parallelism features. After applying our delayed dependency technique, the system took only 8 minutes to build, using the same number of processor cores as Maven's automatic parallelism provided. We are continuing to refine this prototype system and search for other sorts of limitations that build systems might have that limit their performance.

Chapter 7

Conclusions

Advancement in software testing, debugging and tools that support programmer understanding of code are often limited by an invisible wall between the software engineering, software systems, and programming language communities. To make a major contribution in this area requires an ability to both empirically analyze the current challenges that developers face, and the ability to construct new and novel tools and analyses to support these challenges. I believe that the strongest advances in software engineering come from new approaches that leverage both semi-automated tooling and developer insights and feedback, rather than purely automated tools.

In this thesis, I have presented four techniques and their concrete system implementations that all contribute to making it easier for developers to produce reliable software. I've described how program dependency analysis can be an interesting and sometimes overlooked way of approaching software reliability. For instance, I described how testing dominates the overall time needed to build software, which can make it hard to run tests as often as developers might need, making it harder to build reliable software.

I described my first insight towards accelerating testing, removing the need to execute individual test cases in their own process (VMVM, Chapter 2, with my lightweight isolation mechanism. I also considered situations where test cases are not isolated, where it is not safe to use off-the-shelf test acceleration techniques, describing `ELECTRICTEST`, which efficiently detects these dependencies between tests (Chapter 3). Finally, I described my dynamic taint tracking system, `PHOSPHOR` (Chapter 4), a system that leverages taint tracking to make it easier for developers to specify their tests, `PEBBLES` (Chapter 5).

While I am interested in the broad research directions described in Chapter 6, I also look forward to new collaborations and continuing to broaden my research horizons.

Bibliography

Bibliography

- [1] Cookiesbasetest.java. <http://svn.apache.org/repos/asf/tomcat/trunk/test/org/apache/tomcat/util/http/CookiesBaseTest.java>.
- [2] Dependency and data driven unit testing framework for java. <https://code.google.com/p/depunit/>.
- [3] Junit: A programmer-oriented testing framework for java. <http://junit.org/>.
- [4] Next generation java testing. <http://testng.org/doc/index.html>.
- [5] Ohloh, inc. <http://www.ohloh.net>.
- [6] dm-crypt: Linux kernel device-mapper crypto target. <https://code.google.com/p/cryptsetup/wiki/DMCrypt>, 2013.
- [7] Anand Basu. Facebook Apps Leak User Information. <http://www.reuters.com/article/2010/10/18/us-facebook-idUSTRE69H0QS20101018>, 2010.
- [8] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11*, pages 355–366, New York, NY, USA, 2011. ACM.
- [9] Apache Software Foundation. The apache ant project. <http://ant.apache.org/>.
- [10] Apache Software Foundation. Apache harmony - open source java platform. <http://harmony.apache.org>.

- [11] Apache Software Foundation. The apache maven project. <http://maven.apache.org/>.
- [12] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.
- [13] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.
- [14] Mohammad Reza Azadmanesh and Mohsen Sharifi. Towards a system-wide and transparent security mechanism using language-level information flow control. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, SIN '10, pages 19–26, New York, NY, USA, 2010. ACM.
- [15] Thomas Ball. On the limit of control flow analysis for regression test selection. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 134–142, New York, NY, USA, 1998. ACM.
- [16] Jonathan Bell and Gail Kaiser. Phosphor: Dynamic taint tracking for the jvm. <https://github.com/Programming-Systems-Lab/phosphor>.
- [17] Jonathan Bell and Gail Kaiser. Vmvm: Unit test virtualization in java. <https://github.com/Programming-Systems-Lab/vmvm>.
- [18] Jonathan Bell and Gail Kaiser. Unit test virtualization with vmvm. Technical Report CUCS-021-13, Columbia University Dept of Computer Science, <http://mice.cs.columbia.edu/getTechreport.php?techreportID=1549&format=pdf>, September 2013.
- [19] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *OOPSLA*, 2014.

