REFLECTION-AWARE STATIC REGRESSION TEST SELECTION

BY

MILICA HADŽI-TANOVIĆ

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Darko Marinov

ABSTRACT

Regression test selection (RTS) aims to speed up regression testing by rerunning only tests that are affected by code changes. RTS can be performed using *dynamic* or *static* analysis techniques. A recent study showed that static and dynamic RTS can perform similarly for some medium-sized Java projects. However, the results also showed that static RTS can be sometimes *unsafe*, missing to select some tests that dynamic RTS selects, and reflection was the only cause of unsafety among the evaluated projects.

In this thesis, we investigate five techniques—three purely static techniques and two hybrid static-dynamic techniques—to make static RTS safe with respect to reflection. We implemented four of these reflection-aware techniques as extensions to one reflection-unaware (RU) static RTS technique in a tool called STARTS. We evaluated the fifth technique but did not yet fully implement it. To assess reflection-aware SRTS techniques, we measured benefits and costs of four implemented reflection-aware techniques by comparing their end-to-end times with the RU technique and with RetestAll—the latter runs all tests after every code change. We also compared safety and precision of all five static RTS techniques relative to Ekstazi, a state-of-the-art dynamic RTS technique.

Our results on 805 revisions of 22 open-source Java projects show that all reflection-aware techniques we evaluated can make static RTS safe with respect to reflection, but their costs vary widely. The best purely static technique in our study is based on border analysis with minimal border methods which avoids analyzing JDK and saves, on average, 14.1% of the end-to-end time of RetestAll. Furthermore, the results show that a hybrid technique based on per-test analysis is very promising in terms of safety and precision. On the other hand, the worst techniques were based on string analysis; these techniques are imprecise and often lead to selecting to re-run all tests. Taken together, these results show the need for more research into purely static techniques for making static RTS reflection aware.

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor Professor Darko Marinov, for his guidance and support. It is not that often to come across an advisor who is able to provide unconditional support to his advisees but at the same time openly point out the areas where they need improvement.

I would also like to thank my group mates, Vimuth Fernando, Suleman Mahmood, Deniz Arsan, August Shi, Farah Hariri, Alex Gyori, and Peiyuan Zhao, who were there to support me and who made my time at UIUC more enjoyable. I would like to extend special thanks to Owolabi Legunsen who dedicated much of his time to mentoring me. I have learned a lot from Owolabi and I am grateful for having the opportunity to work with him.

My research assistantships were funded by NSF Grant No. CCF-1409423. My teacher assistenships were funded by the University of Illinois at Urbana-Champaign. I am also grateful for financial support from the Siebel Scholars Foundation.

Further, I would like to thank my beloved family. Without my parents and my sister this whole journey would not have been possible.

I would like to thank my friends for being there for me through all the good times and bad times. At last, but certainly, not the least, I would like to extend special thanks to my dear friends Boban Vukosavljevic and Stefan Markovic, who made me feel like I have a small piece of home here on the other side of the world.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 6
2.1 Static Regression Test Selection 6
2.2 Reflection 7
2.3 Motivating Example 7

CHAPTER 3 REFLECTION-AWARE SRTS 9
3.1 Static Reflection Detection 9
3.2 Hybrid Static-Dynamic Reflection Detection 11

CHAPTER 4 IMPLEMENTATION 14
4.1 RU Analysis in STARTS 14
4.2 String Analysis 14
4.3 Border Analysis 15
4.4 Dynamic Analysis 16

CHAPTER 5 EVALUATION 17
5.1 Experimental Setup 17
5.2 RQ1: Test-Level Safety and Precision 19
5.3 RQ2: Dependency-Level Safety and Precision 21
5.4 RQ3: Selection Rates 22
5.5 RQ4: Time Savings from Reflection-aware SRTS 23
5.6 RQ5: Dependency Graph Results 24

CHAPTER 6 RELATED WORK 26

CHAPTER 7 CONCLUSION 27

REFERENCES 28

CHAPTER 1: INTRODUCTION

Regression testing [42] reruns tests after every code change to check against regression bugs that occur when code changes break previously working functionality. Regression testing is an important activity during software evolution, and successful software systems spend most of their lifespan in a maintenance phase during which they continuously undergo different kinds of changes. However, running all tests in a test suite after every change—often called *RetestAll*—can be quite expensive both in terms of disrupting programmers' workflow (programmers have to wait for test results) and requiring machine time (potentially many tests or long-running tests need to be run). For example, we found out from a recent private communication with a Huawei engineer that the regression test suite for one Huawei product takes over seven *weeks* to run. Other companies, such as Google and Microsoft, have also publicly reported their ever-growing costs of regression testing and some steps they are taking to reduce such costs [11, 12, 15, 17, 31, 39].

Regression test selection (RTS) [14, 22, 23, 32, 33, 35–37, 43] is a way to reduce regression testing costs by rerunning only *affected tests* whose pass/fail behavior may flip as a result of code changes. In other words, RTS saves the time that would have been spent on needlessly running the tests whose behavior cannot flip. An RTS technique first finds the dependencies that each test requires; then, given a code change, the technique selects, as affected, all tests for which at least one dependency changed. It is desirable that an RTS technique be *safe* [36], i.e., select to rerun all affected tests, so that it does not miss to catch any regression bug. Additionally, an RTS technique is *precise* if it selects to rerun only affected tests.

RTS can collect dependencies statically or dynamically, and previous research has mostly focused on dynamic approaches [14, 33, 37, 42, 43]. Recently, both Ekstazi [13, 14] (the state-of-the-art dynamic RTS technique for Java) and STARTS [22, 23] (a purely static RTS technique) demonstrated that, for both dynamic and static RTS, performing RTS at the class level gave a better end-to-end speedup over RetestAll than performing RTS at the method level. Ekstazi instruments the test code and the code under test to collect class-level test dependencies while running the tests. Practitioners have started to adopt the Ekstazi tool [13] and integrated it in the build systems of some open-source projects, like Apache Camel [1], Apache Commons Math [2], and Apache CXF [3].

Despite the recent progress and adoption, dynamic RTS has some limitations due to its reliance on dynamic test dependency collection [9]. For example, in cases of exceptions or non-exhaustive thread-schedule exploration, dynamic RTS may miss to collect complete coverage, resulting in unsafe RTS. The overhead of dynamic dependency collection may

also be prohibitive in resource-constrained settings where dynamic coverage collection can cause tests to exceed tight time bounds (e.g., real-time systems), or in environments where storing and updating of coverage information would be too costly (e.g., ultra-large software ecosystems like those at Google and Microsoft). Static RTS does not suffer from these problems of dynamic analysis, because static RTS uses static analysis to over-approximate the test dependencies for safe RTS. Static RTS performs its analysis at compile time and does not need to instrument the code or to run the tests for its analysis.

A recent study [22] showed that static RTS [5, 20, 35] can perform similarly as dynamic RTS at the class level for some medium-sized Java projects. Static RTS does not require instrumenting code and running tests to find test dependencies, but instead computes test dependencies statically. Specifically, at the class-level, static analysis over-approximates test dependencies by constructing and traversing an *inter-type relation graph* (IRG) [33], in which nodes are types (e.g., Java classes, interfaces, enums, etc.) and edges represent use or inheritance relationships among nodes. The results showed that static, class-level RTS has a similar performance as Ekstazi. However, static RTS was not always safe; in a small number of cases, it missed to select some tests that Ekstazi selected. The only observed cause of unsafety in those experiments was reflection [22].

Reflection is widely used in object-oriented programming languages and allows applications to examine or modify their runtime behavior in a manner not possible with compile time. For example, in Java, one class, `A`, can pass the *name* of another class as a string, `"B"`, to some API which creates instances of `B` which can then be used by instances of A at runtime. The standard Java library methods for dynamically creating instances of B from instances of A are `Class.forName` and `Class.newInstance`; they allow creating objects that represent classes and creating instances of those classes, respectively. Although reflection is a powerful feature that makes code more extensible, it poses significant challenges for any static analysis [8, 21, 26–30, 41]. In particular, for reflection-unaware (RU) static class-level RTS, the IRG would not contain the *reflective edges*, such as from `A` to `B` (unless `A` happens to also have a compile-time static dependency on `B`). Thus RU technique could miss to select some test that is affected by code changes.

Many open-source Java projects use reflection either directly in their own source code or via third-party libraries on which they depend. Our analysis shows that 43.7% of the 1,000 most-forked Java projects on GitHub invoke some reflection API *directly* in their source code projects. Other researchers have also confirmed the wide-spread use of reflection among Java projects; Li et al. [27] reported that 87.6% of 500 randomly chosen Android apps use reflection, and Landman et al. [21] found that analyzing reflection is necessary for 77.9% of the 457 Java projects sampled from Ohloh/OpenHub. It is therefore critical

that static RTS techniques handle reflection to be safe in practice. The problem we address is to investigate reflection-aware (RA) techniques for static RTS, which can (statically or dynamically) recover reflective edges that RU techniques may miss, and add them to the IRG, at the lowest possible cost in terms of the end-to-end time.

