# Reflection-Aware Static Regression Test Selection

AUGUST SHI, University of Illinois at Urbana-Champaign, USA
MILICA HADZI-TANOVIC, University of Illinois at Urbana-Champaign, USA
LINGMING ZHANG, University of Texas at Dallas, USA
DARKO MARINOV, University of Illinois at Urbana-Champaign, USA
OWOLABI LEGUNSEN, University of Illinois at Urbana-Champaign, USA

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 *static* or *dynamic* analysis techniques. Our prior study showed that static and dynamic RTS perform similarly for medium-sized Java projects. However, the results of that prior study also showed that static RTS can be *unsafe*, missing to select tests that dynamic RTS selects, and that reflection was the only cause of unsafety observed among the evaluated projects.

In this paper, we investigate five techniques—three purely static techniques and two hybrid static-dynamic techniques—that aim to make static RTS safe with respect to reflection. We implement these reflection-aware (RA) techniques by extending the reflection-unaware (RU) class-level static RTS technique in a tool called STARTS. To evaluate these RA techniques, we compare their end-to-end times with RU, and with RetestAll, which reruns all tests after every code change. We also compare safety and precision of the RA techniques with Ekstazi, a state-of-the-art dynamic RTS technique; precision is a measure of unaffected tests selected.

Our evaluation on 1173 versions of 24 open-source Java projects shows negative results. The RA techniques improve the safety of RU but at very high costs. The purely static techniques are safe in our experiments but decrease the precision of RU, with end-to-end time at best 85.8% of RetestAll time, versus 69.1% for RU. One hybrid static-dynamic technique improves the safety of RU but at high cost, with end-to-end time that is 91.2% of RetestAll. The other hybrid static-dynamic technique provides better precision, is safer than RU, and incurs lower end-to-end time—75.8% of RetestAll, but it can still be unsafe in the presence of test-order dependencies. Our study highlights the challenges involved in making static RTS safe with respect to reflection.

CCS Concepts: • **Software and its engineering** → *Automated static analysis*; *Software testing and debugging*; *Software evolution.*

Additional Key Words and Phrases: regression test selection, reflection, regression testing, static analysis, class firewall, string analysis

**ACM Reference Format:**
August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 187 (October 2019), 29 pages. https://doi.org/10.1145/3360613

## 1 INTRODUCTION

Regression testing [Yoo and Harman 2012] reruns tests after every code change to check against regression bugs that break previously working functionality. Regression testing is important during

Authors' addresses: August Shi, University of Illinois at Urbana-Champaign, USA, awshi2@illinois.edu; Milica Hadzi-Tanovic, University of Illinois at Urbana-Champaign, USA, milicah2@illinois.edu; Lingming Zhang, University of Texas at Dallas, USA, lingming.zhang@utdallas.edu; Darko Marinov, University of Illinois at Urbana-Champaign, USA, marinov@illinois.edu; Owolabi Legunsen, University of Illinois at Urbana-Champaign, USA, legunse2@illinois.edu.

2475-1421/2019/10-ART187
https://doi.org/10.1145/3360613

software evolution. However, running all tests after every change—i.e., *RetestAll*—can be expensive both in terms of disrupting developers' workflow (developers have to wait for test results) and requiring machine time (running potentially many tests or long-running tests). Companies, e.g., Facebook, Google and Microsoft, publicly reported ever-growing costs of regression testing and their work to reduce the costs [Elbaum et al. 2014; Esfahani et al. 2016; Gupta et al. 2011; Herzig and Nagappan 2015; Machalica et al. 2019; Srivastava and Thiagarajan 2002; York 2011; Zhang 2018].

Regression test selection (RTS) [Chen et al. 1994; Gligoric et al. 2015b; Legunsen et al. 2016, 2017; Machalica et al. 2019; Öqvist et al. 2016; Orso et al. 2004; Ren et al. 2003; Rothermel and Harrold 1993, 1997; Zhang 2018; Zhang et al. 2011] is an approach to reduce regression testing costs by rerunning only *affected tests* whose pass/fail outcome may flip as a result of code changes. That is, RTS saves time that would have been spent on needlessly rerunning tests whose outcome cannot flip. Generally, an RTS technique first finds the dependencies that each test requires on one program version. Then, given code changes that result in a new program version, the technique selects, as affected, all tests for which at least one dependency changed. It is desirable that an RTS technique be *safe* [Rothermel and Harrold 1993], i.e., select to rerun all affected tests, so it does not miss to catch regression bugs. For performance reasons, an RTS technique should be *precise*—it should not select to rerun tests that are not affected.

RTS can collect dependencies statically or dynamically, and previous research has mostly focused on dynamic approaches [Gligoric et al. 2015b; Orso et al. 2004; Rothermel and Harrold 1997; Yoo and Harman 2012; Zhang 2018; Zhang et al. 2011]. Dependencies can also be collected at various granularity levels. Recently, both Ekstazi [Gligoric et al. 2015a,b] (the state-of-the-art dynamic RTS technique for Java) and STARTS [Legunsen et al. 2016, 2017] (a purely static RTS technique) showed that performing RTS at the class level gave better speedup 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 started to adopt Ekstazi [Gligoric et al. 2015a] and integrated it in the build systems of some open-source projects, like Apache Camel [Apache Software Foundation 2019a], Apache Commons Math [Apache Software Foundation 2019b], and Apache CXF [Apache Software Foundation 2019c].

Despite the recent progress and adoption, dynamic RTS has been known to have limitations due to its reliance on dynamic test dependency collection [Chen et al. 1994]. For example, in cases of exceptions or non-exhaustive thread-schedule exploration, dynamic RTS may fail to collect complete coverage and result 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 Facebook, Google and Microsoft). For example, the authors of Ekstazi reported on their industrial collaboration for applying RTS at Samsung, in which they used static RTS because of the aforementioned limitations of dynamic RTS [Çelik et al. 2018].

Static RTS does not suffer from these problems of dynamic analysis, because it performs static analysis that does not require instrumenting the code or running the tests. Static class-level RTS finds test dependencies statically by over-approximating test dependencies through constructing and traversing an *Inter-type Relation Graph* (IRG) [Orso et al. 2004], in which nodes are types (e.g., Java classes, interfaces, enums, etc.) and edges represent use or inheritance relationships among nodes. In our prior study [Legunsen et al. 2016], we showed that static RTS [Kung et al. 1995] can perform similarly as dynamic RTS at the class level for medium-sized Java projects. However, the results of that prior study also showed that static RTS was not safe; in a small number of cases, it missed to select tests that dynamic RTS selected. The only cause of static RTS unsafety that we observed during the experiments in our prior study [Legunsen et al. 2016] was reflection.

Reflection is widely used in object-oriented programming languages and allows applications to examine or modify their runtime behavior [Chiba 2000; Guimarães 1998]. For example, in Java, one class, `A`, can pass the *name* of another class as a string, `"B"`, to some API that creates instances of `B`, which can then be used by instances of `A` at runtime. The Java standard library (JSL) methods for dynamically creating instances of `B` from `A` include `Class.forName` and `Class.newInstance`, which allow creating objects that represent classes from a string and creating instances of those classes, respectively. Although reflection is a powerful feature that makes code more extensible, it poses significant challenges for static analysis [Bodden et al. 2011; Landman et al. 2017; Li et al. 2016a,b, 2014, 2015b; Livshits et al. 2015; Smaragdakis et al. 2015; Thies and Bodden 2012]. In particular, for reflection-unaware (RU) static class-level RTS, the IRG would not contain *reflective edges*, such as from `A` to `B`. Thus, RU could miss to select some affected test. Furthermore, many Java projects use reflection either directly in their own code or indirectly through third-party libraries they depend on [Landman et al. 2017; Li et al. 2016b].

We investigate five reflection-aware (RA) techniques that aim to 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*. The purely static techniques over-approximate test dependencies to account for reflection, while the hybrid static-dynamic techniques find more precise dependencies as they rely on light-weight dynamic instrumentation to recover reflective edges.

We measure the benefits and costs of these RA techniques relative to RetestAll, and to reflection-unaware static RTS [Legunsen et al. 2016], henceforth called *RU Analysis*. We previously implemented RU Analysis in our open-source tool, STARTS [Legunsen et al. 2017; STARTS Team 2018], and we build the RA techniques on the RU Analysis in STARTS. All the RA techniques will not select fewer tests than the baseline RU Analysis, as they attempt to make static RTS safer by adding potential reflective edges. We compare the safety and precision of various static and hybrid techniques against Ekstazi [Gligoric et al. 2015a,b], a state-of-the-art dynamic class-level RTS technique. We compare the end-to-end time of our techniques against RetestAll and RU Analysis, and we evaluate the reflection-aware techniques on 1173 versions of 24 open-source Java projects.

