Faster, More Reliable Software Builds

Thesis proposal

Jonathan Bell
Department of Computer Science
Columbia University
jbell@cs.columbia.edu

October 27, 2015

Abstract

Slow, flaky builds remain a plague for software developers. 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, and a single failed test typically signals a completely failed build. 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*).

This proposal outlines three approaches for consistently accelerating testing, all which leverage observations about *test dependenices*. 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.

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 proposed approach, *ElectricTest*, soundly identifies data dependencies between test cases, allowing for sound test acceleration.

Unfortunately, while *ElectricTest* is sound (i.e. it has no false negatives), it is not very precise: while one test may read data written by a previous test (causing a data dependency), this read may be inconsequential to the test execution, leading *ElectricTest* to perhaps over-constrain test selection and parallelization by over-reporting dependencies. We propose to further refine the dependencies detected in our approach by creating *Beroendet*, which will perform a dynamic data-flow analysis of the tests during execution to eliminate these false positives, based on our previous data-flow analysis tool, PHOSPHOR.

Contents

1 Introduction

When creating unit and integration tests, engineers create scripts that setup the application under test and then feed inputs into units, observing the results. Programs may have explicit inputs (e.g. for a chat server, the message passed to the server), or implicit inputs (e.g. for a chat server, the accumulated state of what users are connected and in what rooms). Software that is stateful, or in other words, software that behaves differently according to what actions have been performed on it in the past, poses many difficulties for software testing due to its abundance of implicit inputs. Even if software is intended to be stateless, when testing it we must still assume that it might be accumulating some state, in order to test that it is indeed stateless.

We therefore may rely on *pre-test* functions in our testing procedures to bring the system to an acceptable state (be it a real state, or a simulated "mocked" state) before testing the unit under scrutiny, and *post-test* functions to clean up any accumulated state. Consider the greatly simplified code example shown in Figure 1 consisting of two classes: `Singleton` and `Example`. The method `Example.doStuff` contains a hidden crash, which occurs only when the application is in a specific state: when `Singleton.flag` is set to true.

If the class `Singleton` is a black box, third party API, then it may be hard for developers to determine the set of actions to exercise all of its states — this is a typical challenge for automated test input generation tools. However, there could also be a dependency *between* test cases: with this state stored in a `static` field, unless it is explicitly cleaned up, it will remain there even after the test finishes, until the application is shut down. If multiple test cases interact with `Singleton.startBlocking` or `Singleton.stopBlocking` and are run in the same JVM, then the tests may interfere with each other — what we will call a dependency on state between test cases. In some cases, developers use a brute force approach to prevent these dependencies from occuring - isolating each test in its own process, imposing a costly slowdown. Detecting these dependencies by hand is risky, and prior work has shown that incorrectly identifying them can lead to unexpected test failures, or worse, undetected faults, despite the existence of test cases that seem to be correct [60].

```java
//Our code under test.
//File: Example.java
public class Example{
  public void doStuff(int input){
    if(Singleton.getFlag()) crash();
  }
}
//Source code for external library we use, which is treated as a black-box
//File: libs/evil.jar
public class Singleton{
  private static int n = 0;
  private static boolean flag = false;
  public static void startBlocking(){
    if(n > 2) flag = true;
    n++;
  }
  public static void stopBlocking(){
    flag = false;
  }
  public static boolean getFlag(){
    return flag;
  }
}
```

Figure 1: **An example of a state dependency**

Hypothesis: The testing process will be made faster and more stable by modifying testing tools to detect otherwise hidden state dependencies.

Sound knowledge of state dependencies between tests can be used to reduce the time needed to isolate test cases from each other. The previous mechanism for isolating dependencies between tests in Java required completely restarting the JVM — our approach instead only reinitializes relevant portions of memory that may be involved in such a dependency, speeding up test suite execution by 62% on average [12]. This approach, VMVM, is described in Chapter 2 of this proposal and received an ACM SIGSOFT Distinguished Paper Award at ICSE 2014.

While VMVM focused on *isolating* the dependencies between test cases, consider the case where several test cases are dependent on each other, intentionally left unisolated: in these cases, we may want to *enforce* the dependencies. Test suite prioritization is a useful technique that reorders the execution of test cases, for example, first executing those test cases most effected by a recent code commit. However, if a dependency is expected to occur between tests, by reordering test cases we may break these dependencies and hence create invalid results. The previous technique for detecting these dependencies required running all possible combinations of tests [86] — requiring an exponential amount of time to detect them (relative to the number of tests). Our approach, *ElectricTest*, detects data dependencies between tests, allowing for sound test acceleration using off-the-shelf techniques through automatic enforcement of these dependencies. *ElectricTest* appeared at FSE 2015 [14], and is described further in Chapter 3 of this proposal.