We investigate five RA techniques that make static RTS as safe as dynamic RTS with respect to reflection. Three techniques are purely static—*Naïve Analysis*, *String Analysis*, and *Border Analysis*— and the other two are hybrid static-dynamic—*Dynamic Analysis* and *Per-test Analysis*.

Naïve Analysis recovers reflective edges by adding an IRG edge from each class using reflection to all other classes in the project. String Analysis [10] statically approximates class names as string values at reflection API method call sites to determine the potential target classes in the project code, third-party libraries, and the Java standard library (JDK) classes. Border Analysis recovers reflective edges without performing expensive analysis of the JDK, based on our discovery that the imprecision of String Analysis is due to its analysis of the JDK. The key idea in Border Analysis is to identify (manually or automatically), *a priori*, so called *border methods*—methods in the JDK which, when invoked, will eventually lead to the invocation of some reflective API. Subsequently, only non-JDK classes need to be analyzed to build the IRG. Our lightweight static analysis finds classes that invoke a border method and marks these classes as potentially able to reach all other classes in the IRG. Border Analysis improves the imprecision of Naïve Analysis and String Analysis by not including JDK classes in its IRG; commonly used JDK classes that invoke reflection APIs are not approximated to reach all other classes.

Dynamic Analysis is a hybrid approach that uses reflective edges that are dynamically recovered while running *all* the tests to augment the statically constructed IRG; the key idea in Dynamic Analysis is to perform very lightweight instrumentation of test executions to record target classes at reflection sites in one program version [8, 41], and then add the recovered reflective edges to the IRG that is used for performing RTS on the next version. Dynamic Analysis is not a purely static approach; dynamically recovered reflective edges are combined with statically computed dependencies to make for safer RTS. Finally, Per-test Analysis is a modification of Dynamic Analysis. Whereas Dynamic Analysis collects reflective edges from the execution of all tests together, Per-test Analysis collects the reflective edges for each test class individually.

We measured the benefits and costs of these reflection-aware techniques relative to rerunning all tests after every change (i.e., RetestAll) and to RU static RTS [22] (henceforth called *RU Analysis*). RU Analysis is the baseline on which the reflection-aware techniques are built, and is implemented in the publicly available tool, STARTS [23, 40]. We compare the safety

and precision of various static techniques against Ekstazi, and also compare the end-to-end time of all static RTS techniques against RetestAll. We evaluated the reflection-aware techniques on 805 revisions of 22 open-source Java projects, which are a mix of projects from recent studies of static RTS and "reflection-heavy" projects that make more use of reflection than the projects used in studies of static RTS [22, 23, 32].

The results showed that Naïve Analysis and String Analysis were *completely ineffective*; they made static RTS safe but at the cost of always rerunning all the tests after every code change, and were slower than RetestAll, partly because they also analyze JDK classes. Border Analysis and Dynamic Analysis are as safe as Ekstazi on our experimental subjects, but they solve the reflection-related safety issues of RU Analysis at widely varying costs in terms of precision and end-to-end time. Normalized to RetestAll, Border Analysis (59.7%) and Dynamic Analysis (66.7%) select to run more tests than both Ekstazi (22.7%) and RU Analysis (38.9%). Per-test Analysis (40.0%) performs the best with regards to precision while being as safe as Ekstazi. To measure cost, we evaluated the reflection-aware static RTS techniques in both an *online* mode (where the time to collect test dependencies is included in the end-to-end time) and an *offline* mode (where the time to collect test dependencies is not included). The *offline* mode of Border Analysis performed the best among the reflection-aware RTS techniques that we evaluated; on average, Border Analysis was 12.9 percentage points (pp) slower than RU Analysis, but saves 14.1% of RetestAll time. We did not measure the time to execute Per-test Analysis, because we only evaluated the benefit that this analysis can provide but did not have an actual implementation to evaluate its cost.

Beyond evaluating which *tests* are selected by various techniques, an additional technical contribution of this work is to evaluate the *test dependencies* that are computed by various techniques. (We consider transitive and not just direct dependencies in the IRG.) Evaluating test dependencies provides more insights into (potential) test-selection behavior of RTS techniques—it can help to understand why a technique selects or misses to select a test. To the best of our knowledge, no prior study of RTS [42], including the most recent study of static RTS [22], evaluated the impact of computed test dependencies to understand whether static RTS happens to be safe. Evaluating test dependencies can help to understand whether static RTS happened to be safe in prior studies in the presence of reflection because (1) test dependencies are not (largely) under-approximated by missing reflective edges, or (2) test dependencies are under-approximated, but the selection is accidental because actual code changes do not frequently touch dependencies that are only reachable via reflection. In brief, there was no previous evaluation of test dependencies that an RTS technique may be missing. Our evaluation of test dependencies revealed that with RU Analysis, many tests miss some dependencies that Ekstazi finds, showing that reflection-unaware static RTS can

potentially miss to select many tests. However, interestingly, we find that much fewer tests are *actually* missed by RU Analysis. Moreover, we find that Border Analysis, our best purely static reflection-aware RTS technique, as well as Dynamic Analysis and Per-test Analysis, which are hybrid static-dynamic techniques, do not miss any test dependency that Ekstazi finds.

This thesis makes the following contributions:

- ⋆ **Reflection-Aware Static RTS.** We are the first to investigate techniques to make static RTS as safe as dynamic RTS with respect to reflection. Three of the techniques that we evaluated—Border Analysis, Dynamic Analysis, and Per-test Analysis—are as safe as Ekstazi; our current implementations also confirm that two—Border Analysis and Dynamic Analysis—are faster than RetestAll, and we also expect Per-test Analysis to be faster, but we do not have an actual implementation for it.
- ⋆ **Analysis of RTS at the Level of Dependencies.** We present the first analysis of RTS in terms of test dependencies and not just tests. While using RU Analysis leads to many tests missing some test dependencies, making RU Analysis reflection-aware through Border Analysis, Dynamic Analysis, and Per-test Analysis leads to no tests missing dependencies.
- ⋆ **Implementation.** We implemented four reflection-aware static RTS techniques as extensions to RU Analysis, which is publicly available in a Maven-based tool STARTS [23, 40].
- ⋆ **Empirical Study.** We present an empirical study of reflection-aware static RTS on 805 revisions of 22 open-source Java projects. The results showed that three of our techniques, Border Analysis, Dynamic Analysis, and Per-test Analysis, can make static RTS safe with respect to reflection at acceptable costs.

CHAPTER 2: BACKGROUND

In this chapter, we provide background on static regression test selection (SRTS) and reflection. We also show, by means of a motivating example, how the reflection unaware (RU) static RTS technique can be unsafe due to its inability to handle reflection. Recall that an RTS technique is unsafe if it fails to select tests that depend on changed parts of the code.

2.1 STATIC REGRESSION TEST SELECTION

Researchers have proposed SRTS techniques that track dependencies at different granularity levels [5,20,33,35]. Legunsen et al. [22] recently performed an extensive study of SRTS techniques that track dependencies at both class and method levels. The experimental results showed that method-level SRTS based on method call graphs is more imprecise/unsafe and costly than class-level SRTS based on the class-level dependencies. Moreover, the class-level SRTS was comparable to the state-of-art class-level dynamic RTS (DRTS) technique, Ekstazi [14], on some medium-sized projects. Therefore, we focus on improving class-level SRTS. The idea of class-level SRTS originates from the notion of *firewall* [24], which aims to identify code modules that may be impacted by code changes. Kung et al. [20] extended the firewall concept to handle object-oriented language features, e.g., inheritance, and proposed the concept of *class firewall*. Later on, Orso et al. [33] generalized class firewall to the Java language with interfaces.

Given a set of changed classes, a class firewall computes the set of classes that may be impacted by the changes, thus building a "firewall" around the changed classes. Formally, a type (e.g., a class or interface) τ is impacted by a changed type τ_c iff τ can transitively reach τ_c via a sequence of (use or inheritance) edges, denoted as $\tau_c \in \tau \circ E^*$, where * denotes the reflexive and transitive closure, E denotes the set of all edges in the program's IRG, and $\circ$ denotes the relational image. Then, given a program with a set of changed types $\mathcal{T}_c$, the class firewall can be defined as any type that can transitively reach any changed type, i.e., $firewall(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^*$, where $^{-1}$ denotes the inverse relation. Given any two program versions together with the regression test suite T, after the class firewall computation, the class-level SRTS directly returns all the test classes within the class firewall as the selected tests, T_s: $T_s = T \cap firewall(\mathcal{T}_c)$. In theory, class-level SRTS should be safe since it selects all tests that could be impacted by the code changes. However, according to the previous work [22], edges that can only be reached via reflection are missing from the IRG, causing

the reflection-unaware SRTS to miss to select some impacted tests.