Overall, our evaluation shows negative results—current techniques for making static RTS reflection-aware incur very high costs. Naïve Analysis and String Analysis are *completely ineffective*; they make static RTS safe but at the cost of always rerunning all tests after every code change. These two techniques are slower than RetestAll; they spend/waste time performing analysis, including of JSL classes. The third purely static RA technique, Border Analysis, can be as safe as Ekstazi and runs faster than RetestAll in our experiments, but the average reduction in end-to-end time is rather small—85.8% of RetestAll time. Among hybrid static-dynamic techniques, Dynamic Analysis is safe but selects many more tests than RU Analysis, 67.0% versus 34.1% of all tests, respectively, and it has end-to-end time that is, on average, 91.2% of RetestAll time. Per-test Analysis is much more precise than Dynamic Analysis and all purely static RA techniques, selecting 36.9% of all tests versus the 34.1% that RU Analysis selects. Although Per-test Analysis has better end-to-end time (75.8% of RetestAll time) than Dynamic Analysis, Per-test Analysis incurs non-negligibly higher end-to-end time than RU Analysis (69.1% of RetestAll time). More importantly, Per-test Analysis dependency collection has two issues. A fast collection of dependencies (with all tests run in one JVM) makes Per-test Analysis still unsafe with respect to reflection when tests have order dependencies [Gyori et al. 2015; Lam et al. 2019, 2015; Shi et al. 2019; Zhang et al. 2014]. Running each test in its own JVM to collect dependencies could alleviate the test-order dependency problem, but it is slower than running all tests in one JVM [Bell and Kaiser 2014; Bell et al. 2015].

Beyond evaluating which *tests* are selected by various techniques, an additional contribution of this paper is an evaluation of the *test dependencies* computed by various techniques. (All techniques 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 understand why a technique selects or misses to select a test. No prior study of RTS [Yoo and Harman 2012], including our own recent studies [Legunsen et al. 2016; Zhang 2018; Zhu et al. 2019], evaluated the computed test dependencies. Evaluating test dependencies can help to understand whether static RTS was safe in prior studies in the presence of reflection because (1) test dependencies are not under-approximated by missing reflective edges, or (2) test dependencies are under-approximated, but actual code changes do not frequently touch dependencies that are *only* reachable via reflective edges. In brief, there was no previous evaluation of test dependencies that an RTS technique may be missing. Our evaluation of test dependencies reveals 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, we find RU Analysis *actually* misses much fewer tests. We also find that Border Analysis, our best purely static reflection-aware RTS technique, as well as Dynamic Analysis, which is a hybrid static-dynamic technique, do not miss any test dependency that Ekstazi finds.

This paper makes the following contributions:

★ **Reflection-Aware Static RTS.** We are the first to investigate techniques that aim to make static RTS as safe as dynamic RTS with respect to reflection. Two techniques that we evaluate—Border Analysis and Dynamic Analysis—are as safe as Ekstazi in our experiments.

★ **Analysis of RTS at the Level of Dependencies.** We present the first analysis of RTS in terms of test dependencies and not just selected tests. While using RU Analysis leads to many tests missing some test dependencies, making RU Analysis reflection-aware through Border Analysis and Dynamic Analysis leads to no test missing any dependency in our experiments.

★ **Implementation.** We implement five reflection-aware static RTS techniques by extending our publicly available Maven-based tool STARTS [Legunsen et al. 2017; STARTS Team 2018], which already performs RU Analysis.

★ **Empirical Study.** We present an empirical study of reflection-aware static RTS on 1173 versions of 24 open-source Java projects. The results are negative, showing that making static RTS reflection-aware is currently impractical.

## 2 BACKGROUND

In this section, we provide background on static regression test selection (SRTS) and reflection. We also show, by means of a motivating example, how the reflection-unaware static RTS technique, RU Analysis, can be unsafe due to its inability to handle reflection. Recall that an RTS technique is unsafe if it fails to select affected 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 [Badri et al. 2005; Kung et al. 1995; Orso et al. 2004; Ren et al. 2003]. We performed an extensive prior study of SRTS techniques that track dependencies at both class and method levels [Legunsen et al. 2016]. In that study, we selected to run tests at the level of test classes as opposed to test methods (selecting to run all test methods in an affected test class)[1] Experimental results from our prior study [Legunsen et al. 2016] showed that method-level SRTS based on method call graphs is much more imprecise/unsafe and costly than class-level SRTS based on class-level dependencies. (Class-level SRTS was better than method-level SRTS for a number of call-graph analyses: CHA, RTA, 0-CFA, and 0-1-CFA.) Moreover, the class-level SRTS was comparable to the state-of-art

[1]In the rest of this paper, we say "test" to mean a test class unless otherwise noted.

class-level dynamic RTS technique, Ekstazi [Gligoric et al. 2015b], on several medium-sized projects. Therefore, we focus on evaluating class-level SRTS in the presence of reflection. Class-level SRTS originates from the *firewall* notion [Leung and White 1990], which aims to identify code modules that may be impacted by code changes. Kung et al. [1995] 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. [2004] adapted class firewall to the Java language by handling interfaces.

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) $\tau$ is impacted by a changed type $\tau_c$ iff $\tau$ can transitively reach $\tau_c$ via a sequence of (use or inheritance) edges, denoted as $\tau_c \in \{\tau\} \circ E^*$, where $E$ denotes the set of all edges in the program's IRG, $^*$ denotes the reflexive and transitive closure, 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 a changed type, i.e., $\mathit{firewall}(\mathcal{T}_c) = \mathcal{T}_c \circ (E^{-1})^*$, 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 affected tests, $T_a = T \cap \mathit{firewall}(\mathcal{T}_c)$. In theory, class-level SRTS should be safe since it selects all tests that could be impacted by the code changes. However, in practice, like most other static analyses [Livshits et al. 2015], class-level SRTS can be unsafe, specifically if it misses edges. For example, as we showed in our prior study [Legunsen et al. 2016], reflection-unaware class-level SRTS can miss to select some affected tests because the IRG does not have edges that can only be reached via reflection.

## 2.2 Reflection

The main feature of reflection that is relevant to class-level SRTS is the ability 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 constructed dynamically without static use of the class name. The static analysis used in the reflection-unaware class-level SRTS fails 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 that are relevant for class-level SRTS are those that create Java `Class` objects either from string input representing the name of the class or from bytecode that defines the class. We call such `Class`-creating methods *reflection methods*. The returned `Class` can be used to create instances at runtime. Through manual inspection of the Java standard library (JSL) reflection API, we identify four reflection methods through which all class-related reflection usage eventually happens: `Class.forName`, `ClassLoader.loadClass`, `ClassLoader.findSystemClass`, and `ClassLoader.defineClass`. The first three reflection methods take a `String` name and return the `Class` represented by that name. The fourth reflection method takes a `byte` array and returns the `Class` defined by that `byte` array. We find that all other possible `Class`-related uses of reflection either eventually invoke these four reflection methods or refer to some `Class` that is statically referenced in the code, which static analysis can detect. Therefore, focusing on detecting usages of these four reflection methods suffices to detect all reflection usages for *class*-level SRTS.