However, it's very likely that not all data dependencies between tests truly need to be enforced. That is, we have found many data dependencies between tests that (upon manual inspection) do not actually influence the result of the test. In these cases, *ElectricTest* is over-constraining test execution, by requiring data dependencies to be respected that do not actually need to be respected. To detect which data dependencies impact the outcome of a given test, we created a dynamic taint tracking system, PHOSPHOR. PHOSPHOR can track all data dependencies within the JVM, including those from both implicit and explicit data flow. PHOSPHOR appeared at OOPSLA 2014 [11] and ISSTA 2015 [13], and is described further in Chapter 4. We propose to use PHOSPHOR to further refine the test dependencies reported by a tool like *ElectricTest*, with the goal of reporting only dependencies that must be respected for test correctness. This tool, *Beroendet*, will precisely identify all hidden inputs (dependencies) between tests. From this refined list of dependencies, we can further accelerate the sound application of testing techniques such as test selection and test parallelization, by reducing the number of ordering constraints on tests.

1.1 Testing Dominates Build Times

To validate our assumption that accelerating testing will result in a significant acceleration of overall build time, we studied the relative amount of time needed for each build phase in a sampling of open source projects. We also examined the build configuration for these projects, to determine if a build would be halted completely in response to a failed test, or if failed tests did not break the build. 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, sbt, 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.

Phase	**All Projects**	**Only projects building in:**	
		>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%	

Figure 2: **Top three phases in Java builds.**

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.

Figure 2 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. In all but two of these projects, we found that a single failed test would result in a complete build failure. Therefore, to answer our question, we find that testing dominates build times, especially in long running builds. This conclusion underscores the importance of accelerating testing.

1.2 Related work

Test Acceleration. Much work in Java test acceleration focuses on selecting a reduced set of tests to run, or re-ordering the tests as they run to increase the likelihood of finding a failing test early on in the process.

Test Suite Minimization (TSM) is an approach where test cases that do not increase coverage metrics for the overall suite are removed, as redundant [46]. This optimization problem is NP-complete, and there have been many heuristics developed to approximate the minimization [23,24, 45,46,52,53,76,81]. 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., [48]).

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 [81, 82]. On larger applications, Rothermel et al. reported a reduction in fault finding ability of over 50% for more than half of the suites considered [69]. 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 [81] selection of small test suites (on average, less than 7 test cases) reduced the opportunities for loss of fault finding effectiveness [69].

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 [31, 32, 67, 74, 80]. In this way, large test suites can still run in their entirety, with the hopes that faults are detected earlier in the process. Haidry and Miller propose several test prioritization techniques that consider dependencies between tests when performing the minimization, but require developers to manually specify dependencies [42].

Our techniques are complementary to these approaches: for projects that isolate their tests, VMVM can be used to more efficiently isolate them during test execution, and for those that do not isolate their tests, *ElectricTest* can be used to automatically detect dependencies allowing for safe test selection or minimization.

Test Isolation and Dependencies. 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 [60]. 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 [35, 58, 59]. 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 [58]. *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 [86]. 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) [41]. 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 [65] or Metamorphisis [40]. 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 parallelized. 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 [2] or TestNG [3] running with Ant [6] or Maven [7]) 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, eliminates in-memory dependencies between tests without requiring running each test in its own process, greatly reducing the overhead for isolation [12]. 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 [29] 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 [1] and TestNG [3] framework allow developers to specify dependencies between tests, while JUnit [2] allows developers to specify the order to run tests.

VMVM can be seen as similar in overall goal to sandboxing systems [5, 51, 56, 66]. 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., [19, 22, 27, 33, 37]). 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 [63], whereas VMVM functions on any commodity JVM. Xu et al. created a generic language-level technique for snapshotting Java programs [83], however our approach eliminates the need for explicit checkpoints, instead always reinitializing the system to its starting state.

Taint Tracking and Dataflow Analysis. 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 ([75] and others). However, PHOSPHOR tracks taint tags by instrumenting application byte code. Huo and Clause's *OraclePolish* tool utilizes taint tracking to detect overly-constrained test oracles [**?**], and serves as an inspiration for our proposed work, *Beroendet* in Chapter 5.1.