2.2 REFLECTION

The main feature of reflection that is relevant to the class-level SRTS is that reflection allows code to construct instances of a class from its name or bytecode representation. The name of the class (whose instance is to be constructed via reflection) can be computed dynamically without statically referencing the class name. The static analysis used in the reflection-unaware class-level SRTS (described in Section 2.1) can fail to detect the use of classes constructed through reflection, making reflection-unaware class-level SRTS potentially unsafe.

In Java, the methods in the reflection API (i.e., reflection methods) that are relevant for class-level SRTS are those that return Java `Class` objects either from string input representing the name of the class or from some bytecode representation that defines the class. The returned `Class` can be used to create instances at runtime. From our manual inspection of the JDK reflection API, we identified four core reflection methods through which all class-related reflection usage eventually happens: `Class.forName()`, `ClassLoader.loadClass()`, `ClassLoader.findSystemClass()`, and `ClassLoader.defineClass()`. The first three of these core reflection methods take a `String` name and return the `Class` represented by that name. The fourth core reflection method takes a byte array and returns the `Class` defined by that byte array. We find that all other possible uses of reflection eventually need some `Class` from these four reflection methods. Therefore, focusing on detecting usages of these four methods suffices to detect all reflection usages for *class*-level SRTS.

2.3 MOTIVATING EXAMPLE

Figure 2.1 presents a code snippet showing example code and tests. In the example, `L` is a standard JDK class; `A1`, `A2`, `A3`, and `A4` are classes within the application, and classes `T1`, `T2`, `T3`, and `T4` form the regression test suite. Suppose that class `A4` is changed (marked in a gray background). Using the reflection-unaware analysis (RU Analysis) in the basic class firewall technique, we find that the changed class `A4` has test class `T4` as its only static dependent because `T4.t4()` directly creates a new instance of `A4`. Figure 2.2(a) shows the static IRG based on RU Analysis, where `T4` would be the only test class affected by this change, and is included in the class firewall (gray area in Figure 2.2(a)). However, selecting only `T4` is unsafe, as more tests also depend, via reflection, on the changed class `A4`. In the example,

```
1 //library code
2 class L {
3   void m() {} //empty method
4   void m(String s) {L.refl(s);...}
5   static Class refl(String s) {...} // reflection API
6 }
7
8 //source code
9 class A1 extends L {
10   void m1(){m("A4");}
11   void m1(boolean b){
12     L.refl("A"+getNum(b));
13     ...}
14   private String getNum(boolean b){
15     return b?"1":"3";}
16 }
17 class A2 {
18   void m2() {} // empty method
19 }
20 class A3 {
21   static void m3(){
22     (new L()).m();}
23 }
24 class A4 {...} //changed code
```

```
1 //test code
2 class T1 {
3   void t1() {
4     A1 a1 = new A1();
5     a1.m1();
6     a1.m1(true);
7   }
8 }
9 class T2 {
10   void t2() {
11     A2 a2 = new A2();
12     a2.m2();
13   }
14 }
15 class T3 {
16   void t3() {
17     A3.m3();
18   }
19 }
20 class T4 {
21   void t4() {
22     A4 a4 = new A4();
23   }
24 }
```

Figure 2.1: Example code

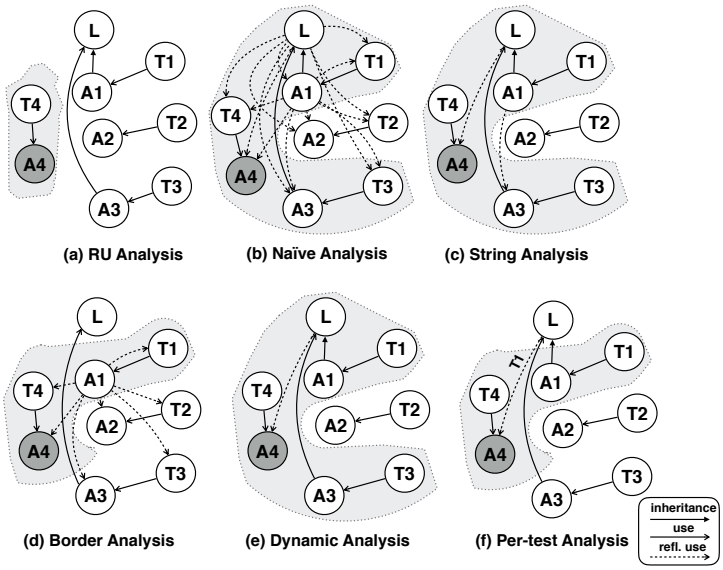

Figure 2.2: Illustration for various reflection-aware analyses

`T1.t1()` creates an instance of `A1` and invokes `A1.m1()`, which invokes `L.m()`, which in turn uses the reflection API (`L.refl()`) to construct an instance of `A4`. As such, `T1` also depends on `A4`, but since RU Analysis is reflection-unaware, it fails to select `T1`, and is thus unsafe.

CHAPTER 3: REFLECTION-AWARE SRTS

We describe the five techniques that we used to augment RU Analysis to become reflection-aware. Essentially, the statically-constructed IRG used in RU Analysis misses reflective edges. Therefore, techniques to make SRTS reflection-aware involve recovering potential reflective edges into the IRG, after which the SRTS algorithm proceeds normally. Recovering missing reflective edges can be done statically or dynamically.

3.1 STATIC REFLECTION DETECTION

We first characterize three purely static reflection-aware techniques that can be used to make SRTS safer with respect to reflection.

3.1.1 Naïve Analysis

The simplest, but the most imprecise approach to detecting reflective edges is to treat each class that invokes a reflection method as having edges to all other classes in the IRG. For ease of presentation, we represent an edge to all classes as an edge to the special node, $*$. The example in Figure 2.1, shows that `A1` and `L` both use the reflection method `L.refl(String)`. Thus, we add to the IRG edges from `A1` and `L` to $*$ in the IRG. The IRG containing additional reflective edges from Naïve Analysis is shown in Figure 2.2(b). There, all test cases that reach `A1` and `L`, namely `T1` and `T3`, now also can reach `A4`, and are thus also in the class firewall (shown in gray), in addition to `T4`. However, `T3` does not use reflection, and does not depend on `T4` which changed. Thus, `T3` should not be selected. In fact, experiments showed that for projects in our data-set, Naïve Analysis always selects all test classes for every change.

3.1.2 String Analysis

String Analysis [10,19,25] is a purely static analysis technique that can be directly applied to approximate potential target classes in reflective edges, based on the `String`-valued class name passed to reflection methods. For example, in Figure 2.1, String Analysis can be used to determine that the reflection invocation site (Line 4) in class `L` can only receive the name, "`A4`". Also, for the reflection site in `A1` (Line 12), String Analysis approximates the class name to match the regular expression, "`A1|A3`". Based on these String Analysis results, as

shown in Figure 2.2(c), reflective edges from `L` to `A4` and from `A1` to `A3` are added to the IRG (a potential edge from `A1` to `A1` can be ignored because self-edges have no effect for SRTS). Thus, when `A4` changes, String Analysis, in addition to `T4`, would correctly select `T1`, which reaches `L` that can, in turn reach `A4` in the IRG. However, String Analysis will also imprecisely select `T3` because analyzing the JDK classes (e.g., `L`) results in many commonly-used internal JDK classes reaching classes in the client code. In this example, although `T3` just uses `L` without using the reflection method `L.refl()`, it still gets selected.

In sum, using String Analysis to recover reflective edges can make SRTS safe but also imprecise because it over-approximates. During our initial experiments, we found that String Analysis incurs large imprecision because it analyzes the internals of the JDK. More specifically, String Analysis often cannot resolve the exact names of classes used as arguments at reflection method call sites in the internals of the JDK without also including additional usage context from those reflection sites. To illustrate, consider JDK internal class, `java.lang.Class`, which uses reflection method to manipulate the Java class that it represents. Statically, it is not known what the exact class being manipulated is. Therefore, String Analysis can only determine that `java.lang.Class` can depend on any class, i.e., $*$. Almost all commonly-used classes in Java (e.g., `java.lang.String`, `java.lang.Integer`) utilize methods from `java.lang.Class` which are not necessarily reflection methods. Unfortunately, adding an edge from `java.lang.Class` to $*$ in the IRG leads every class to depend on $*$, with the implication that all test classes are selected for any code change.