## 2.3 Motivating Example

Figure 1 presents a code snippet showing example code and tests. In the example, `L` is a class in the JSL; `A1`, `A2`, `A3`, and `A4` are classes within the application, and test classes `T1`, `T2`, `T3`, and `T4` form the test suite. Suppose that class `A4` is changed (marked in a gray background). Using the reflection-unaware analysis (RU Analysis), i.e., the 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(a) shows the static IRG based on RU Analysis, where `T4` would be the only test class affected by this change, and it is included in the class firewall (gray area in Figure 2(a)). However,

```
// JSL code
class L {
  public void p() {} // empty method

  public void m(String s) {
    // reflection method invocation
    Class c = Class.forName(s);
    ...
  }
}

// source code
class A1 extends L {
  void m1() {
    m("A4");
  }

  void m1(boolean b) {
    m("A" + getNum(b));
    ...
  }

  private String getNum(boolean b) {
    return b ? "1" : "3";
  }
}

class A2 {
  void m2() {} // empty method
}

class A3 {
  static void m3() {
    (new L()).p();
  }
}
class A4 {...} // changed code
```

```
// test code
class T1 {
  @Test
  void t1() {
    A1 a1 = new A1();
    a1.m1();
    a1.m1(true);
  }
}

class T2 {
  @Test
  void t2() {
    A2 a2 = new A2();
    a2.m2();
  }
}

class T3 {
  @Test
  void t3() {
    A3.m3();
  }
}

class T4 {
  @Test
  void t4() {
    A4 a4 = new A4();
  }
}
```

Fig. 1. Example code to illustrate our techniques

selecting only `T4` is unsafe, as more tests depend, via reflection, on the changed class `A4`. In the example, `T1.t1` creates an instance of `A1` and invokes `A1.m1`, which invokes `L.m`, which in turn uses a reflection method (`Class.forName`) to construct an instance of `A4`. Therefore, `T1` also depends on `A4`, but since RU Analysis is reflection-unaware, it fails to select `T1`, and is unsafe.

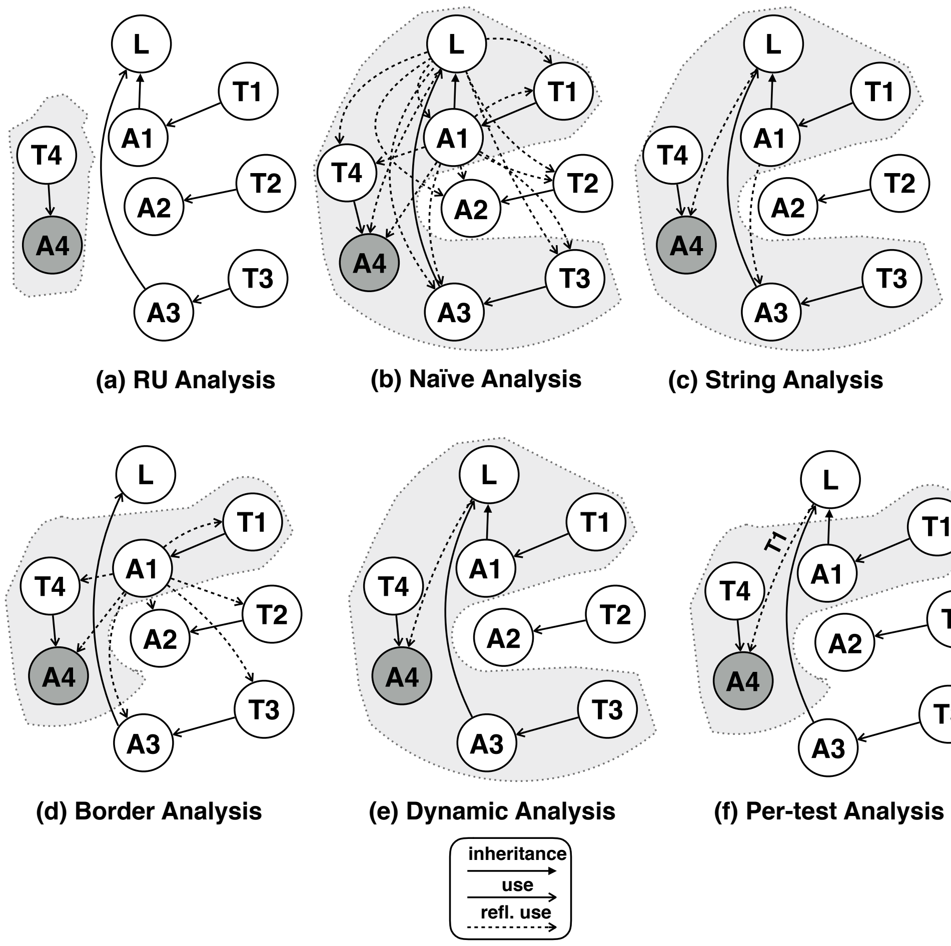

Fig. 2. Illustration for various reflection-aware analyses

## 3 REFLECTION-AWARE STATIC REGRESSION TEST SELECTION

We describe the five techniques that we use 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 describe three purely static reflection-aware techniques that can make SRTS safer with respect to reflection.

*3.1.1 Naïve Analysis.* The simplest, but the most imprecise, approach to detecting reflective edges is what we call Naïve Analysis, which adds edges to the IRG from each class that invokes a reflection method to all other classes. For ease of presentation, we often refer to edges from a class to all other classes in the IRG as an edge to the special node, "$*$". Figure 1 shows that both A1 and L can invoke the reflection method `Class.forName` (`A1` can do so if either of its overloaded `m1` methods is invoked). Thus, we add to the IRG edges from `A1` and `L` to all other nodes in the graph. The IRG containing additional reflective edges from Naïve Analysis is shown in Figure 2(b). There, all tests 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, does not statically depend on the changed `A4`, and need not be selected. In fact, our experiments show that Naïve Analysis always selects all tests after every change (because test classes depend directly or transitively on JSL classes, through which they may invoke a reflection method, e.g., `java.lang.Class`).

*3.1.2 String Analysis.* String Analysis [Christensen et al. 2003; Grech et al. 2018; Kirkegaard et al. 2004; Li et al. 2015a] is a static analysis technique that can approximate potential target classes in reflective edges, based on the `String`-valued class name passed to reflection methods. In Figure 1, String Analysis can determine that the reflection method call site in class `L` (line 7) can only receive the name `"A4"` from invoking method `L.m` on line 15 of `A1`. Also, for the invocation of `L.m` on line 19 of `A1`, String Analysis approximates the received class name to match the regular expression `"A1|A3"`. As shown in Figure 2(c), String Analysis then adds these reflective edges from `L` to `A4` and from `A1` to `A3` to the IRG. A self-edge from `A1` to `A1` need not be added to the IRG since class firewall computes a reflexive and transitive closure. 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 also imprecisely selects `T3` because analyzing JSL classes (e.g., `L`) results in many commonly-used internal JSL classes reaching client code classes. In this example, although `T3` uses `L` in a way that cannot lead to invoking the reflection method `Class.forName`, `T3` 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 find that String Analysis incurs large imprecision because it needs to analyze the internals of the JSL. 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 JSL without also including additional usage context from those call sites. To illustrate, consider JSL internal class, `java.lang.Class`, which uses reflection methods 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., `"*"`. However, 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 or methods that eventually invoke a reflection method. Unfortunately, adding an edge from `java.lang.Class` to all other nodes in the IRG causes each class to depend on every other class, with the result that all test classes are selected after any code change.

*3.1.3 Border Analysis.* The severe imprecision of String Analysis for recovering reflective edges is due to the fact that most commonly-used classes in the JSL transitively reach classes that are connected to `"*"` in the IRG. We propose *Border Analysis*, which avoids analyzing classes in the JSL, while still aiming to be safe. Our observation is that not all, but only a subset of, methods from the JSL, when invoked, can eventually invoke a reflection method. We define a *border method* as a public method in the JSL that, when invoked, *may* eventually invoke a reflection method. A class that invokes a border method, either directly or via third-party library code, may eventually invoke a reflection method. Therefore, Border Analysis only adds to the IRG additional edges that connect any non-JSL class that invokes a border method to `"*"`. Border Analysis takes as input a set of border methods identified offline *a priori* and avoids subsequent re-analysis of the JSL internals during the test selection process. Border methods can be identified automatically or through manual inspection. In Figure 1, method `L.m`, which is in the JSL, is a border method, because invoking it may (in fact, must) eventually invoke the reflection method `Class.forName`. As shown in Figure 2(d), Border Analysis adds reflective edges to the IRG from `A1` to all other classes because `A1` is the only class invoking the identified border method in the example. Thus, Border Analysis precisely selects tests `T1` and `T4`, without re-analyzing the JSL internals.

In theory, a developer with domain knowledge could manually identify the border methods specific to their project. However, Border Analysis would be more practical if there are automated approaches for determining border methods, which do not require manual developer work. As discussed in Section 2.2, we recognized only four reflection methods in the JSL (`Class.forName`, `ClassLoader.loadClass`, `ClassLoader.findSystemClass`, and `ClassLoader.defineClass`) that directly create classes through reflection. We propose different approaches for finding border methods which, when invoked, can eventually invoke one of these four reflection methods.