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 [28] (where a slowdown of 1x means that the system now takes twice as much time to run). TaintTrace only performs data flow tainting (like PHOSPHOR), and achieves an average slowdown of 5.53x [25]. 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 [54]. 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 [79], Python [84], PHP [62, 72, 84], Dalvik [34] and the JVM [21, 61]. 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 [61], an approach by Chandra et al. [21], and TaintDroid [34]. 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 [21]. Neither the Jikes nor the Kaffe JVM support the complete Java language specification. Both of these approaches are severely limited in portability, as they are tied directly to these two (infrequently used in production) JVMs. TaintDroid is a popular taint tracking system for Android's Dalvik Virtual Machine (DVM), implemented by modifying the Dalvik interpreter [34]. 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 of [11]) 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 [11].

2 Efficiently Isolating Test Dependencies

2.1 Background and Overview

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 many 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 3 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%).

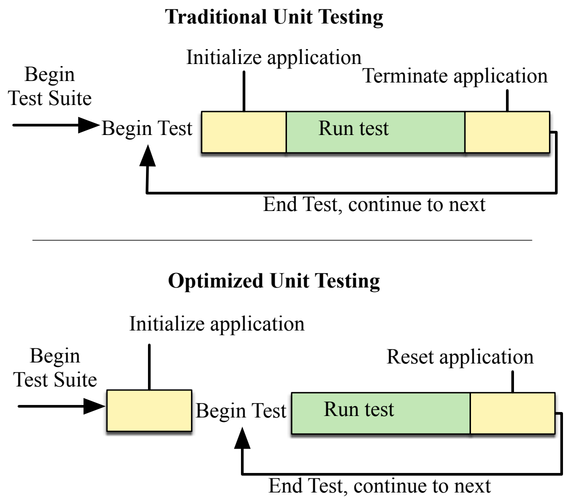

Figure 3: **The test execution loop:** In traditional unit testing, the application under test is restarted for each test. In optimized unit testing, the application is started only once, then each test runs within the same process, which risks in-memory side effects from each test case.

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 3), 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: developers 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 [12]), 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 byte code 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 [2], ant [6] and maven [7], and it is available for download via GitHub [9].

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.

2.2 Project Details

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. 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.

Architecturally, VMVM consists of a static bytecode instrumenter (implemented with the ASM instrumentation library [20]) 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 4.

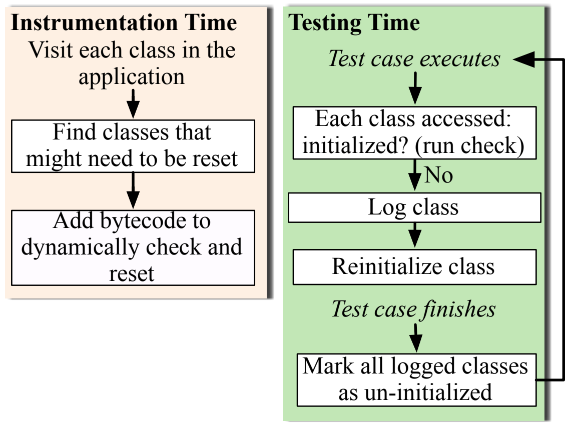
Instrumentation Time
Visit each class in the application
Find classes that might need to be reset
Add bytecode to dynamically check and reset
Testing Time
Test case executes
Each class accessed: initialized? (run check)
No
Log class
Reinitialize class
Test case finishes
Mark all logged classes as un-initialized

Figure 4: **Implementation of VMVM**

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) 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.

2.2.1 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 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

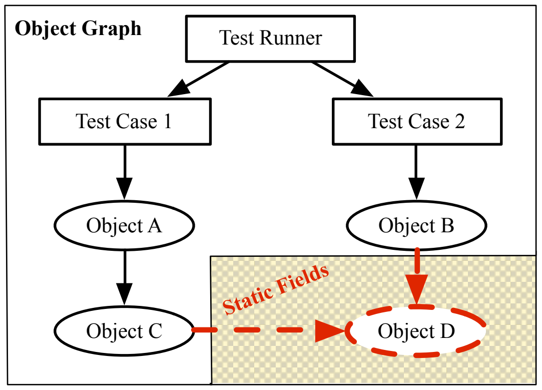

Figure 5: **A leaked reference between two tests.** Notice that the only link between both test cases is through a static field reference.