3.1.3 Border Analysis

The severe imprecision of String Analysis for recovering reflective edges happens because most commonly-used classes in the JDK transitively reach classes that are connected to $*$ in the IRG. We propose *Border Analysis*, which avoids analyzing classes in the JDK, while still being safe. Our intuition is that not all methods from the internal JDK, when invoked, can lead to reflection usage. Rather, only a subset of methods internal to the JDK, which we call *border methods*, can lead to the use of reflection. The idea in Border Analysis is to first identify a set of border methods within the JDK. Then, only add edges from non-JDK classes (i.e., client code or third-party libraries) that invoke a border method to $*$ in the IRG (such classes can potentially reach any other class). Note that Border Analysis takes as input border methods that are identified offline *a priori* and avoids subsequent analyses of the JDK internals during the test selection process. Border methods can be identified through manual inspection or automatically, based on heuristics. For the example in Figure 2.2, methods `L.m(String)` and `L.refl(String)` will be identified as border methods, since invoking both of

them from non-JDK classes may potentially lead to the execution of the reflection method, `L.refl(String)`. As shown in Figure 2.2(d), additional reflective edges are added from `A1` to all other classes because it is the only class invoking the identified border methods in the example. This way, tests `T1` and `T4` are precisely selected, without analyzing the JDK internals.

Before rerunning any of our experiments to evaluate Border Analysis, we performed a one-time experiment to identify what are the border methods to use for each project. We determined border methods automatically by instrumenting the execution of all tests in the earliest version of each of our evaluation subjects to capture and process the call stack whenever one of the four reflection methods is executed. Recall the four reflection methods from Section 2.2: `Class.forName()`, `ClassLoader.loadClass()`, `ClassLoader.findSystemClass()`, and `ClassLoader.defineClass()`. The call stack at the point of executing any of these four methods is processed as follows: we find the last method in the stack from a non-JDK class to call a method in a JDK class. This method from a JDK class that is called by the last non-JDK method is returned as a border method. Since we observed the border methods captured in this way to always lead to reflection in at least one calling context, we approximate that all border methods detected this way will lead to reflection usage in all calling contexts and should be used in Border Analysis for subsequent versions of that project. We acknowledge potential imprecision because the border methods that we identified in this manner might only lead to reflection in some, but not all, calling contexts. Note that our collection of border methods was performed once and offline. We envisage that developers may perform such identification of border methods from time to time as their code bases evolve. Finally, some automatically-identified border methods are such that they can only add edges to the IRG that RU Analysis would already find. We therefore also evaluated *minimal Border Analysis*, a variant of Border Analysis which uses a manually identified subset of border methods (which we call *minimal border methods*) that do not lead to finding edges that RU Analysis already finds. We provide more details on our manual process for identifying minimal border methods in Section 4.3.

3.2 HYBRID STATIC-DYNAMIC REFLECTION DETECTION

Next, we describe two hybrid static-dynamic techniques that also make SRTS safe with respect to reflection.

3.2.1 Dynamic Analysis

Reflective edges can also be obtained through Dynamic Analysis, as was done in the previous work on reflection analysis for other testing or analysis tasks [8, 41]. The idea is to execute the tests while instrumenting only the aforementioned four reflection methods to record the class that is constructed by each invocation of the reflection methods. Then, for each invocation of a reflection method, we add an edge to IRG from the class which invoked the reflection method to the class constructed by the reflection method.

The example in Figure 2.2, shows that instrumenting reflection methods during test executions helps discover that test `T1` executes class `L` which uses reflection to target class `A4`. Test `T1` also executes class `A1` that uses reflection to target class `A1`. We add these recovered edges to the IRG, shown in Figure 2.2(e), and SRTS determines that `T1`, `T3`, and `T4` should be selected when `A4` changes. Although Dynamic Analysis is a hybrid static-dynamic RTS approach, it can lead to more precise test selection than with Naïve Analysis and String Analysis. Dynamic Analysis uses very lightweight instrumentation; it only instruments call sites of the four reflection methods. However, Dynamic Analysis still suffers some imprecision because it does not keep track of the test classes during whose execution each invocation of a reflection method occurred. In the example from Figure 2.2, Dynamic Analysis finds only from executing `T1` that there is an exact reflective edge from `L` to `A4`, but the recovered edge from `L` to `A4` is added to the IRG on which reachability for all tests is computed. Therefore, when SRTS finds that `T3` can reach `L`, Dynamic Analysis imprecisely determines that `T3` can also reach `A4`.

3.2.2 Per-test Analysis

Per-test Analysis improves the imprecision of Dynamic Analysis. Dynamic Analysis is imprecise because it combines reflective edges recovered during the execution of all test classes together in the same IRG, leading nodes for certain test classes to have spurious paths to some changed class. In other words, once a reflective edge recovered by Dynamic Analysis is added to the IRG, it is no longer possible to distinguish the test class whose execution necessitated the edge. Thus, the transitive closure of the augmented IRG may now include unnecessary dependencies for some test classes. Per-test Analysis reduces the imprecision of Dynamic Analysis by only adding reflection edges to the IRG when computing dependencies for the test class during whose execution those reflective edges were recovered. Reflective edges computed during the execution of each test class are used to find dependencies only for that test class—these edges are not added to the same IRG that is used for computing

dependencies for all tests. In Figure 2.2(e), Dynamic Analysis selects to rerun `T1`, `T3`, and `T4` because it added the reflective edges recovered from executing all tests to the IRG. More precisely, Dynamic Analysis adds an edge from `L` to `A4` to the IRG, which spuriously makes `T3` transitively dependent on `A4`. On the other hand, Per-test Analysis does not have the edge from `L` to `A4` in the IRG when computing the dependencies for `T3`, but this edge is added to the IRG only when computing dependencies for `T1` (Figure 2.2(f) labels the reflective edge with the test that exercises that edge). The result is that Per-test Analysis does not imprecisely select `T3` when `A4` changes.

CHAPTER 4: IMPLEMENTATION

We implemented four reflection-aware techniques as extensions to the Maven plugin for RU Analysis in the publicly available STARTS tool [40]. We only evaluated the benefits of Per-test Analysis; we did not yet implement Per-test Analysis. In this chapter, we describe the RU Analysis plugin that we extend, and we provide details about our extensions to STARTS. The evaluation of Per-test Analysis is described in the next chapter.

4.1 RU ANALYSIS IN STARTS

The RU Analysis Maven plugin in STARTS implements the class firewall technique described in Section 2.1. It works in three main steps: (i) **Change Computation:** The bytecode comparison feature of Ekstazi [14] is used to compute, as changed, only bytecode files where some non-debug-related information changed; (ii) **Graph Construction:** The bytecode from all program code and third-party dependencies is parsed with jdeps [18] to quickly discover dependency relationships among classes in the program, which are then used to construct the IRG; and (iii) **Graph Traversal:** Given the IRG and the nodes that changed since the last version, find the affected tests as all test classes whose nodes in the IRG can transitively reach to the changed nodes. For each of the reflection-aware techniques, we merely extend the graph construction step to recover reflective edges, as described below, and add such edges to the IRG that it constructs. We did not change the other two steps.

4.2 STRING ANALYSIS

For String Analysis, we use an existing tool, JSA [10], to analyze reflection call sites in both the application code for each project and in the external libraries; all classes that could be loaded into the JVM during test execution need to be analyzed. All Java projects share the same internal Java libraries during class loading. Therefore, to speed up string analysis experiments, we performed String Analysis offline, only once *a priori* (before performing any of our experiments) for all the internal JDK classes and all third-party, non-JDK dependencies. We cache the reflective edges recovered by String Analysis to be reused during RTS. Finally, we extended STARTS to reuse these edges during String Analysis experiments for each project, and only perform String Analysis for the classes in the project itself.

4.3 BORDER ANALYSIS

Recall that border methods can be collected manually or automatically, and we collect border methods for each project once, before running test selection experiments. To collect border methods, we executed the tests in the initial version of each project with a Java Agent attached to the JVM in which the tests are executed. Java Agent instruments the four reflection methods described in Section 2.2: `Class.forName()`, `ClassLoader.loadClass()`, `ClassLoader.findSystemClass()`, and `ClassLoader.defineClass()`. The instrumentation analyzes the stack trace at each invocation of such methods to determine the border method, which is the method to invoke one of the four reflection methods, as described in Section 3.1.3. The Java Agent outputs the collected border methods in a `ShutdownHook`, which is invoked upon JVM shutdown, after all tests have been executed. In our experiments, we collected border methods per subject program in the oldest version among the set of versions that we chose for each subject program. The border methods from each project are then reused in the experiments for all the versions of that program. This automated way of collecting border methods may not detect all possible border methods, because it is limited by the coverage of the tests in the initial version. However, we note that it is sufficient for making SRTS safe until the test coverage changes significantly enough, at which point the developers may rerun analysis to update the border methods. Additionally, developers could also run the analysis to update the border methods periodically during off-peak periods, e.g., overnight or during weekends.

The list of border methods is input to STARTS, and used to perform Border Analysis as follows. First, RU Analysis is done to create an initial IRG. Then, for each class in the project, STARTS uses ASM [4] to statically find invocations of border methods. Next, for any class that STARTS finds to invoke a border method, STARTS creates an edge from that class to $*$. Finally, these recovered reflective edges are added to the initial IRG, and the augmented IRG is used to perform RTS. Internal JDK classes are not added to the IRG for Border Analysis.