First, we consider a purely static approach to finding *all* border methods. The main idea is to statically find out what public JSL methods can eventually invoke any of the aforementioned four reflection methods. We perform a call-graph analysis using every public method in the JSL as an entry point, and output, as border methods, public JSL methods that can reach any of the four reflection methods in the call graph. However, such a static analysis to find border methods can be very imprecise and result in many more border methods than can eventually invoke reflection methods at runtime. Our analysis found 55453 such methods (of 124196 public JSL methods)! Therefore, Border Analysis using all border methods that are obtained from statically analyzing the JSL can be very imprecise and select almost all tests always, similar to String Analysis. We call this Border Analysis variant *static Border Analysis*: it uses, as border methods, the full set of public methods that are statically computed from the JSL independently of any software project.

Statically analyzing the JSL for all possible border methods is one extreme; it can be too imprecise. As another extreme, we also investigate *four-method Border Analysis*—Border Analysis that uses as border methods only the four reflection methods. Code that invokes any of these four reflection methods uses reflection. However, using only these four reflection methods as border methods can lead to unsafety, because it excludes cases where a project does not directly invoke a reflection method but instead calls a border method that can eventually invoke a reflection method at runtime.

We further propose a heuristic to automatically determine a set of border methods that falls between the two extremes of static Border Analysis and four-method Border Analysis. Before rerunning any of our experiments to evaluate Border Analysis, we perform a one-time experiment to identify what border methods to use for each project. We determine border methods automatically by instrumenting the execution of all tests in the earliest version of each of the projects in our experiments to capture and process the call stack at a call site of any of the four reflection methods. The call stack is processed as follows: we find the last method in the stack from a non-JSL class to call a method in a JSL class. This method from a JSL class that is called by the last non-JSL method is returned as a border method. Because we observe that these border methods invoke a reflection method in at least one calling context, we over-approximate that all border methods detected this way can eventually invoke a reflection method in *all* calling contexts. We refer to Border Analysis that uses such dynamically obtained border methods as *dynamic Border Analysis*. However, we acknowledge potential imprecision, because the border methods identified dynamically might only eventually invoke a reflection method in some, but not all, calling contexts. Note that our collection of border methods is performed once, offline. As code bases evolve, including changes to the tests themselves, the changes could be such that existing border methods no longer suffice for safe SRTS. In such cases, developers need to rerun the analysis to identify current border methods. In brief, dynamic Border Analysis that uses border methods identified from a version of a project is only safe to the degree that the border methods are up to date.

Finally, some automatically dynamically-identified border methods for a project are such that they can only add edges to the IRG that RU Analysis would already find. We therefore also perform a limit study with *minimal Border Analysis*, a variant of dynamic Border Analysis that uses a manually identified subset of border methods from dynamic Border Analysis that do not lead to finding edges that RU Analysis already finds; we refer to this subset of border methods as *minimal border methods*. Section 4.4 discusses our manual identification of minimal border methods.

## 3.2 Hybrid Static-Dynamic Reflection Detection

We describe two hybrid static-dynamic techniques that make SRTS safer with respect to reflection.

*3.2.1 Dynamic Analysis.* We can obtain reflective edges dynamically as was done in previous work on reflection analysis for other testing or analysis tasks, e.g., by Bodden et al. [2011] and Thies and Bodden [2012]; we refer to this technique as Dynamic Analysis. The idea is to execute the tests while instrumenting only the aforementioned four reflection methods to record the classes constructed from invocations of the reflection methods. Then, for each invocation of a reflection method, Dynamic Analysis adds an edge to the IRG from the class that invoked the reflection method to the class constructed by the reflection method.

Instrumenting the reflection methods during test executions for the example in Figure 1 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`. After adding both recovered edges to the IRG, shown in Figure 2(e), SRTS determines that `T1`, `T3`, and `T4` should be selected when `A4` changes. Dynamic Analysis is a hybrid static-dynamic RTS technique, and it can lead to more precise test selection than Naïve Analysis or String Analysis. Dynamic Analysis uses a 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 1, Dynamic Analysis finds only from executing `T1` that there is a 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 precision 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 certain test classes to have spurious paths to some changed class in the IRG. 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 other test classes. Per-test Analysis reduces the imprecision of Dynamic Analysis by only adding reflective edges to the IRG when computing dependencies for the test class during whose execution those reflective edges were recovered. Reflective edges recovered while executing each test class are only used to find dependencies 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(e), recall that Dynamic Analysis selects to rerun `T1`, `T3`, and `T4`, because it added the reflective edges recovered from executing all tests to the IRG. 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 it adds this edge to the IRG only when computing dependencies for `T1` (Figure 2(f) labels the reflective edge with the test that necessitates that edge). The result is that Per-test Analysis does not imprecisely select `T3` when `A4` changes.

One might expect Per-test Analysis to be safe because it seemingly adds the right reflective edges for each test as needed; however, it is possible for Per-test Analysis to miss to select tests due to test-order dependencies [Gyori et al. 2015; Lam et al. 2019, 2015; Shi et al. 2019; Zhang et al. 2014]. Consider the example code in Figure 3. The `Server` class contains a static field, `sessClz`, of type `Class`. Field `sessClz` is initialized in the static constructor, using `Class.forName`. The two test classes

```
class Server {

 static Class sessClz;

 static {
   try {
     sessClz = Class.forName("SessionImpl");
   } catch (Exception ex) {
      ...
   }
 }

 public static Session createSession()
   throws Exception {
  return (Session)sessClz.newInstance();
 }
}
```

```
class T1 {

 void t1()
   throws Exception {
  Session sess =
    Server.createSession();
  ...
 }
}

class T2 {

 void t2()
   throws Exception {
  Session sess =
    Server.createSession();
  ...
 }
}
```

Fig. 3. Example showing that Per-test Analysis can be unsafe

`T1` and `T2` both call the `createSession` static method from `Server` that uses the static field `sessClz` to reflectively create a new instance of type `SessionImpl`. Statically, there is a use edge from `T1` and `T2` to `Server` and `Session`. When tests are run using Per-test Analysis, assume `T1` is run first. During `T1` execution, `Server` is referenced and the static constructor is invoked. At this time, Per-test Analysis records the reflective edge from `Server` to `SessionImpl`, due to `Class.forName`. However, when `T2` is run afterwards, `Server`'s static constructor is not invoked, as `Server` was already used in `T1`, and both tests are run in the same JVM. Therefore, Per-test Analysis would *not* record the reflective edge for `T2`. While both tests eventually end up using an instance of `SessionImpl` due to calling `createSession`, only `T1` would have the reflective edge. If `SessionImpl` were to change, Per-test Analysis would not select to run `T2`, even though `T2` uses `SessionImpl`. While this examples assumes that `T1` runs first, the situation is dual if `T2` runs first: `T2`, but not `T1`, would have the reflective edge from `Server` to `SessionImpl`. One way for both tests to receive the same reflective edge would be to run each test in a separate JVM, but using a separate JVM for individual test executions is costly [Bell and Kaiser 2014]. Another way would be to use a much more involved runtime data-flow analysis that could track dependencies even when all tests are run in the same JVM [Bell et al. 2015; Gambi et al. 2018], but would add even more overhead, and our results show that Per-test Analysis already has a higher end-to-end time than RU Analysis, relative to RetestAll.

## 4 IMPLEMENTATION

We implement all five reflection-aware techniques by extending our open-source Maven plugin for RU Analysis, STARTS [STARTS Team 2018]. We describe the STARTS plugin that we extend, and we provide details about how we extend STARTS to implement the reflection-aware techniques.

## 4.1 RU Analysis in STARTS

STARTS [Legunsen et al. 2017] performs RU Analysis by implementing the class firewall technique described in Section 2.1. It works in three main steps: (1) **Change Computation:** STARTS uses the bytecode comparison feature from Ekstazi [Gligoric et al. 2015b] to compute, as changed, only bytecode files where some non-debug-related information changed; (2) **Graph Construction:** STARTS uses jdeps [Oracle 2018] to quickly parse all the bytecode from the project code and its third-party libraries to discover dependency relationships among classes in the project. STARTS then uses these dependencies to construct the IRG; (3) **Graph Traversal:** Given the IRG and the nodes that changed since the last version, STARTS finds, as affected tests, test classes whose IRG nodes can transitively reach changed nodes. For each reflection-aware technique, we merely extend the graph construction step to recover and add reflective edges to the constructed IRG. Each technique uses a different approach for recovering reflective edges, as described in detail below. We also modified the graph traversal step to include logic for considering the special node "$*$", needed by the purely static reflection-aware techniques. The meaning of "$*$" is that if a test reaches "$*$" in the IRG, that test is selected to be rerun if any class changes.