- [20] Jonathan Bell and Gail Kaiser. Unit Test Virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering, ICSE '14*, pages 550–561, New York, NY, USA, 2014. ACM.
- [21] Jonathan Bell and Gail Kaiser. Dynamic taint tracking for java with phosphor (demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015*, pages 409–413, New York, NY, USA, 2015. ACM.
- [22] Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 770–781, New York, NY, USA, 2015. ACM.
- [23] Jonathan Bell, Eric Melski, Mohan Dattatreya, and Gail Kaiser. Vroom: Faster Build Processes for Java. In *IEEE Software Special Issue: Release Engineering*. IEEE Computer Society, March/April 2015. To Appear. Preprint: <http://jonbell.net/s2bel.pdf>.
- [24] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press.
- [25] Jennifer Black, Emanuel Melachrinoudis, and David Kaeli. Bi-criteria models for all-uses test suite reduction. In *Proceedings of the 26th International Conference on Software Engineering, ICSE '04*, pages 106–115, Washington, DC, USA, 2004. IEEE Computer Society.
- [26] Black Duck Software. Black duck unveils ohloh open data initiative, launches beta code search capability. <http://www.blackducksoftware.com/news/releases/2012-07-18>.
- [27] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*, pages 169–190, New York, NY, USA, 2006. ACM.

- [28] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under unix. *ACM Trans. Comput. Syst.*, 7(1):1–24, January 1989.
- [29] Michael Brackett. *Data Resource Design: Reality Beyond Illusion*. IT Pro. Technics Publications Llc, 2012.
- [30] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Inf. Softw. Technol.*, 51(1):16–30, January 2009.
- [31] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
- [32] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking java grande applications. In *in Proceedings of ACM 1999 Java Grande Conference*, pages 81–88. ACM Press, 1999.
- [33] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot: A technique for cheap recovery. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6, OSDI’04*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.
- [34] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 463–475, Dec 2007.
- [35] Deepak Chandra. Personal Communication (Email). July 10, 2014.
- [36] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.
- [37] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS ’08*, pages 39–50, New York, NY, USA, 2008. ACM.
- [38] T.Y. Chen and M.F. Lau. A new heuristic for test suite reduction. *Information and Software Technology*, 40(5–6):347 – 354, 1998.

- [39] T.Y. Chen and M.F. Lau. A simulation study on some heuristics for test suite reduction. *Information and Software Technology*, 40(13):777 – 787, 1998.
- [40] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, Washington, DC, USA, 2006. IEEE.
- [41] Monica Chew. Writing for the 98%, blog post. <http://monica-at-mozilla.blogspot.com/2013/02/writing-for-98.html>, 2013.
- [42] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, SWS '09. ACM, 2009.
- [43] Ge-Ming Chiu and Cheng-Ru Young. Efficient rollback-recovery technique in distributed computing systems. *IEEE Trans. Parallel Distrib. Syst.*, 7(6):565–577, June 1996.
- [44] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium (Sec)*, 2004.
- [45] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium (Sec)*, 2005.
- [46] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07*. ACM, 2007.
- [47] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. Spandex: Secure password tracking for android. In *Proceedings of the USENIX Security Symposium (Sec)*, 2014.
- [48] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
- [49] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

- [50] CVE Details. Vulnerability distribution of cve security vulnerabilities by types. <http://www.cvedetails.com/vulnerabilities-by-types.php>.
- [51] Dex2Jar Project. dex2jar - tools to work with android .dex and java .class files - google project hosting. <https://code.google.com/p/dex2jar/>.
- [52] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.
- [53] Hyunsook Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a junit testing environment. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 113–124, 2004.
- [54] Alan M. Dunn, Michael Z. Lee, Suman Jana, Sangman Kim, Mark Silberstein, Yuanzhong Xu, Vitaly Shmatikov, and Emmett Witchel. Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [55] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE '01*, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.
- [56] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, May 1992.
- [57] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI'10*, Berkeley, CA, USA, 2010. USENIX Association.
- [58] William Enck, Peter Gilbert, Byung-gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An information-flow tracking system for realtime