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 5. 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.

2.2.2 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 1 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).

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

Listing 1: Example of static fields

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 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.

2.2.3 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 [57], 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 §2.3 required such changes.

2.2.4 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.

2.2.5 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 §2.2.3), 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.

2.2.6 Test Automation Integration

VMVM plugs directly into the popular unit testing tool JUnit [2] and build automation systems ant [6] and maven [7]. 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.

Further details regarding the implementation of VMVM, including a more detailed and technical discussion of the instrumentation passes performed, are available in our accompanying technical report [10] or directly on GitHub [9].

2.3 Select Findings

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 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.

2.3.1 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. [85]. 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 [24], Harrold et al's heuristic [46], the GRE heuristic [23, 24], and an ILP model [17]. 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) [30]. 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 [85] 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

Application	LOC (in k)	Test Classes	TSM RS	TSM RT	VMVM RT	Combined 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%

Figure 6: **Test suite optimization with VMVM and with Harrold et al's Test Suite Minimization (TSM) technique [46]**. We show reduction in test suite size (RS, calculated by [85]) for TSM as well as reduction in test execution time (RT) for TSM, VMVM, and the combination of VMVM with TSM.

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.

Figure 6 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 [46] 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 Figure 6. 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% [69] (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.

2.3.2 Study 2: More Applications

To further study the overhead and fault-finding implications of VMVM we applied it to 20 open source Java applications. 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 is available on that project's information page on the Ohloh website [4] and is permanently archived in our accompanying technical report [10].

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

Project	Revisions	LOC (in k)	Age (Years)	# of Tests Classes	# of Tests Methods	Overhead VMVM	Overhead Forking	RT	False Positives VMVM	False Positives 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 1: **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.

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 begin 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 1. 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 1.

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. 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.

2.3.3 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., [45,48,76] 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.

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.

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 paper.

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. These improvements would tie in closely with our proposed work, *Beroendet*, described in Chapter 5.1: 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 [12], is currently available publicly on GitHub [9], and has been the basis for an industrial collaboration with the bay area build acceleration company, ElectricCloud.

3 Detecting Data Dependencies Between Tests

3.1 Background and Overview

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) [23,24,45,46,52,53,76,81], Test Suite Prioritization (which reorders tests to run those most relevant to recent changes first) [31,32,67,74,80], or Test Selection [39,47,64] (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 [86].

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 [68]. 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 [58, 86]. 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 [86]. 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 [13].

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) [86], 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 [86]), *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.

3.2 Project Details

While previous work in the area has focused on detecting *manifest dependencies* between tests [86], 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.

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 [12]), 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 approach 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* also monitors all accesses to files and network hosts, marking any two tests that access the same file or network host as dependent.

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 [42]. 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 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 Select Findings

We evaluated the overhead of *ElectricTest* on the same four subjects studied by Zhang et al., on their tool DTDetector (the previous, state-of-the-art approach for detecting test dependencies) [86], in addition to the ten projects that took the longest to build from our study in Chapter 1.1.

We reproduced Zhang et al.'s experiments [86] in our environment to provide a direct performance comparison between the two tools. Table 2 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 largest open source projects with unisolated tests discovered in our study in Chapter 1.1, recording the number of dependencies detected and the time needed to run the tool.

Table 3 shows the results of this study, showing the number of test classes and test methods in each project, along with the time needed to run the tests normally, 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 2). *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

	# of Tests		Testing Time (Seconds)					*ElectricTest* Speedup vs Dep-Aware
				DTDetector				
Project	**Classes**	**Methods**	**Baseline**	**All 2-pair**	**Dep-Aware Pairs**	**Exhaustive**	***ElectricTest***	
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

Table 2: **Dependency detection times for DTDetector and *ElectricTest* using the same subjects evaluated in [86]**. 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 [86].