Note that for RU Analysis, it is unnecessary to track any classes in the JSL or in third-party libraries, and we do not do so. It is assumed that code in both the JSL and in third-party libraries do not change, and so they would not affect RTS selection. Further, as we showed in our prior study [Legunsen et al. 2016], excluding libraries from the analysis of a class-level SRTS cannot introduce non-reflection related safety issues. However, for reflection-aware techniques, it is necessary to consider both the JSL and third-party libraries, as the JSL and the third-party libraries can use reflection to refer back to classes in the project code that calls them.

## 4.2 Naïve Analysis

For Naïve Analysis, after creating the initial IRG, STARTS uses ASM [OW2 Consortium 2018] to parse all classes, including those in the JSL and third-party libraries, to statically find uses of the reflection methods: `Class.forName`, `ClassLoader.loadClass`, `ClassLoader.findSystemClass`, and `ClassLoader.defineClass`. Classes that use reflection methods get edges to "$*$" in the IRG.

## 4.3 String Analysis

For String Analysis, we use the JSA tool [Christensen et al. 2003], 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 load the same JSL classes. Therefore, to speed up String Analysis experiments, we run JSA offline, only once *a priori* (before performing experiments) for all JSL classes and third-party, non-JSL classes. We cache the reflective edges recovered by JSA to be reused during String Analysis experiments. Finally, we extend STARTS to reuse edges during String Analysis experiments for each project, during which STARTS runs JSA only on classes in the project.

## 4.4 Border Analysis

For static Border Analysis, border methods are all public JSL methods that can statically reach one of the four reflection methods, obtained from our project-independent CHA call-graph analysis of the JSL. Our call-graph analysis uses every public JSL method as an entry point and returns the public methods that reach any of the four reflection methods as border methods. For four-method Border Analysis, the border methods are only the four reflection methods. Dynamic Border Analysis dynamically collects border methods by processing dynamic traces during test executions.

To collect border methods for dynamic Border Analysis, we execute the tests in the initial version of each project with a Java Agent attached to the JVM in which the tests are executed. The Java Agent instruments the four reflection methods, and analyzes the stack trace at each reflection method invocation to determine the border method—the first public JSL method invoked in a trace in which one of the four reflection methods is eventually invoked (Section 3.1.3). The Java Agent outputs the collected border methods in a `ShutdownHook`, which is invoked during JVM shutdown, after all tests have been executed. In our experiments, we collect border methods per experimental subject, using the oldest version among those that we selected. The border methods from each project are then reused in the experiments for all subsequent versions of that project. This automated way of collecting border methods may not detect all possible border methods, because it is limited by test coverage in the initial version. Further, developers may change the code and tests in the future, modifying coverage substantially enough to require update of border methods. However, we note that the initial automated collection of border methods is sufficient for safe SRTS, until developers make such substantial changes. At that point the developers can collect border methods again. Developers could also update the border methods periodically during off-peak periods, e.g., overnight or during weekends. Collecting border methods in this automated way is relatively fast as the instrumentation is light and only requires running the tests once. However, we still do not want to collect the border methods as developers are performing regression testing and want to quickly see what tests should be run based on the changes made. Therefore, in our experiments, dynamic collection of border methods is run offline, and not during regression testing.

Minimal Border Analysis requires further manual filtering of the automatically-collected border methods to create a set of minimal border methods. The goal is to filter out reflective edges that unnecessarily connect classes to "∗" when RU Analysis can already determine the concrete nodes involved. We select 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. We keep only border methods that we think will always create reflective edges that RU Analysis does not find. Two co-authors manually inspected and selected minimal border methods, using the border methods collected for dynamic Border Analysis as a starting point. We divided the subject programs into two groups, each of which was assigned to one co-author for inspection. Each co-author 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 border method that is not a minimal border method is `java.lang.Enum.valueOf`; it 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 finds.

In all Border Analysis variants, the selected border methods are passed as input to STARTS, which uses the border methods as follows. First, STARTS creates an initial IRG. Then, for each class in the project, STARTS uses ASM 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, STARTS adds these recovered reflective edges to the initial IRG, and uses the augmented IRG for RTS. Edges between internal JSL classes are not added to the IRG for Border Analysis.

## 4.5 Dynamic Analysis

To recover reflective edges in Dynamic Analysis, STARTS performs very lightweight instrumentation *during* test execution in each version. The instrumentation is similar to that used for finding border methods (Section 4.4), 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 STARTS discovers a `Class` being returned from the invocation of a reflection method, STARTS records a reflective edge from the calling class to the returned `Class`. STARTS writes all reflective edges collected during test execution to disk in a `ShutdownHook`, invoked on JVM shutdown after executing all tests. Initially, on the first version, STARTS runs all tests and collects reflective edges from running all tests. In the next version, STARTS uses the stored edges to augment the initial IRG that it constructs, resulting in the final IRG used for RTS. In this next version, STARTS only instruments the execution of the selected tests and augments the existing reflective edges with those collected from that run, in preparation for the version after that. Dynamic Analysis has the benefit that there are no edges 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 also more lightweight than the one used in Ekstazi [Gligoric et al. 2015b]—it only instruments the four reflection methods and not all classes.

## 4.6 Per-test Analysis

For Per-test Analysis, STARTS collects reflective edges as in Dynamic Analysis. However, instead of writing out all the reflective edges when the JVM shuts down, STARTS associates the reflective edges with the test that executes the reflection method. STARTS uses a custom `JUnit RunListener` that detects when tests start and end. At the end of each test execution, the listener writes all the reflective edges collected up until then to a file associated with that test, and then it clears out all the collected reflective edges in preparation for executing the next test. After collecting edges per test, STARTS reads in the file and for each test adds only the reflective edges for that test to the initial IRG before traversing the augmented IRG to find dependencies for that test. As with Dynamic Analysis, STARTS only collects reflective edges for selected tests.

## 5 EVALUATION

We evaluate whether reflection-aware SRTS techniques can be safe (and still be faster than RetestAll). We address two research questions on the safety and precision of reflection-aware SRTS techniques:

- **RQ1:** How safe and precise is reflection-aware SRTS *test selection*, compared with Ekstazi?
- **RQ2:** How safe and precise is SRTS *test-dependency computation*, compared with Ekstazi?

Answering RQ1 and RQ2 (Sections 5.2 and 5.3), we find that Border Analysis and Dynamic Analysis are as safe as Ekstazi in our experiments. We then proceed to address these research questions related to how much faster SRTS techniques are relative to RetestAll:

- **RQ3:** How many tests does reflection-aware SRTS select, relative to RetestAll and RU Analysis?
- **RQ4:** What is the end-to-end time of reflection-aware SRTS relative to RetestAll and RU Analysis?

Finally, we analyze the size of the IRG computed by SRTS:

- **RQ5:** What are the sizes of the IRG constructed by RU Analysis and reflection-aware SRTS?

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

## 5.1 Experimental Setup

**Evaluation Subjects:** We evaluate all RTS techniques on 1173 versions of 24 open-source, Maven-based, Java projects selected from GitHub. The projects are a mix of 12 projects from our prior work [Legunsen et al. 2016] (selected because the tests run longer than 15s, on average, across all versions), and 12 additional projects that use reflection. We select these additional projects because they contain classes that directly invoke `Class.forName`, and because we can compile and successfully run the tests in at least 20 versions of these projects *without* any of our analyses in their most recent versions. We start with the most recent 500 versions from each project and select the project if we can obtain 20–50 versions where we can compile and run the tests. In exploring the most recent 500 versions, we stop either after obtaining 50 versions, or after we try all the 500