- privacy monitoring on smartphones. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.
- [59] Martin Fowler. Eradicating non-determinism in tests. <http://martinfowler.com/articles/nonDeterminism.html>, 2011.
- [60] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *FSE '12*, pages 46:1–46:11, New York, NY, USA, 2012. ACM.
- [61] Roxana Geambasu, John P. John, Steven D. Gribble, Tadayoshi Kohno, and Henry M. Levy. Keypad: An auditing file system for theft-prone devices. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.
- [62] Erol Gelenbe. A model of roll-back recovery with multiple checkpoints. In *Proceedings of the 2nd international conference on Software engineering, ICSE '76*, pages 251–255, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.
- [63] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA '07*, pages 57–76, New York, NY, USA, 2007. ACM.
- [64] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. Regression test selection for distributed software histories. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 293–309. Springer International Publishing, 2014.
- [65] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14*, pages 361–372, New York, NY, USA, 2014. ACM.
- [66] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and

- search-based refactoring. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, pages 599–616, New York, NY, USA, 2014. ACM.
- [67] Eugene Gluzberg and Stephen Fink. An evaluation of java system services with microbenchmarks. Technical report, 2000.
- [68] Google. Storage options — android developers. <http://developer.android.com/guide/topics/data/data-storage.html>.
- [69] Valient Gough. encfs. www.arg0.net/encfs, 2010.
- [70] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001.
- [71] GRSecurity. Homepage of pax. <http://pax.grsecurity.net/>.
- [72] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *ISSTA '11*, New York, NY, USA, 2011. ACM.
- [73] Alex Gyori, August Shi, Farah Hairi, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 ACM International Symposium on Software Testing and Analysis*, 2015.
- [74] S. Haidry and T. Miller. Using dependency structures for prioritization of functional test suites. *Software Engineering, IEEE Transactions on*, 39(2):258–275, Feb 2013.
- [75] Michael Austin Halcrow. eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proceedings of the Linux Symposium*, 2005.
- [76] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference, ACSAC '05*, pages 303–311, Washington, DC, USA, 2005. IEEE Computer Society.

- [77] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14*, pages 175–185, New York, NY, USA, 2006. ACM.
- [78] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 738–748, Piscataway, NJ, USA, 2012. IEEE Press.
- [79] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.
- [80] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 312–326, New York, NY, USA, 2001. ACM.
- [81] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2011.
- [82] Reid Holmes and David Notkin. Identifying program, test, and environmental changes that affect behaviour. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 371–380, New York, NY, USA, 2011. ACM.
- [83] Hwa-You Hsu and Alessandro Orso. Mints: A general framework and tool for supporting test-suite minimization. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 419–429, Washington, DC, USA, 2009. IEEE Computer Society.
- [84] Jeff Huang, Peng Liu, and Charles Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 385–386, New York, NY, USA, 2010. ACM.

- [85] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, 2014.
- [86] Shvetank Jain, Fareha Shafique, Vladan Djeriç, and Ashvin Goel. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 95–107, New York, NY, USA, 2008. ACM.
- [87] Dennis Jeffrey and Neelam Gupta. Improving fault detection capability by selectively retaining test cases during test suite reduction. *IEEE Trans. Softw. Eng.*, 33(2):108–123, February 2007.
- [88] James A. Jones and Mary Jean Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Softw. Eng.*, 29(3):195–209, March 2003.
- [89] KeepSafe. Hide pictures - KeepSafe Vault. <https://play.google.com/store/apps/details?id=com.kii.safe>.
- [90] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.
- [91] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, Washington, DC, USA, 2006. IEEE.
- [92] T. R. Leek, G. Z. Baker, R. E. Brown, M. A. Zhivich, and R. P. Lippmann. Coverage maximization using dynamic taint tracing. Technical Report TR-1112, MIT Lincoln Lab, 2007.
- [93] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *Transactions on Information and System Security (TISSEC)*, 12(3):14:1–14:37, January 2009.
- [94] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*, Java SE 7 edition, Feb 2013.

- [95] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 643–653, New York, NY, USA, 2014. ACM.
- [96] Mary Madden and Aaron Smith. Reputation management and social media: How people monitor their identity and search for others online. http://www.pewinternet.org/~media/Files/Reports/2010/PIP_Reputation_Management_with_topline.pdf, 2010.
- [97] Atif M. Memon and Myra B. Cohen. Automated testing of gui applications: Models, tools, and controlling flakiness. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1479–1480, Piscataway, NJ, USA, 2013. IEEE Press.
- [98] Microsoft Corporation. Windows 7 BitLocker executive overview. [http://technet.microsoft.com/en-us/library/dd548341\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/dd548341(WS.10).aspx), 2009.
- [99] Matteo Migliavacca, Ioannis Papagiannis, David M. Eysers, Brian Shand, Jean Bacon, and Peter Pietzuch. Defcon: High-performance event processing with information security. In *Proceedings of the 2010 USENIX ATC*, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.
- [100] Kivanç Muşlu, Bilge Soran, and Jochen Wuttke. Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 496–499, New York, NY, USA, 2011. ACM.
- [101] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2006.
- [102] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