	Number of Tests		Testing Time (Min)	Analysis Relative Slowdown	Test Dependencies						# Resources involved at level:	
					Classes			Methods				
Project	**Classes**	**Methods**			**W**	**R**	**C**	**W**	**R**	**C**	**Classes**	**Methods**
camel	5,919	13,562	109.70	22.3X	1,977	3,465	1,356	4,790	8,399	1,695	4,944	5,490
crunch	62	243	17.58	9.4X	9	20	6	18	43	18	190	207
hazelcast	297	2,623	47.37	37.6X	174	200	186	1,163	1,261	1,482	941	1,020
jetty.project	554	5,603	20.08	9.2X	223	261	54	4,016	4,079	424	713	828
mongo-java-driver	58	576	74.25	1.4X	36	36	34	342	362	357	32	33
mule	2,047	10,476	9,698	117.45	185	859	119	2,049	6,279	1,400	11,844	12,387
netty	289	4,601	62.95	5.4X	128	120	63	2,928	3,297	2,926	640	1,104
spring-data-mongodb	141	1,453	121.38	3.0X	114	130	110	1,407	1,404	1,401	1,469	1,489
tachyon	53	362	34.47	2.6X	9	13	9	55	93	13	125	157
titan	177	1,191	81.82	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

Table 3: **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.

still much less than DTDetector's, the previous system for detecting test dependencies [86].

3.4 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. 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 previously ("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. Figure 7 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 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.

	Naive ET		**Greedy ET**		**Unsound**	
Project	**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

Figure 7: **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.

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. Still, some of these detected data dependencies are very likely to not necessarily represent true "manifest" dependencies between tests – the kind that will influence the outcome of a test. Our proposed work, *Beroendet*, (Chapter 5.1) suggests a solution to this limitation.

3.4.1 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 [15,49] 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.

4 Efficiently Performing Dynamic Data-flow Analysis in the JVM

4.1 Background and Overview

To begin looking at which data dependencies might truly influence the outcome of a test, we turned to dynamic taint tracking. 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 [50], end user privacy testing [34, 73] and debugging [36, 55].

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 [28] 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 [21, 50, 61], 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. Our proposed system *Beroendet* (in Chapter 5.1) also requires dynamic taint tracking in Java. Other previous general purpose taint tracking systems for the JVM [21, 61] 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 [26,43,44], 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 [11]. 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 [78].

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 [34], finding that our approach is far more portable, is more precise, and is comparable in performance.

4.2 Project Details

The greatest implementation challenge that PHOSPHOR must overcome is how to associate arbitrary metadata (i.e. taint tags) with variables, without requiring modifications to the JVM. Before describing how taint tags are stored and retrieved in PHOSPHOR, we quickly review JVM memory organization. The JVM is a stack machine with a managed memory environment, where variables are either pointers to an object, pointers to an array, or a primitive value (which include the basic types boolean, byte, char, double, float, int, long, and short). Variables can be stored directly on the operand stack, as local variables within the stack, or on the heap as static fields of classes or instance fields of objects. Method arguments are passed from the operand stack of the call site to the local variable area of the callee.

Previous systems that performed taint tracking in unmodified JVMs tended to do so by making a simple observation: much data (notably, Strings) in the JVM are represented as objects (instances of classes) [26,43,44]. For objects, storing a taint tag is easy: we can simply add an additional field to the definition of every class to store the tag. This approach easily addresses arrays of objects, as each object pointed to by the array still has its own taint tag. However, it does not address primitive values or arrays of primitive values.

PHOSPHOR tracks taint tags for primitive values by adding an additional variable for each primitive variable (similarly, an additional array is added for each primitive array) to store the tags. The tag is stored in a location adjacent to the original primitive variable: as another field of the same class, as another local variable within the stack, an adjacent method argument, or directly adjacent on the operand stack. If a method returns a primitive value, PHOSPHOR changes its return type to return instead a pre-allocated structure containing the original return and its taint tag; just after invocation at the call site, PHOSPHOR inserts instructions to extract both the value and the tag to the stack.

```
// Original Code
int foo(int in){
 int ret = in + val;

 return ret;
}
```

Figure 8: **Source code for a very simple method.** Figure 9 shows the sorts of modifications that PHOSPHOR makes to this codes to store taint tags.

Taint tags for primitive method arguments are always passed just before the argument that is tagged, which simplifies stack shuffling prior to method invocation. PHOSPHOR modifies almost all bytecode operations to be aware of these additional variables. For example, instructions that load primitive values to the operand stack are modified to also load the taint tag to the stack; a complete listing of all of the JVM bytecode operators and the modifications made by PHOSPHOR appears in the Appendix of [11]. PHOSPHOR also wraps all reflection operations to propagate tags through these same semantics as well.