Table 1. Summary of all projects

| ID | Project SLUG | SHAs | All Tests | RTA[s] |
|---|---|---|---|---|
| P1 | apache/commons-email | 50 | 17.4 | 14.0 |
| P2 | apache/commons-codec | 50 | 52.0 | 15.4 |
| P3 | apache/incubator-fluo | 50 | 25.1 | 23.2 |
| P4 | apache/commons-compress | 50 | 118.3 | 20.9 |
| P5 | square/retrofit | 50 | 49.6 | 24.3 |
| P6 | apache/commons-collections | 50 | 160.0 | 25.3 |
| P7 | apache/commons-lang | 50 | 147.7 | 27.9 |
| P8 | apache/commons-imaging | 50 | 72.6 | 30.3 |
| P9 | robovm/robovm | 50 | 32.2 | 43.1 |
| P10 | ninjaframework/ninja | 50 | 103.7 | 44.4 |
| P11 | graphhopper/graphhopper | 50 | 128.5 | 43.0 |
| P12 | google/guice | 50 | 131.8 | 60.9 |
| P13 | apache/opennlp | 50 | 169.3 | 66.4 |
| P14 | apache/commons-io | 50 | 99.4 | 80.9 |
| P15 | apache/commons-math | 50 | 446.3 | 84.8 |
| P16 | brettwooldridge/HikariCP | 50 | 30.3 | 108.6 |
| P17 | addthis/stream-lib | 50 | 24.1 | 115.1 |
| P18 | undertow-io/undertow | 50 | 231.7 | 171.7 |
| P19 | openmrs/OpenMrs | 50 | 269.4 | 169.0 |
| P20 | opentripplanner/OpenTripPlanner | 50 | 136.1 | 224.7 |
| P21 | Activiti/Activiti | 50 | 494.5 | 292.1 |
| P22 | apache/accumulo | 23 | 342.2 | 291.4 |
| P23 | apache/commons-pool | 50 | 20.0 | 301.1 |
| P24 | aws/aws-sdk-java | 50 | 173.4 | 397.9 |
| | Avg | 48.9 | 144.8 | 111.5 |

versions. We do not consider versions past the most recent 500 as we believe it unlikely that they would be successful if we did not obtain enough versions among the most recent 500 versions. To answer the RQs in more detail, we split the 24 projects in our study into two groups: (i) 11 *small* projects, for which the end-to-end time of running all the tests is between 15s and 60s on average, and (ii) 13 *big* projects, for which the end-to-end time of running all the tests is longer than 60s, on average. Note that for RTS evaluation, the end-to-end time of running tests is a more important factor than the size of the codebase. Table 1 summarizes statistics about each of the projects in our experiments, showing for each project a project ID (which we use to reference each project in subsequent tables and plots), the project SLUG (useful for finding the project's GitHub repository), the number of versions/SHAs used in our experiments, the average number of tests run by RetestAll across all versions, and the average time to run RetestAll across all versions. The running times of these projects are representative of those commonly used in recent RTS research, which also used some of the same subjects, e.g., Çelik et al. [2017]; Dini et al. [2016]; Gligoric et al. [2015b]; Öqvist et al. [2016]; Shi et al. [2015]; Wang et al. [2018]; Zhang [2018].

**Running Experiments:** We perform all experiments involving SRTS, both reflection-aware and reflection-unaware, using STARTS [STARTS Team 2018] (Section 4). For dynamic RTS comparison, necessary only for RQ1, we use Ekstazi [Gligoric et al. 2015a,b]. We automate the run of tests across the selected versions of all projects in our study. All timing experiments are performed on Amazon

Table 2. Average violations across projects; Tests Selected Diff is diff in tests selected, Dep Diff is diff in dependencies found

| ID | Tests Selected Diff (%) | | | | | | | | | | Dep Diff (%) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *X-RU* | *RU-X* | *X-*$B_d$ | $B_d$*-X* | *X-*$B_s$ | $B_s$*-X* | *X-D* | *D-X* | *X-P* | *P-X* | *X-RU* | *X-*$B_d$ | *X-*$B_s$ |
| P1 | 0.0 | 1.7 | 0.0 | 27.9 | 0.0 | 29.9 | 0.0 | 31.9 | 0.0 | 5.8 | 17.2 | 0.0 | 0.0 |
| P2 | 0.0 | 5.8 | 0.0 | 44.2 | 0.0 | 50.2 | 0.0 | 44.2 | 0.0 | 5.8 | 0.0 | 0.0 | 0.0 |
| P3 | 0.0 | 46.7 | 0.0 | 61.9 | 0.0 | 63.1 | 0.0 | 60.2 | 0.0 | 46.7 | 25.9 | 0.0 | 0.0 |
| P4 | 0.0 | 31.7 | 0.0 | 59.7 | 0.0 | 64.2 | 0.0 | 62.2 | 0.0 | 32.0 | 1.1 | 0.0 | 0.0 |
| P5 | 0.0 | 16.2 | 0.0 | 26.2 | 0.0 | 26.2 | 0.0 | 25.9 | 0.0 | 21.5 | 6.2 | 0.0 | 0.0 |
| P6 | 0.0 | 21.2 | 0.0 | 50.6 | 0.0 | 50.8 | 0.0 | 50.8 | 0.0 | 24.3 | 0.6 | 0.0 | 0.0 |
| P7 | 0.0 | 20.5 | 0.0 | 60.8 | 0.0 | 61.5 | 0.0 | 59.5 | 0.0 | 21.7 | 0.7 | 0.0 | 0.0 |
| P8 | 0.0 | 30.6 | 0.0 | 47.6 | 0.0 | 49.9 | 0.0 | 49.9 | 0.0 | 30.6 | 0.0 | 0.0 | 0.0 |
| P9 | 0.0 | 42.2 | 0.0 | 59.5 | 0.0 | 59.5 | 0.0 | 57.5 | 0.0 | 42.2 | 3.1 | 0.0 | 0.0 |
| P10 | 36.1 | 20.9 | 0.0 | 59.6 | 0.0 | 61.3 | 0.0 | 61.1 | 6.9 | 23.9 | 45.8 | 0.0 | 0.0 |
| P11 | 0.0 | 46.9 | 0.0 | 81.2 | 0.0 | 82.3 | 0.0 | 54.5 | 0.0 | 47.3 | 10.1 | 0.0 | 0.0 |
| P12 | 0.0 | 14.1 | 0.0 | 32.0 | 0.0 | 32.0 | 0.0 | 29.4 | 0.0 | 20.8 | 12.4 | 0.0 | 0.0 |
| P13 | 0.0 | 36.4 | 0.0 | 81.4 | 0.0 | 90.4 | 0.0 | 74.4 | 0.0 | 36.7 | 2.9 | 0.0 | 0.0 |
| P14 | 0.0 | 12.6 | 0.0 | 39.5 | 0.0 | 51.0 | 0.0 | 51.0 | 0.0 | 17.0 | 0.0 | 0.0 | 0.0 |
| P15 | 0.0 | 13.7 | 0.0 | 34.5 | 0.0 | 34.6 | 0.0 | 34.6 | 0.0 | 13.8 | 4.9 | 0.0 | 0.0 |
| P16 | 0.1 | 13.4 | 0.0 | 21.3 | 0.0 | 25.4 | 2.0 | 24.8 | 17.8 | 9.0 | 79.3 | 0.0 | 0.0 |
| P17 | 0.0 | 17.2 | 0.0 | 73.7 | 0.0 | 78.1 | 0.0 | 74.2 | 0.0 | 21.1 | 0.0 | 0.0 | 0.0 |
| P18 | 7.7 | 35.7 | 0.4 | 50.0 | 0.0 | 51.0 | 0.5 | 49.7 | 4.1 | 37.7 | 99.6 | 0.0 | 0.0 |
| P19 | 21.4 | 24.2 | 0.0 | 69.6 | 0.0 | 69.8 | 0.0 | 69.7 | 15.7 | 38.7 | 74.6 | 0.0 | 0.0 |
| P20 | 0.0 | 52.0 | 0.0 | 66.8 | 0.0 | 68.5 | 0.0 | 68.5 | 0.0 | 52.5 | 36.2 | 0.0 | 0.0 |
| P21 | 0.0 | 31.7 | 0.0 | 39.6 | 0.0 | 39.6 | 0.0 | 37.2 | 0.0 | 31.8 | 85.7 | 0.0 | 0.0 |
| P22 | 0.6 | 58.9 | 0.0 | 81.5 | 0.0 | 81.7 | 0.0 | 86.8 | 0.0 | 86.5 | 35.8 | 0.0 | 0.0 |
| P23 | 0.0 | 17.3 | 0.0 | 31.0 | 0.4 | 40.1 | 0.0 | 38.6 | 0.0 | 21.5 | 55.0 | 0.0 | 0.0 |
| P24 | 73.3 | 19.9 | 71.2 | 24.2 | 71.2 | 24.4 | 72.8 | 22.3 | 73.2 | 20.0 | 32.6 | 0.0 | 0.0 |
| Avg | **5.8** | **26.3** | **3.0** | **51.0** | **3.0** | **53.6** | **3.1** | **50.8** | **4.9** | **29.5** | **26.2** | **0.0** | **0.0** |

EC2 "`m5.xlarge`" instances (four 2.5 GHz Intel Xeon Platinum 8175M processors, 16 GB of RAM, 100GB of SSD storage) running Ubuntu 16.04.03 and Oracle Java 1.8.0_144-b01.