- [103] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16, February 2008.
- [104] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2005.
- [105] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In Ryôichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura, editors, *SEC*, pages 295–308. Springer, 2005.
- [106] Vladimir Nikolov, Rüdiger Kapitza, and Franz J. Hauck. Recoverable class loaders for a fast restart of java applications. *Mobile Networks and Applications*, 14(1):53–64, February 2009.
- [107] NQ Mobile Security. Vault-Hide SMS, Pics & Videos. <https://play.google.com/store/apps/details?id=com.netqin.ps>.
- [108] Oracle. Jvm tool interface. <http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html>.
- [109] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, SIGSOFT '04/FSE-12*, pages 241–251, New York, NY, USA, 2004. ACM.
- [110] John Ousterhout. 10–20x faster software builds. *USENIX ATC*, 2005.
- [111] Vasilis Pappas, Vasileios P. Kemerlis, Angeliki Zavou, Michalis Polychronakis, and Angelos D. Keromytis. CloudFence: Data flow tracking as a cloud service. In *Proceedings of the Symposium on Research in Attacks, Intrusions and Defenses*, 2013.
- [112] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, VEE '11*, pages 157–168, New York, NY, USA, 2011. ACM.

- [113] Pendragon Software Corporation. Caffeinemark 3.0. <http://www.benchmarkhq.ru/cm30/>.
- [114] Pendragon Software Corporation. Caffeinemark 3.0. <http://www.benchmarkhq.ru/cm30/>, 1997.
- [115] Radia Perlman. File system design with assured delete. In *Proceedings of the IEEE International Security in Storage Workshop (SISW)*, 2005.
- [116] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 33:1–33:11, New York, NY, USA, 2012. ACM.
- [117] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the USENIX Security Symposium (Sec)*, 2012.
- [118] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *Proceedings of the International Workshop on the Web and Databases (WebDB)*, 2009.
- [119] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '99)*, pages 179–188, 1999.
- [120] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–441, August 1996.
- [121] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43.
- [122] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *PLDI '09*, pages 63–74, New York, NY, USA, 2009. ACM.

- [123] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.
- [124] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM (CACM)*, 1974.
- [125] SEAndroid. SEforAndroid. <http://selinuxproject.org/page/SEAndroid>.
- [126] SELinux. Selinux project wiki. http://selinuxproject.org/page/Main_Page.
- [127] Margo Seltzer. Pass: Provenance-aware storage systems. <http://www.eecs.harvard.edu/syrah/pass/>.
- [128] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [129] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. In *OOPSLA '11*, pages 657–676, New York, NY, USA, 2011. ACM.
- [130] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *CCS '13*, New York, NY, USA, 2013. ACM.
- [131] Craig A.N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005.
- [132] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *OSDI*, 2014.
- [133] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *OOPSLA '11*. ACM, 2011.

- [134] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '02, pages 97–106, New York, NY, USA, 2002. ACM.
- [135] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, pages 85–96, New York, NY, USA, 2004. ACM.
- [136] Symantec Corporation. PGP whole disk encryption. <http://www.symantec.com/whole-disk-encryption>, 2012.
- [137] Sriraman Tallam and Neelam Gupta. A concept analysis inspired greedy algorithm for test suite minimization. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 35–42, New York, NY, USA, 2005. ACM.
- [138] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. CleanOS: Mobile OS abstractions for managing sensitive data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [139] Yang Tang, Patrick P.C. Lee, John C.S. Lui, and Radia Perlman. FADE: Secure overlay cloud storage with file assured deletion. In *Proceedings of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.
- [140] The Chaos Computing Club (CCC). CCC breaks Apple TouchID. <http://www.ccc.de/en/updates/2013/ccc-breaks-apple-touchid>, 2013.
- [141] The Jikes RVM Project. Jikes rvm - project status. <http://jikesrvm.org/Project+Status>.
- [142] The Kaffe Team. Kaffe vm. <https://github.com/kaffe/kaffe>.
- [143] John Toman and Dan Grossman. Staccato: A bug finder for dynamic configuration updates. ECOOP, 2016.