Border Methods Used: For each project, we evaluated Border Analysis using two sets of border methods—(1) the full border methods that are obtained directly from instrumentation, and (2) a smaller subset of minimal border methods that we obtained from manually filtering out border methods that lead to unnecessarily adding reflective edges to $*$ when RU Analysis can already determine the concrete nodes involved. We selected the minimal border methods in an attempt to reduce the imprecision that can result from using the larger set, at the risk of potentially being more unsafe. To create the set of minimal border methods, we manually inspected the full set of border methods and kept only those that

we think will always create reflective edges that RU Analysis does not already capture. We extended STARTS to allow users to supply the set of border methods as inputs, and ran all our experiments in this way.

Two collaborators manually inspected the full border methods to select minimal border methods and double-checked the selections. We divided the subject programs into two groups, each of which was assigned to one collaborator for inspection. Each collaborator then double-checked the other's selections to ensure that there was sufficient justification for removing a method from the set of full border methods. An example of a border method that we did not include in the set of minimal border methods is `java.lang.Enum.valueOf()`, which uses reflection to find the `Class` of its `String` argument but merely connects an `Enum` to its declared values—a dependency that RU Analysis already captures.

4.4 DYNAMIC ANALYSIS

To recover reflective edges in Dynamic Analysis, STARTS performs very lightweight instrumentation *during* test executions in each version. The instrumentation is similar to that used for finding border methods (Section 4.3), except that the instrumentation for Dynamic Analysis records the `Class` returned from an invocation of one of the four reflection methods (Section 2.2). Once a `Class` is discovered as being returned from the invocation of a reflection API method call, STARTS records a reflective edge from the calling class to the returned `Class`. These reflective edges are collected during test execution and written to disk in a `ShutdownHook` that is invoked when the JVM shuts down after running all tests. The recovered reflective edges are then used by STARTS to augment the initial IRG from RU Analysis, and the final IRG used to perform RTS. Dynamic Analysis has the benefit that there can be no edge to $*$ in the recovered reflective edges—the exact `Class` returned from invocations of reflection methods are known at runtime. The instrumentation for Dynamic Analysis is more lightweight than the one used in a DRTS technique like Ekstazi [14] because it only instruments the four reflection methods as opposed to instrumenting all classes.

CHAPTER 5: EVALUATION

Our goal is to evaluate whether reflection-aware SRTS techniques can be safe (and still be faster than RetestAll). We first address the two research questions which are related to whether reflection-aware SRTS techniques are safe:

- **RQ1:** What is the safety and precision of the *tests selected* by reflection-aware SRTS, compared with Ekstazi?
- **RQ2:** What is the safety and precision of the *test dependencies computed* by SRTS, compared with Ekstazi?

Chapter 1 discussed why it is important to study not only tests selected but also the test dependencies computed. Answering RQ1 and RQ2 (sections 5.2 and 5.3), we find that reflection-aware SRTS techniques are safe. We then proceed to address the following research questions which are related to how much faster SRTS techniques are relative to RetestAll:

- **RQ3:** How many tests do reflection-aware SRTS techniques select compared with RetestAll, Ekstazi, and RU Analysis?
- **RQ4:** What is the end-to-end time of reflection-aware SRTS techniques compared with RetestAll, Ekstazi, and RU Analysis?
- **RQ5:** What is the impact of reflection-aware SRTS techniques on the size of the IRG computed by SRTS?

We do not show any detailed results for Naïve Analysis and String Analysis, because we found them to be too imprecise and much slower than RetestAll (Section 3.1.2).

5.1 EXPERIMENTAL SETUP

Evaluation Projects: We evaluated all RTS techniques on 805 revisions of 22 open-source, Maven-based, Java projects selected from GitHub. The projects are a mix of 12 projects from the study by Legunsen et al. [22] (selected because their tests run longer than 20s, on average, across all revisions), and 10 additional projects that use reflection (selected because they contain classes that directly invoke `Class.forName()`, and because we could compile and successfully run the tests in these projects *without* any of our analyses in 50 of their 500 most recent revisions). To answer the RQs in more detail, we split the 22 projects in our study into two groups: (i) 11 *small* projects, for which the end-to-end time of running all the tests takes between 20s and 60s on average, and (ii) 11 *big* projects, for which the end-to-end time of running all the tests takes longer than 60s, on average.

Table 5.1: Average violations across projects; *Tests Selected Diff* is diff in tests selected, *Dep Diff* is diff in dependencies found

Project	Tests Selected Diff										Dep Diff	
	X-RU	*RU-X*	*X-*B_m	B_m*-X*	*X-*B_f	B_f*-X*	*X-D*	*D-X*	*X-P*	*P-X*	*X-RU*	*X-*B_m
commons-codec	0.0	5.8	0.0	43.7	0.0	44.2	0.0	44.1	0.0	5.8	0.0	0.0
commons-email	0.0	1.7	0.0	27.8	0.0	27.8	0.0	29.9	0.0	1.7	17.2	0.0
incubator-fluo	0.0	46.7	0.0	61.9	0.0	61.9	0.0	60.2	0.0	46.7	25.9	0.0
commons-compress	0.0	31.7	0.0	59.5	0.0	59.7	0.0	62.2	0.0	31.7	1.5	0.0
retrofit	0.0	16.2	0.0	19.8	0.0	26.1	0.0	25.9	0.0	16.2	6.2	0.0
commons-collections	0.0	21.2	0.0	50.1	0.0	50.5	0.0	50.8	0.0	21.2	0.6	0.0
commons-lang	0.0	20.5	0.0	60.8	0.0	60.8	0.0	59.5	0.0	20.5	0.7	0.0
commons-imaging	0.0	30.5	0.0	42.5	0.0	47.6	0.0	49.9	0.0	30.5	0.0	0.0
graphhopper	0.0	47.0	0.0	81.1	0.0	81.3	0.0	54.5	0.0	47.0	10.1	0.0
robovm	0.0	42.2	0.0	59.5	0.0	59.5	0.0	57.5	0.0	42.2	3.1	0.0
ninja	35.4	26.1	0.0	59.7	0.0	59.7	0.0	61.2	0.0	27.2	45.4	0.0
commons-math	0.0	13.7	0.0	34.5	0.0	34.5	0.0	34.6	0.0	13.7	4.9	0.0
commons-io	0.0	10.5	0.0	37.4	0.0	38.1	0.0	50.3	0.0	10.5	0.0	0.0
HikariCP	0.0	9.9	0.0	17.8	0.0	17.8	0.0	22.8	0.0	9.9	51.8	0.0
undertow	3.4	45.4	0.0	56.4	0.0	57.3	0.0	57.3	0.0	49.7	17.9	0.5
OpenMrs	23.3	24.6	0.0	78.2	0.0	78.3	0.0	78.4	3.3	29.3	36.2	0.0
OpenTripPlanner	0.0	57.8	0.0	69.3	0.0	72.2	0.0	67.7	0.0	57.8	12.8	0.0
commons-pool	0.0	21.7	0.0	29.7	0.0	29.7	0.0	33.8	0.0	21.7	24.2	0.0
Activiti	0.0	30.8	0.0	36.2	0.0	36.2	0.0	36.2	0.0	30.8	36.1	0.0
aws-sdk-java	61.7	25.4	60.9	29.4	60.9	29.9	61.0	30.1	0.0	25.4	13.6	0.2
mapdb	1.0	34.0	0.0	48.3	0.0	48.3	0.0	48.5	0.0	37.8	1.6	0.0
accumulo	0.0	39.6	0.0	83.4	0.0	83.5	0.0	53.1	0.0	39.7	37.1	0.0
Average(OVERALL)	**5.7**	**27.4**	**2.8**	**49.4**	**2.8**	**50.2**	**2.8**	**48.6**	**0.1**	**28.1**	**15.1**	**0.0**

Note that for RTS evaluation, the end-to-end time of running tests is a more important factor than the size of the codebase.

Running Experiments: All experiments involving SRTS, both reflection-aware and reflection-unaware, were performed using STARTS [40] (Section 4). For dynamic RTS, we used Ekstazi [13, 14]. Both STARTS and Ekstazi are publicly available. We automated the run of tests across all the revisions of all the projects in our study. All timing experiments were performed on Amazon EC2 "`m4.xlarge`" instances with four 2.3 GHz Intel Xeon E5-2686 v4 or 2.4 GHz Intel Xeon E5-2676 v3 processors, 16 GB of RAM, 100GB of SSD storage, and running Ubuntu 16.04.03 and Oracle Java 1.8.0_144-b01. We ran all versions of each project in the same EC2 instance. We also compared some timing ratios from EC2 with our runs on a quiet machine and found EC2 to be giving highly accurate ratios.