## 5.2 RQ1: Test-Level Safety and Precision

Table 2 shows the comparison of the test-level safety and precision of RU Analysis and the reflection-aware SRTS techniques, compared with Ekstazi. 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 RTS safety and precision, we compare the *safety violations* and *precision violations* of SRTS against Ekstazi, as we defined in our prior work [Legunsen et al. 2016]: "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 use the union of tests selected by both Ekstazi and the technique to avoid errors due to division by zero where Ekstazi does not select any test but an SRTS technique selects some tests. When both select to run no tests, the value recorded is zero, as there is no safety or precision violation for such a case."[2] In Table 2 (and all later tables), we represent Ekstazi as $X$, RU Analysis as $RU$, dynamic Border Analysis as $B_d$, static Border Analysis as $B_s$, Dynamic Analysis as $D$, and Per-test Analysis as $P$. We do not show detailed results for four-method Border Analysis and minimal Border Analysis; their results are similar to dynamic Border Analysis. Columns $X$-$RU$, $X$-$B_d$, $X$-$B_s$, $X$-$D$, and $X$-$P$ show the safety violations of the SRTS techniques (compared with Ekstazi). Columns $RU$-$X$, $B_d$-$X$, $B_s$-$X$, $D$-$X$, and $P$-$X$ show the precision violations.

[2]Using just $|E|$ as the denominator can make precision violations go over 100%; the same applies for computing safety violations using just $|T|$. Alternatively, if we use $|E \setminus T|/|E|$ for safety violations and $|T \setminus E|/|T|$ for precision violations, there could be different sets of versions that are valid; some may only have $|E|$ being 0 while others have only $|T|$ being 0.

Concerning safety violations, the *X-RU* column in Table 2 shows that RU Analysis has an average safety violation of 5.8% across all projects. RU Analysis is reflection-unaware, missing to select some affected tests and therefore unsafe relative to Ekstazi. For all variants of Border Analysis, the average safety violations across all projects are the same, 3.0%. Dynamic Analysis appears unsafe as well. However, our inspection shows that *these reflection-aware techniques are as safe as Ekstazi*, although it would appear from Table 2 that the SRTS techniques are unsafe.

For the aws-sdk-java project (P24), our manual inspection shows that the safety violations in project are actually caused by imprecision in Ekstazi. 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.) Note that this imprecision in Ekstazi makes RU Analysis appear less safe than it actually is in our experiments.

For the commons-pool project (P23), only static Border Analysis appears unsafe. We find that it seems unsafe only in one version, where the test-running process timed out and not all selected tests were run. However, the tests that we report as affected in this paper are those that are rerun—these are the tests whose test-execution time we also measure. In P23, static Border Analysis is actually safe; it statically selects all affected tests—if a timeout had not happened, static Border Analysis would have run all tests that Ekstazi also runs, just as the other Border Analysis variants do.

For project undertow (P18), again, we find dynamic Border Analysis to be safe, even though it appears unsafe in Table 2. Our manual inspection shows that in one version of P18, during the Ekstazi run, automatic code generation produces bytecode files that are different from the previous version, so Ekstazi correctly selects to rerun tests that depend on the auto-generated files. However, during the dynamic Border Analysis run, the automatically generated bytecode files are the same as in the previous version, so no tests that depend on the auto-generated files are selected. The bytecode auto-generation process can be non-deterministic (the generated files can have different contents even with no changes to any developer-written code), so the two techniques simply did not start with the same set of changed files; this should not be considered a safety violation for dynamic Border Analysis. Finally, although dynamic Border Analysis is as safe as Ekstazi in our experiments, in general, code changes can cause it to become unsafe, necessitating dynamic Border Analysis to update its border methods to become safe again.

Per-test Analysis is safer than RU Analysis, but more unsafe than other reflection-aware techniques (4.9% safety violations). However, as mentioned in Section 3.2.2, violations are expected given the way the collection of reflective edges works. For example, in our inspection of the ninja project (P10), we find a case of safety violation due to the use of dependency injection and singletons. For the version pair, (`6c952f`,`116521`), the only change is to a class `CookieEncryption`, which is both an injected dependency and a singleton. We find that the reflective edge is recorded against the first test class that is run and that this edge happens through the dependency injection library that reaches `CookieEncryption`. However, all subsequent tests do not record the reflective edge, because the singleton is already instantiated so there is no need to re-instantiate it via reflection. Thus, Per-test Analysis safety violations in this version pair are all due to missing test classes which use `CookieEncryption`, but did not have the reflective edge recorded, like our example in Section 3.2.2.

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 leads to selecting many more tests: where RU Analysis has an average precision violation of 26.3%, dynamic Border Analysis has 51.0%, static Border Analysis has 53.6%, and Dynamic Analysis has 50.8%. Although reflection-aware SRTS improves 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 29.5%, which is close to that of RU Analysis. However, Per-test Analysis incurs more safety violations than the other reflection-aware SRTS techniques.

## 5.3 RQ2: Dependency-Level Safety and Precision

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 an SRTS technique. (At the test level, we consider the percentage of *only selected tests*.) Table 2—the *Dep Diff (%)* column—shows the dependency-level safety violations of RU Analysis and the purely static reflection-aware RTS techniques (the Border Analysis variants). Once again, we do not show four-method Border Analysis and minimal Border Analysis as their results are similar to dynamic Border Analysis. Under the Dep Diff (%) column, $X\text{-}RU$, $X\text{-}B_d$, and $X\text{-}B_s$ show the average percentage of all tests for which Ekstazi finds some dependency that RU Analysis, dynamic Border Analysis, and static Border Analysis do not find, respectively. For example, the 26.2% average violation for $X\text{-}RU$ means that 26.2% of tests have at least one dependency that Ekstazi finds but RU Analysis does not find.

As shown in Table 2, there are only four (out of 24) projects where RU Analysis does 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 99.6% of all tests (in undertow, P18) missing at least one dependency that Ekstazi finds. In contrast with what we reported before [Legunsen et al. 2016], it is clear that reflection-unawareness can *potentially* lead to many more affected tests being missed during RTS, showing how unsafe RU Analysis can be. On the other hand, Table 2 shows that, at the dependency level, dynamic Border Analysis and static Border Analysis are as safe as Ekstazi in our experiments. This result demonstrates how the reflective edges recovered in Border Analysis help by over-approximating the test dependencies, which leads to safe RTS in our experiments (again, note that dynamic Border Analysis may become unsafe at some point if the code and tests change enough to require a new set of border methods).

## 5.4 RQ3: Selection Rates

Figure 4 shows the average percentage of tests selected by the different RTS techniques in our study. For each project, the figure shows bars representing the average percentage of RetestAll tests selected by each technique. Each bar represents an average across all versions in each project. In the figure, we show the selection rates for RU Analysis, dynamic Border Analysis, static Border Analysis, Dynamic Analysis, and Per-test Analysis. We do not show results for four-method Border Analysis and minimal Border Analysis; their results are similar to dynamic Border Analysis.

The results in Figure 4 show that reflection-awareness for SRTS comes at the cost of selecting more tests than RU Analysis, which selects on average 34.1% ± 20.4% ($\mu \pm \sigma$, i.e., average ± standard deviation) of all tests, and RU Analysis is already more imprecise than Ekstazi (Table 2). Overall, on average, four-method Border Analysis selects 52.2% ± 24.6%, minimal Border Analysis selects 55.1% ± 23.1%, dynamic Border Analysis selects 57.1% ± 23.9%, static Border Analysis selects 68.8% ± 20.1%, and Dynamic Analysis selects 67.0% ± 20.1% of all tests. Of the Border Analysis variants, four-method Border Analysis performs the best in terms of selection percentages. On the other hand, static Border Analysis on average selects the most number of tests among all reflection-aware SRTS techniques, showing how an imprecise set of border methods can adversely affect test selection. Finally, Per-test Analysis selects tests at a rate, 36.9% ± 24.7%, that is closest to the selection rate of RU Analysis, but Per-test Analysis can be unsafe due to test-order dependencies.