Figures 8 and 9 show at a basic level the sort of changes that PHOSPHOR makes to store and propagate taint tags in Java code. A thorough description of the specific changes that PHOSPHOR

```java
//With Int Tag Tainting
TaintedIntWithIntTag doMath$$PHOSPHOR(
      int in_tag,int in){
  int ret = in + val;
  int ret_tag = in_tag | val_tag;
  return TaintedIntWithIntTag.
   valueOf(ret_tag, ret);
}
```

(a) Modified class, using integer tags for tainting

```java
//With Object Tag Tainting
TaintedIntWithObjTag doMath$$PHOSPHOR(
      Taint in_tag,int in){
  int ret = in + val;
  Taint ret_tag = in_tag.combine(in);
  return TaintedIntWithObjTag.
   valueOf(ret_tag, ret);
}
```

(b) Modified class, using object tags for multi-tainting

Figure 9: The code shown in Figure 8, as would be modified by PHOSPHOR for taint tracking, with changed sections underlined. Example shown at the source level, for easier reading.

makes are detailed in our OOPSLA 2014 publication [11]. While these examples are shown at the level of source code, note that PHOSPHOR functions entirely through bytecode instrumentation, requiring no access to source. PHOSPHOR uses stubs to wrap calls to methods from native code, allowing it to track tags heuristically, a limitation described in detail in our previous work [11].

While most traditional taint tracking systems use integers as tags, PHOSPHOR can allow arbitrary objects to be used as tags, simplifying development of more complicated analyses. Integer tags allow for very low overhead in taint tag propagation, as combining them can be as simple a single instruction (to bitwise OR them), but such a technique limits the total number of taint marks to only 32. On the other hand, by using arbitrary Objects as tags, there can be an arbitrary number of tags (2^{32}), and PHOSPHOR maintains the multiple relationships between each tag ("multi-taint tagging"), but tracking these relationships adds a runtime overhead.

4.2.1 Taint Tag Propagation

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 10 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.

```java
public String leakString(String in){
  String r = "";
  for(int i = 0; i < in.length; i++)
  {
    switch(in.charAt(i)){
      case 'a':
        r+="a";
        break;
      ...
      case 'z':
        r+="z";
        break;
    }
  }
  return r;
}
```

Figure 10: **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.

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 10 to propagate
tags through the method. Our implementation of control flow dependency tracking mirrors that of prior work from Clause et al [28], 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.

4.3 Select Findings

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 Dalivk 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).

In this proposal, we will summarize several of our performance findings: a complete evaluation of PHOSPHOR is available in our OOPSLA 2014 publication [11].

We focus our performance evaluation on macro benchmarks, specifically from the DaCapo [18] benchmark suite (9.12 "bach"). 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. 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. For this experiment, we configured PHOSPHOR to perform data flow tracking only (i.e. to not track implicit flows), and to use integer tags. To control for JIT and other factors, we executed 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 [38]). 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.

Table 4 presents the results of this study, showing detailed results for Oracle's HotSpot JVM

	Oracle Hotspot 7						Other JVMs		
	Runtime (ms)			Heap Size (MB)			Runtime Overhead		
Benchmark	T_b	T_p	Overhead	M_b	M_p	Overhead	HotSpot 8	IcedTea 7	IcedTea 8
avrora	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%

Table 4: **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.

(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% (runtime overhead for other JVMs is shown in Table 4). 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 [11]).

There are several interesting factors that can contribute to the heap overhead growing to be more then 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.

PHOSPHOR appeared originally at OOPSLA 2014 [11], and additional information on its control flow tracking appeared at ISSTA 2015 [13]. PHOSPHOR is available on GitHub [8], and passed the OOPSLA 2014 artifact evaluation process.

5 Future Projects and Research Plan

5.1 Precisely Detecting Manifest Dependencies Between Tests

We propose to significantly enhance our previous *ElectricTest* system by automatically filtering out data dependencies that do not effect test execution, building *Beroendet*. Figure 11 shows the number of dependencies in the ground truth (labeled DTD), and the dependencies detected by *ElectricTest* (W and R). *ElectricTest* found far more data dependencies than we knew actually impacted the outcome of each test. However, gathering the ground truth (by running all possible permutations of tests) is incredibly time intensive, and not feasible for any applications of even nominal size (Table 2 shows the size of each test suite, the longest of which took 22 seconds to run).

	Dependencies			**ET Shared Resource Locations**	
		ET			
Project	**DTD**	**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

Figure 11: **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**).