Simulating Per-test Analysis: We simulated Per-test Analysis, because we did not yet fully implement it in STARTS. We performed our simulation in the following manner. First, we collected test dependencies by running Ekstazi on all versions of each project. Ekstazi saves dependencies for each test class in a separate file on disk. Second, we used STARTS with RU Analysis to build the IRG. Finally, we computed the dependencies of each test class as all nodes reachable in the IRG from all dependencies that Ekstazi found for that test class. By computing reachability from all dependencies that Ekstazi found, our simulated Per-test Analysis recovers any reflective edges that were missing in the IRG computed from RU Analysis. Also, in our simulation of Per-test Analysis, we do not add any edges to the IRG that is used to compute dependencies for all test classes (which is the source of imprecision in Dynamic Analysis). Rather, the nodes reachable from each dependency that Ekstazi found for a test class are simply returned as the dependencies for that test class.

5.2 RQ1: TEST-LEVEL SAFETY AND PRECISION

Table 5.1 shows the comparison of the test-level safety and precision of reflection-aware SRTS with (i) RU Analysis (to see whether reflection-awareness makes RU Analysis safer), and (ii) Ekstazi (to check whether reflection-awareness makes SRTS as safe as Ekstazi with respect to reflection). Recall that a technique is safe if it selects to rerun *all* affected tests and precise if it selects to rerun *only* the affected tests. In the absence of a ground truth for safety and precision in RTS, we compared the *safety violations* and *precision violations* of SRTS against Ekstazi, as defined by Legunsen et al. [22]: "Let E be the set of tests selected by Ekstazi and T be the set of tests selected by another technique on some version. Safety, respectively precision, violations are computed as $|E \setminus T|/|E \cup T|$, respectively $|T \setminus E|/|E \cup T|$, and measure how much a technique is less safe, respectively precise, than Ekstazi; lower

percentages are better. We consider the union of tests selected by both Ekstazi and the technique to avoid division by zero in cases where Ekstazi does not select any test but an SRTS technique selects some tests." In Table 5.1 (and all subsequent tables), we represent Ekstazi as X, RU Analysis as RU, Border Analysis with full border methods as B_f, Border Analysis with minimal border methods as B_m, Dynamic Analysis as D, and Per-test Analysis as P. Columns X-RU, X-B_m, X-B_f, X-D, and X-P show the safety violations of the five SRTS techniques. Columns RU-X, B_m-X, B_f-X, D-X, and P-X show the precision violations.

Concerning safety violations, Table 5.1 shows that RU Analysis has an average safety violation of 5.7% across all projects. RU Analysis is reflection-unaware, missing to select some affected tests and therefore unsafe relative to Ekstazi; the X-RU column shows the degree to which RU Analysis is unsafe. For Border Analysis, with full border methods and minimal border methods, the average safety violations across all projects are the same, 2.8%. Dynamic Analysis appears unsafe as well. However, our inspection shows that *these three reflection-aware techniques are as safe as Ekstazi*, although it would appear from Table 5.1 that the SRTS techniques are unsafe for the `aws-sdk-java` project. Our manual inspection shows that these safety violations in `aws-sdk-java` are actually caused by imprecision in Ekstazi. In other words, Ekstazi was imprecise, and the test classes seemingly "missed" by the SRTS techniques need not be selected. More specifically, these classes do not contain any test methods of their own but only contain nested test classes that, in turn, contain test methods. Ekstazi does not currently track the JUnit runner for such test classes (`org.junit.experimental.runners.Enclosed`) and always selects to run these test classes, even when no code changes. We reported this issue to the Ekstazi developers. We leave it as future work to inspect OpenMrs.

The precision violations are higher for reflection-aware SRTS than for reflection-unaware RU Analysis, showing that reflection-awareness amplifies the inherent imprecision of SRTS. In particular, reflection-awareness causes many more tests to be selected: where RU Analysis has an average precision violation of 27.4%, Border Analysis with minimal border methods has 49.4%, Border Analysis with full border methods has 50.2%, and Dynamic Analysis has a precision violation of 48.6%. Although reflection-aware SRTS improved test-level safety issues of RU Analysis, it also incurs a high cost due to the increased imprecision. In contrast, Per-test Analysis has an average precision violation of 28.1%, which is similar to RU Analysis. Since we did not implement Per-test Analysis, we did not measure its end-to-end time. A possible approach to measuring time would be to run each test separately to collect its reflective edges to use them to compute dependencies in IRG. However, that would incur a high cost and would not be comparable with the other techniques.

Table 5.2: Average percentage (%) of ALL test classes selected by RTS

Project	SHAs	ALL	Selected [%]				
			RU	B_m	B_f	*D*	*P*
commons-codec	50	52.0	7.5	20.0	21.0	50.0	7.5
commons-email	50	17.4	21.2	44.9	44.9	50.0	21.2
incubator-fluo	50	25.1	30.6	56.5	56.5	47.5	30.6
commons-compress	50	118.3	29.7	50.5	52.2	74.0	29.7
retrofit	50	49.6	51.4	54.1	58.3	57.8	51.4
commons-collections	50	160.0	16.0	55.2	58.1	60.0	16.0
commons-lang	50	147.7	23.8	63.6	63.6	68.0	23.8
commons-imaging	50	72.6	56.2	65.5	75.5	82.0	56.2
graphhopper	50	128.5	45.4	50.5	52.2	57.5	45.4
robovm	50	32.2	22.1	50.3	50.3	49.0	22.1
ninja	50	103.7	32.6	82.3	82.3	87.0	45.4
Average(SMALL)	50.0	82.5	**30.6**	**54.0**	**55.9**	**62.1**	**31.8**
commons-math	50	446.3	17.0	43.6	43.6	44.0	17.0
commons-io	50	99.4	16.0	21.6	22.0	60.1	16.0
HikariCP	28	29.8	85.6	93.9	93.9	100.0	85.6
undertow	9	231.0	79.8	97.3	100.0	100.0	85.8
OpenMrs	23	268.1	38.6	94.2	94.6	94.8	41.1
OpenTripPlanner	17	136.0	65.6	84.1	94.5	94.1	65.6
commons-pool	21	20.0	42.1	49.3	49.3	61.9	42.1
Activiti	20	321.0	48.3	53.7	53.7	53.7	48.3
aws-sdk-java	20	172.1	15.8	26.2	28.9	30.7	17.3
mapdb	7	166.1	83.1	99.5	99.5	100.0	83.4
accumulo	10	341.0	28.2	57.0	58.1	45.6	28.3
Average(BIG)	23.2	202.8	**47.3**	**65.5**	**67.1**	**71.3**	**48.2**
Average(OVERALL)	36.6	142.6	**38.9**	**59.7**	**61.5**	**66.7**	**40.0**

5.3 RQ2: DEPENDENCY-LEVEL SAFETY AND PRECISION

We computed dependency-level safety violations in a manner slightly different from the test-level safety violations. At the dependency level, safety violation is the percentage of *all tests* for which there is a non-zero number of dependencies computed by Ekstazi but not by a SRTS technique. (At the test level, we consider the percentage of *only selected tests*.) Table 5.1 shows (*Dep Diff* column) the dependency-level safety violations of RU Analysis and the reflection-aware RTS technique that we found to select the fewest tests: Border Analysis with minimal border methods. There, *X-RU* shows the average percentage of all tests for which Ekstazi finds some dependency that RU Analysis did not find, and X-B_m shows the average percentage of all tests for which Ekstazi finds some dependency that Border Analysis with minimal border methods did not find.

We can see that there were only four (out of 22) projects where RU Analysis did not have a safety violation at the dependency level. For all other projects, RU Analysis misses

dependencies for a large percentage of tests, up to 51.8% for `HikariCP`. In contrast with what the literature reported [22], it is clear that reflection-unawareness can *potentially* lead to many more affected tests being missed during RTS, which shows how unsafe RU Analysis can really be. The table also shows that, at the dependency level, Border Analysis with minimal border methods is actually as safe as Ekstazi, except in two projects, `undertow` and `aws-sdk-java`, which have mostly very small percentage of tests with a dependency-level safety violation. Our manual inspection of `undertow` found the tests with dependency-level safety violations to miss the same dependency, `io.undertow.testutils.DebuggingSlicePool`, which is a class in `undertow` itself. Further inspection revealed that the issue is due to test-order dependency [7, 44]. Specifically, `undertow` has many tests that are run with a custom JUnit runner defined by the `undertow` developers, which adds a new `RunListener` that uses an instance of the `DebuggingSlicePool` to the `RunNotifier` passed from JUnit to the custom runner. This `RunNotifier` is used later even by the basic JUnit runner, so when later tests are run by the regular JUnit runner, the added `RunListener` is still invoked (even when it is unnecessary as the test is not run by the custom runner), causing Ekstazi to add a dependency from those tests to `DebuggingSlicePool`. As such, we determine that tests with a missing dependency on `DebuggingSlicePool` are not missing a true dependency, as Ekstazi does not find a dependency on `DebuggingSlicePool` for those same tests when the test-run order is changed so that they are run before tests that are run with the custom runner. The dependency-level safety violations in `aws-sdk-java` was also due to test-order dependency.