- [144] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *PLDI '09*, pages 87–97, New York, NY, USA, 2009. ACM.
- [145] TrueCrypt Foundation. Truecrypt – free open-source on-the-fly encryption. <http://www.truecrypt.org/>, 2007.
- [146] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4), December 2007.
- [147] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems (TOCS)*, 2007.
- [148] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*, 2014.
- [149] Matej Vításek, Walter Binder, and Matthias Hauswirth. Shadowdata: Shadowing heap objects in java. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 17–24, New York, NY, USA, 2013. ACM.
- [150] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013*. ACM, 2013.
- [151] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, ISSRE '97, Washington, DC, USA, 1997. IEEE Computer Society.
- [152] W. Eric Wong, Joseph R. Horgan, Saul London, and Aditya P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

- [153] W.E. Wong, J.R. Horgan, A.P. Mathur, and A. Pasquini. Test set size minimization and fault detection effectiveness: a case study in a space application. In *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, pages 522–528, 1997.
- [154] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM (CACM)*, 1974.
- [155] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07*, pages 85–94, New York, NY, USA, 2007. ACM.
- [156] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.
- [157] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *SOSP '09*, pages 291–304, New York, NY, USA, 2009. ACM.
- [158] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2009.
- [159] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, March 2012.
- [160] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

- [161] Lingming Zhang, D. Marinov, Lu Zhang, and S Khurshid. An empirical study of junit test-suite reduction. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 170–179, 2011.
- [162] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*, 3(1-2):805–814, 2010.
- [163] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanc Muslu, Michael Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA '14*, pages 384–396, New York, NY, USA, 2014. ACM.
- [164] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.
- [165] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 2011.

Appendices

JVM Bytecode Opcode Reference

PHOSPHOR modifies the operation of many bytecode instructions by inserting additional instructions around them. This table lists all bytecode instructions supported by the Java Virtual machine, and for each one, a brief description of the change(s) that PHOSPHOR makes.

Table 1: All JVM bytecodes, annotated with descriptive transformation information

Opcode(s)	Brief Description	PHOSPHOR Modifications
aastore	Stores reference to array	Removes the taint tag for the index before storing; if the ref. is to a primitive array, boxes before storing
aaload	Loads reference to array	Removes the taint tag for the index to load
anewarray	Allocates new array for references	If the array type is a mutli-d primitive array, change to a container type
arraylength	Returns length of array as integer	Place the tag "0" just below return val on the operand stack after execution
areturn	Exit a method, returning the object reference at the top of the stack	If the top of the stack is a primitive array, boxes the array and its taint tags before return

Table 1: All JVM bytecodes, annotated with descriptive transformation information

Opcode(s)	Brief Description	PHOSPHOR Modifications
astore	Store an object to a local variable	If the variable type is a primitive array, store the taint tags also to their variable. If the variable type is "Object" and the item being stored is a primitive array, box it.
baload, caload, daload, faload, saload	Loads a value from a primitive array	Removes the taint tag for the index to load; loads the taint tag for the corresponding element too
bastore, castore, dastore, iastore, fastore, lastore, sastore	Stores a value to a primitive array	Removes the taint tag for the index to store to; stores the taint tag for the corresponding element too
bipush, iconst, dconst, fconst	Loads a constant to the stack	Loads the taint tag "0" before loading the constant requested
checkcast	Casts the top Object ref	If casting to a primitive array, unbox the boxed primitive array
Xadd, Xmul, Xdiv, Xrem, Xsub, Xand, Xor, Xshl, Xshr, Xushr, Xxor, lcmp, dcmpl, dcmpeg	Performs binary-operand math on top two stack elements	Moves taint tags of operands out of way and ORs them, placing new tag just below the result
dload, fload, iload, lload	Load a primitive local variable	Load the taint tag, just before loading the requested variable

Table 1: All JVM bytecodes, annotated with descriptive transformation information