Beroendet will be a completely new system, built in collaboration with Darko Marinov at UIUC and his student Alex Gyori, that will detect and filter data dependencies to present only those that truly can impact the outcome of a test. Our approach to filtering irrelevant data dependencies leverages dynamic taint tracking (using PHOSPHOR) to track the relevance of every piece of data that is read as part of a dependency. When a test case reaches an assertion, we will mark all data that the assertion depends on (which would include transitively all data that that data depended on) as relevant. Then, we will filter the dependencies to report only those that an assertion depended on.

5.2 Progress and Completion Timeline

Table 5 shows my plan for completion of the research.

Timeline	Work
2010	Complete teaching requirement in December as an MS student
2011	Begin PhD studies and course work in January
2012	Complete course work
2013	Begin work on Java in-process isolation (VMVM) Complete candidacy Begin work on Java data flow analysis (Phosphor)
2014	Extend VMVM work to IEEE Software journal version Summer research with ElectricCloud Begin work on *ElectricTest*
2015	Extend Phosphor to perform control flow taint tracking Thesis Proposal Begin *Beroendet*
2016	Complete *Beroendet*, submit a paper on it to FSE Write and defend thesis (May 2016)

Table 5: **Plan for completion of my research**

Notable previous work not included in this thesis:

- 2010-2013: HALO - Gamifying software engineering education; published at GAS workshop [70], SSE workshop [16], and CSEET conference [71]
- 2012-2013: Chronicler - Lightweight record and replay to reproduce field failures; published at ICSE [15]
- 2013-2014: Pebbles - Fine-grained data management abstractions for Android; published at OSDI [73]
- 2014-2015: Synapse - A data integration architecture for agile microservice development; published at EuroSys [77]

References

[1] Dependency and data driven unit testing framework for java. https://code.google.com/p/depunit/.

[2] Junit: A programmer-oriented testing framework for java. http://junit.org/.

[3] Next generation java testing. http://testng.org/doc/index.html.

[4] Ohloh, inc. http://www.ohloh.net.

[5] 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.

[6] Apache Software Foundation. The apache ant project. http://ant.apache.org/.

[7] Apache Software Foundation. The apache maven project. http://maven.apache.org/.

[8] Jonathan Bell and Gail Kaiser. Phosphor: Dynamic taint tracking for the jvm. https://github.com/Programming-Systems-Lab/phosphor.

[9] Jonathan Bell and Gail Kaiser. Vmvm: Unit test virtualization in java. https://github.com/Programming-Systems-Lab/vmvm.

[10] 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.

[11] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *OOPSLA*, 2014.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Jonathan Bell, Swapneel Sheth, and Gail Kaiser. Secret ninja testing with halo software engineering. In *Proc. of the 4th Int'l Workshop on Social Software Engineering*, 2011.

[17] 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.

[18] 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.

[19] 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.

[20] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[21] 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.

[22] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[23] T.Y. Chen and M.F. Lau. A new heuristic for test suite reduction. *Information and Software Technology*, 40(5–6):347 – 354, 1998.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07*. ACM, 2007.

[29] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Martin Fowler. Eradicating non-determinism in tests. `http://martinfowler.com/articles/nonDeterminism.html`, 2011.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *FSE*, 2014.

[51] Shvetank Jain, Fareha Shafique, Vladan Djeric, 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification*, Java SE 7 edition, Feb 2013.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] John Ousterhout. 10–20x faster software builds. *USENIX ATC*, 2005.

[66] 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.

[67] 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.

[68] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–441, August 1996.

[69] 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 *In Proceedings of the International Conference on Software Maintenance*, pages 34–43.

[70] Swapneel Sheth, Jonathan Bell, and Gail Kaiser. HALO (Highly Addictive, sociaLly Optimized) Software Engineering. In *Proc. of the 1st Int'l Workshop on Games and Software Engineering*, 2011.

[71] Swapneel Sheth, Jonathan Bell, and Gail Kaiser. A competitive-collaborative approach for introducing software engineering in a cs2 class. In *Proceedings of the 2013 Conference on Software Engineering Education and Training*, CSEET 2013, Piscataway, NJ, USA, May 2013. IEEE Press.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] Nicolas Viennot, Mathias Lecuyer, Jonathan Bell, Roxana Geambasu, and Jason Nieh. Synapse: New data integration abstractions for agile web application development. In *Proceedings of The 2015 European Conference on Computer Systems (EuroSys)*, 2015.

[78] Matej Vitásek, 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.

[79] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *ISSTA 2013*. ACM, 2013.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.