5.4 RQ3: SELECTION RATES

Table 5.2 represents the average percentage of tests selected that resulted from using different RTS techniques in our evaluation. The table shows, for each project, the average total number of tests (ALL) across all the revisions (SHAs) and the percentage of tests selected by each technique in the project (Selected [%] column). The numbers in Selected [%] (Table 5.2) are averaged across all revisions in each project. The overall averages are shown in the Average(OVERALL) row.

The results in Table 5.2 show that reflection-awareness for SRTS comes at the cost of selecting more tests than RU Analysis, which is already more imprecise than Ekstazi (Table 5.1). Overall, on average, Border Analysis with minimal border methods selects 59.7%, Border Analysis with full border methods selects 61.5%, and Dynamic Analysis selects 66.7% of all tests. Border Analysis with minimal border methods performs the best out of all purely static reflection-aware SRTS techniques in terms of selection numbers and percentages. On the other hand, Per-test Analysis selects at the rate similar to the selection rate of RU

Table 5.3: Average % of RetestAll time (RTA[s]) for RTS in "online" mode (*on[%]*) and "offline" mode (*off[%]*)

Project	RTA[s]	X	RU		B_m		B_f		D
		on[%]	off[%]	on[%]	off[%]	on[%]	off[%]	on[%]	on[%]
commons-codec	22.0	52.1	54.3	56.5	68.0	70.6	67.6	70.8	99.2
commons-email	24.7	66.9	60.7	63.0	67.7	70.3	65.2	68.5	104.3
incubator-fluo	27.4	92.5	116.4	141.3	122.5	157.1	121.9	156.6	210.2
commons-compress	28.8	55.2	73.5	78.8	90.5	95.9	90.4	95.9	121.5
retrofit	30.1	88.1	90.9	103.3	92.4	106.7	92.5	107.0	248.4
commons-collections	30.6	53.7	67.8	71.5	83.9	87.7	84.4	88.0	105.5
commons-lang	33.8	49.7	60.6	64.7	90.3	95.0	90.1	95.0	112.6
commons-imaging	40.9	75.4	83.9	87.6	93.5	97.5	93.4	97.4	114.9
graphhopper	52.9	69.2	79.3	85.0	85.2	95.7	85.7	96.2	133.9
robovm	52.9	96.1	106.2	111.9	111.2	119.8	111.5	120.2	122.1
ninja	58.4	142.2	84.5	105.6	108.0	135.4	111.6	136.5	254.0
Average(SMALL)	36.6	**76.5**	**79.8**	**88.1**	**92.1**	**102.9**	**92.2**	**102.9**	**147.9**
commons-math	118.5	35.2	39.4	40.4	58.6	59.9	58.8	59.9	65.0
commons-io	135.3	26.4	31.5	32.0	36.1	36.6	36.4	36.9	74.1
HikariCP	162.0	98.9	95.4	97.3	100.0	102.0	99.8	101.8	119.5
undertow	187.1	74.0	89.7	93.9	102.4	107.8	102.9	108.2	111.7
OpenMrs	207.9	75.4	51.1	55.4	99.1	105.4	97.8	104.7	127.0
OpenTripPlanner	271.5	70.1	95.2	98.3	103.3	106.7	103.6	107.1	104.8
commons-pool	310.2	48.3	61.1	61.3	65.5	65.8	65.7	65.9	65.7
Activiti	312.9	46.8	62.7	63.9	67.2	68.8	67.0	68.6	76.6
aws-sdk-java	337.2	39.5	59.9	60.4	65.8	66.6	66.2	66.9	69.0
mapdb	367.3	92.9	95.2	97.1	107.5	109.8	106.4	108.5	577.3
accumulo	639.4	34.6	47.9	53.2	71.8	81.3	71.9	81.6	78.2
Average(BIG)	277.2	**58.4**	**66.3**	**68.5**	**79.8**	**82.8**	**79.7**	**82.7**	**133.5**
Average(OVERALL)	156.9	**67.4**	**73.0**	**78.3**	**85.9**	**92.8**	**86.0**	**92.8**	**140.7**

Analysis, 40.0%, while still being as safe as Ekstazi.

5.5 RQ4: TIME SAVINGS FROM REFLECTION-AWARE SRTS

Table 5.3 shows the time savings obtained from performing RTS on all evaluated projects. For each project, the table shows the average time to run all tests across all revisions (RTA[s]), and also shows for each RTS technique (on[%] column), the average *end-to-end* RTS time—the time to compile, compute changes, analyze dependencies to find affected tests, execute the selected tests, and update the test dependencies for the test selection on the next revision ("online" mode)—as the percentage of RTA[s]. For the purely static techniques, it also shows the time as a percentage of RTA[s] for an "offline" mode (off[%]). The off[%] columns do not include the time to update the test dependencies, while the on[%]

columns include this time. The upper part of Table 5.3 shows the results for the 11 small projects where the end-to-end time to run all tests is between 20s and 60s, on average. The lower part of the table shows the results for the 11 big projects where the average end-to-end time to run all tests is greater than 60s.

For small projects, the results in Table 5.3 show that RTS, particularly SRTS, does *not* save as much time compared to RetestAll—the SRTS techniques are more often slower than RetestAll for these small projects. This slowdown shows that small projects do not benefit from RTS. The savings from RTS are greater for the big projects. Ekstazi saves the most time among all RTS techniques, but the purely static SRTS techniques, on average, are faster than RetestAll for these big projects. Dynamic Analysis is slower than RetestAll on average, partly because of its imprecision, which results in selecting to run all tests in three of the big projects. RU Analysis takes 68.5% of RetestAll time, but is reflection-unaware. For the reflection-aware SRTS techniques, Border Analysis with minimal border methods is slightly more precise than Border Analysis with full border methods, but they have essentially the same end-to-end running time: 82.8% of RetestAll time. We did not measure the time for Per-test Analysis because we only performed simulations using the data from Ekstazi.

An RTS technique not only selects what tests to run for the current project revision but also needs to update test dependencies for the next revision. Updating test dependencies can be easily done in the background for purely static SRTS, as measured in the "offline" mode. For dynamic (or hybrid) RTS, an offline mode is less practical, because it would have a higher total machine-time cost, requiring that tests be run twice: once without instrumentation to provide faster feedback to the developers and once with instrumentation to collect dependencies for the next revision [14]. We measure the time to perform RTS for RU Analysis and Border Analysis without including the time to update dependencies. In this offline mode, the time savings improves for all the purely SRTS techniques as seen in the average rows for the off[%] columns in Table 5.3. Although the SRTS techniques still run slower than Ekstazi, the offline modes are faster than online modes, and Border Analysis performs the best, with minimal border methods taking on average 85.9% of RetestAll time across all projects.

5.6 RQ5: DEPENDENCY GRAPH RESULTS

As an internal measure of the complexity of the projects that are analyzed for SRTS, we compute the number of nodes and edges in the IRGs constructed for each SRTS technique. The IRGs constructed by RU Analysis have, on average across all projects, 28889.1 nodes and 324401.5 edges. The graphs used for RU Analysis contain nodes from both the program's

classes and classes from the third-party libraries that the program depends on. This is different from the implementation used in the study by Legunsen et al. [22], which only tracked classes in the program's code—a choice justified by the fact that their technique is reflection-*unaware* and external dependencies that do not change could not make it more unsafe with respect to reflection. Classes from third-party dependencies are included in graphs used in RU Analysis to study dependency-level safety and precision, and provide an initial IRG on which other reflection-aware SRTS techniques can be built. This larger graph partly explains the bigger difference in relative performance of RU Analysis and Ekstazi in terms of end-to-end time, compared with the difference that Legunsen et al. [22] reported.

Adding reflective edges to the IRG increases the number of nodes reachable from the tests in the IRG, allowing SRTS techniques to explore more edges and reach more classes. Dynamic Analysis adds the largest number of extra nodes and edges, having 165419.7 and 1897009.5, respectively, on average. This is expected because Dynamic Analysis tracks the internals of the JDK and therefore finds many classes that are reachable through reflection. Border Analysis IRGs have, on average, 39215.7 nodes and 443717.7 edges, when performing Border Analysis with minimal border methods, and 39215.9 nodes and 445393.1 edges, when performing Border Analysis with full border methods. The trend in the sizes of the IRGs correlates with the selection rates of the various RTS techniques, showing that techniques with fewer/more nodes and edges in their IRG select fewer/more tests to run. Also, observe that the IRG sizes for Border Analysis are essentially the same, regardless of whether one uses minimal border methods or full border methods—a trend that is also correlated with the fact that both techniques have about the same test-selection rates and end-to-end test-running time.