Opcode(s)	Brief Description	PHOSPHOR Modifications
dstore, fstore, istore, lstore	Store a primitive local variable	After storing the requested variable, store the taint tag
dup, dup2, dup2_x1, dup2_x2, dup_x1, dup_x2	Duplicates the top N words on operand stack, possibly placing under the third or fourth word	Also duplicates the taint tag (if there is one) and if placing under other elements, places under their taint tag (if present)
dreturn, ireturn, freturn, lreturn	Exit a method, returning the primitive value at the top of the stack	Boxes the primitive into a container, then executes ARETURN instead
getfield, getstatic	Retrieves the value of an instance field of an object	If applicable, also retrieves the taint tag just before performing the getfield/getstatic
if_acmpeq, if_acmpne	Jump if the top two object references on stack are/aren't equal	If either operand is a primitive array, pops the taint tag before executing if not performing implicit flow tracking, else adds the jump condition's tag to the PC taint
if_icmplt, if_icmpge, if_icmple, if_icmple, if_icmpeq, if_icmpne	Compare top 2 ints and jumps	Pops the taint tag for both integers before executing if not performing implicit flow tracking, else adds the jump condition's tag to the PC taint

Table 1: All JVM bytecodes, annotated with descriptive transformation information

Opcode(s)	Brief Description	PHOSPHOR Modifications
ifeq, ifne, ifgt, ifge, ifle, iflt	Compares top 1 int and jumps	Pops the taint tag before executing if not performing implicit flow tracking, else adds the jump condition's tag to the PC taint
ifnonnull, ifnull	Jump if top reference is/isn't null	If operand is a primitive array, pops taint tag before executing if not performing implicit flow tracking, else adds the jump condition's tag to the PC taint
instanceof	Return 0/1 if the top reference is (or isn't) the instance of a requested type	If the operand is a primitive array, pops the taint tag before executing. Inserts the taint tag "0" just under the result.
invokespecial, invokevirtual, invokeinterface, invokestatic	Invoke a method, popping the arguments from the stack and placing on top the return value	If the callee is a primitive array, pops the taint tag (all cases but invokestatic); Remaps the method descriptor to include taint tags as necessary; If any parameter is of type "Object" but the type being passed is a primitive array, box it into a container. After return, if return was a container, then unbox it
ldc, ldcw, ldc2_w	Loads a constant onto the stack	If loading a primitive type, load taint tag "0" on stack first

Table 1: All JVM bytecodes, annotated with descriptive transformation information

Opcod(s)	Brief Description	PHOSPHOR Modifications
lookup/table switch	Computed jump	Pops the taint tag of the operand before executing
monitorenter	Obtain lock on the ref. on stack	If the ref. is a primitive array, pops the taint tag before executing
monitorexit	Release lock on the ref. on stack	If the ref. is a primitive array, pops the taint tag before executing
newarray	Create a new 1D primitive array of a given length	Remove the taint for the length of the array; Create a 1D int array of same length to store taint tags before executing.
pop, pop2	Removes the top 1 or 2 words from the stack	If a word being popped is a primitive or primitive array, also remove its taint tag
putfield, putstatic	Stores a value to a field	If the value being stored is a primitive or primitive array, also store taint tag. If storing primitive array to a field of type "Object" then box it first
swap	Swaps the top two words on the stack	If either operand has a taint tag, then ensure that the tags are swapped with the values

Table 1: All JVM bytecodes, annotated with descriptive transformation information

Opcode(s)	Brief Description	PHOSPHOR Modifications
multianewarray	Create (and possibly initializes) a multidimensional array	Removes the taint tag of all operands. If element type is primitive, then changes to a container type, and initializes the last dimension if it would have been otherwise
aconst_null	Loads the constant “null” onto the stack	No modification necessary
athrow	Pops an exception off of the top of the stack and throws it	No modification necessary
d2f, d2i, d2l, f2d, f2i, f2l, i2b, i2c, i2d, i2f, i2l, i2s, l2d, l2f, l2i	Casts primitive types	No modification necessary
dneg, fneg, ineg, lneg	Negates a primitive type	No modification necessary
goto, jsr, ret	Unconditional jump	No modification necessary
new	Creates a new uninitialized object	No modification necessary
return	Returns “void” from a method	No modification necessary
iinc	Increments a local variable	No modification necessary
wide	Indicates that the next instruction accesses a local variable with an index greater than 255	No modification necessary