CHAPTER 6: RELATED WORK

Dynamic RTS techniques have been intensively studied in the literature. Rothermel et al. [36, 37] proposed one of the first dynamic RTS techniques for C programs based on detailed basic-block-level analysis. Harrold et al. [16] and Orso et al. [33] later proposed to handle object-oriented features and adapt the basic-block-level RTS for Java programs. In recent years, researchers have started to investigate coarser-grained dynamic RTS analyses due to the increasing software size. For example, Ren et al. [34] and Zhang et al. [43] studied method-level dynamic RTS. Recently, Gligoric et al. [14] proposed the class-level dynamic RTS technique, Ekstazi, and demonstrated that Ekstazi can have even shorter end-to-end testing time than existing method-level dynamic RTS due to the lower overhead of coarse-grained analysis.

Static RTS techniques [20,35] are not as well studied, and their effectiveness and efficiency were largely unknown before the recent work by Legunsen et al. [22]. In that work, they evaluated the effectiveness of static RTS and compared it against the dynamic RTS in Ekstazi. The experiments showed static RTS to be comparable to dynamic RTS. However, there were cases where static RTS was unsafe and failed to select some tests that Ekstazi selects; all the cases were due to reflection. In this work, we focus on the issue of reflection and its influence on static RTS. We propose techniques to handle reflection in static RTS, by statically analyzing strings or border methods, or by dynamically collecting reflective dependencies.

Many researchers have studied the impacts of reflection in modern programming languages on static analysis [6, 8, 26, 27, 27–29, 29, 30, 38, 41]. However, none of the existing studies investigate the impacts of reflection in the context of RTS. In other words, we are the first to address the reflection issue for safe static RTS. The most related previous work [8, 41] studied how to perform static analysis and refactoring in the presence of reflection. Bodden et al. [8] proposed instrumenting reflection sites to dynamically record when classes invoke reflection and what classes they depend on through reflection. We adopt the same in our Dynamic Analysis, except we apply it to SRTS. The results show that Dynamic Analysis performs worse than our Border Analysis.

CHAPTER 7: CONCLUSION

We proposed and empirically studied five reflection-aware RTS techniques—Naïve Analysis, String Analysis, Border Analysis, Dynamic Analysis, and Per-test Analysis. We compared all these techniques with Ekstazi, a dynamic RTS technique, and a baseline reflection-unaware static RTS technique, RU Analysis. The experimental results on 805 revisions of 22 open-source GitHub Java projects demonstrate that all our reflection-aware techniques make static RTS safe with respect to reflection, but their costs vary widely—Naïve Analysis and String Analysis select every test for every studied project; the pure static Border Analysis is the best purely static reflection-aware RTS technique, and its end-to-end time is 18.5pp higher than that of Ekstazi. The hybrid static-dynamic technique, Dynamic Analysis, is safe, but with high imprecision and end-to-end time 40.7% higher than RetestAll; Per-test Analysis has the best safety/precision trade off—it is as safe as Ekstazi and, on average, it selects to rerun a similar number of test as RU Analysis. Furthermore, we also performed the first detailed safety/precision analysis at the test dependency level. The analysis results demonstrate that RU Analysis could be unsafe in practice, indicating the importance of building reflection-aware static RTS techniques.

REFERENCES

[1] Apache Camel. http://camel.apache.org/.

[2] Apache Commons Math. https://commons.apache.org/proper/commons-math/.

[3] Apache CXF. https://cxf.apache.org/.

[4] ASM. http://asm.ow2.org/.

[5] Badri, L., Badri, M., and St-Yves, D. Supporting predictive change impactanalysis: a control call graph based technique analysis: a control call graph based technique. In *APSEC* (2005), pp. 167–175.

[6] Barros, P., Just, R., Millstein, S., Vines, P., Dietl, W., d'Amorim, M., and Ernst, M. D. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE* (2015), pp. 669–679.

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

[8] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE* (2011), pp. 241–250.

[9] Chen, Y.-F., Rosenblum, D. S., and Vo, K.-P. TestTube: A System for Selective Regression Testing. In *ICSE* (1994), pp. 211–220.

[10] Christensen, A. S., Møller, A., and Schwartzbach, M. I. Precise analysis of string expressions. In *SAS* (2003), pp. 1–18.

[11] Elbaum, S., Rothermel, G., and Penix, J. Techniques for Improving Regression Testing in Continuous Integration Development Environments. In *FSE* (2014), pp. 235–245.

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

[13] Gligoric, M., Eloussi, L., and Marinov, D. Ekstazi: Lightweight Test Selection. In *ICSE Demo* (2015), pp. 713–716.

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

[15] Gupta, P., Ivey, M., and Penix, J. Testing at the speed and scale of Google, Jun 2011. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[16] Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S. A., and Gujarathi, A. Regression test selection for Java software. In *ACM SIGPLAN Notices* (2001), vol. 36, pp. 312–326.

[17] Herzig, K., and Nagappan, N. Empirically Detecting False Test Alarms Using Association Rules. In *ICSE* (2015), pp. 39–48.

[18] jdeps. `https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html`.

[19] Kirkegaard, C., Moller, A., and Schwartzbach, M. I. Static analysis of XML transformations in Java. *TSE 30*, 3 (2004), 181–192.

[20] Kung, D. C., Gao, J., Hsia, P., Lin, J., and Toyoshima, Y. Class firewall, test order, and regression testing of object-oriented programs. *JOOP 8*, 2 (1995), 51–65.

[21] Landman, D., Serebrenik, A., and Vinju, J. J. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *ICSE* (2017), pp. 507–518.

[22] Legunsen, O., Hariri, F., Shi, A., Lu, Y., Zhang, L., and Marinov, D. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE* (2016), pp. 583–594.

[23] Legunsen, O., Shi, A., and Marinov, D. STARTS: STAtic Regression Test Selection. In *ASE* (2017), pp. 949–954.

[24] Leung, H. K., and White, L. A study of integration testing and software regression at the integration level. In *ICSM* (1990), pp. 290–301.

[25] Li, D., Lyu, Y., Wan, M., and Halfond, W. G. String analysis for Java and Android applications. In *FSE* (2015), pp. 661–672.

[26] Li, L., Bissyandé, T. F., Octeau, D., and Klein, J. DroidRA: Taming Reflection to Support Whole-Program Analysis of Android Apps. In *ISSTA* (2016), ACM, pp. 318–329.

[27] Li, L., Bissyandé, T. F., Octeau, D., and Klein, J. Reflection-aware static analysis of Android apps. In *ASE* (2016), IEEE, pp. 756–761.

[28] Li, Y., Tan, T., Sui, Y., and Xue, J. Self-inferencing reflection resolution for Java. In *ECOOP* (2014), Springer, pp. 27–53.

[29] Li, Y., Tan, T., and Xue, J. Effective soundness-guided reflection analysis. In *SAS* (2015), Springer, pp. 162–180.

[30] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B.-Y. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In defense of soundiness: A manifesto. *CACM 58*, 2 (2015), 44–46.

[31] Micco, J. Tools for continuous integration at Google scale. In *Google Tech Talk, Google Inc* (2012).

[32] Öqvist, J., Hedin, G., and Magnusson, B. Extraction-Based Regression Test Selection. In *PPPJ* (2016), pp. 1–10.

[33] Orso, A., Shi, N., and Harrold, M. J. Scaling Regression Testing to Large Software Systems. In *ACM SIGSOFT Software Engineering Notes* (2004), pp. 241–251.

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

[35] Ren, X., Shah, F., Tip, F., Ryder, B. G., Chesley, O., and Dolby, J. Chianti: A prototype change impact analysis tool for Java. Tech. Rep. DCS-TR-533, Rutgers University CS Dept., 2003.

[36] Rothermel, G., and Harrold, M. J. A safe, efficient algorithm for regression test selection. In *ICSM* (1993), pp. 358–367.

[37] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *TOSEM 6*, 2 (1997), 173–210.

[38] Smaragdakis, Y., Balatsouras, G., Kastrinis, G., and Bravenboer, M. More sound Static Handling of Java Reflection. In *APLAS* (2015), pp. 485–503.

[39] Srivastava, A., and Thiagarajan, J. Effectively Prioritizing Tests in Development Environment. In *ISSTA* (2002), pp. 97–106.

[40] STARTS Webpage. `https://github.com/TestingResearchIllinois/starts`.

[41] Thies, A., and Bodden, E. Refaflex: safer refactorings for reflective Java programs. In *ISSTA* (2012), pp. 1–11.

[42] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: A survey. *STVR 22*, 2 (2012), 67–120.

[43] Zhang, L., Kim, M., and Khurshid, S. Localizing failure-inducing program edits based on spectrum information. In *ICSM* (2011), pp. 23–32.

[44] Zhang, S., Jalali, D., Wuttke, J., Mucslu, K., Lam, W., Ernst, M. D., and Notkin, D. Empirically Revisiting the Test Independence Assumption. In *ISSTA* (2014), pp. 385–396.