## 5.5 RQ4: Time Savings from Reflection-aware SRTS

We compute for each project the average percentage of RetestAll time each technique takes across all versions. This percentage for each technique is computed based on the *end-to-end* RTS time—the

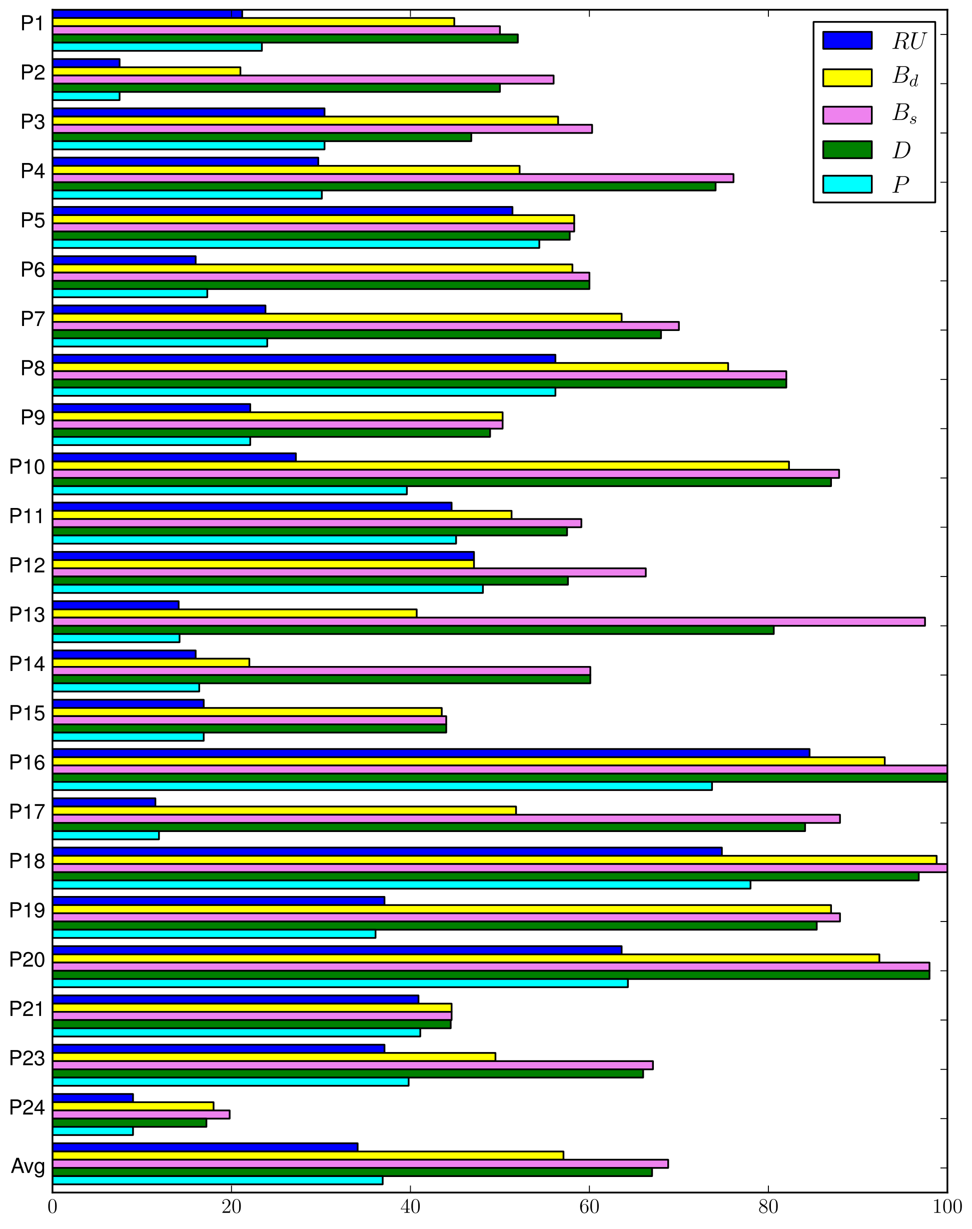

Fig. 4. Average percentage of tests selected

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 version. We compute two variants of this percentage: (1) the percentage of time taken for an "online" mode, which takes into consideration all the components of RTS time, and (2) the percentage of time taken for an "offline" mode, which does not take into consideration the time for updating the test dependencies; the reasoning is that updating test dependencies for the next run can happen offline, after the

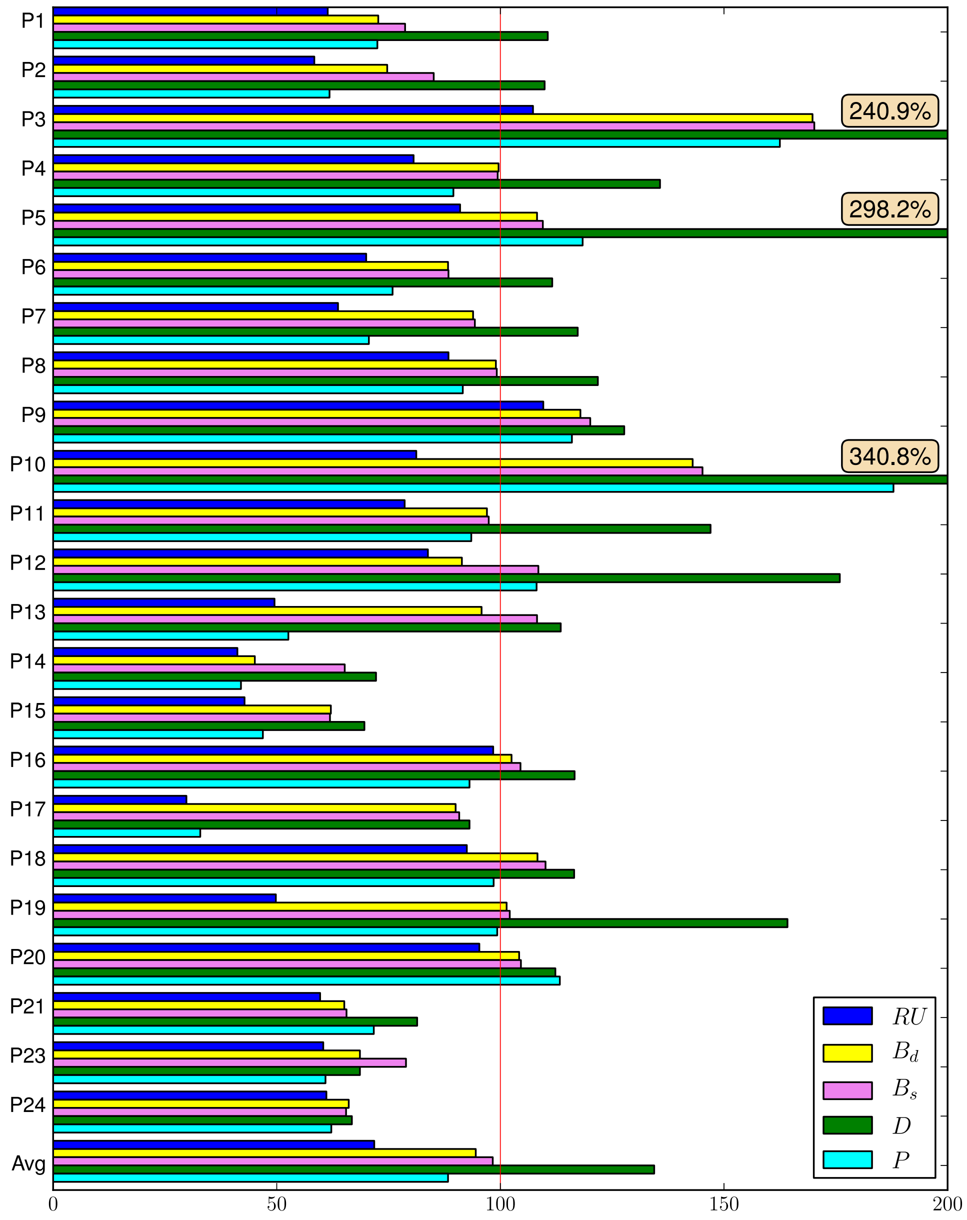

Fig. 5. Average percentage of RetestAll time for RTS in "online mode"

developers see the results of the tests after the current run, so updating test dependencies need not be on the developers' critical path. Figure 5 shows the percentage of time taken by each technique for each project in the "online" mode, while Figure 6 shows the same but for the "offline" mode. Each bar shows for a technique the average percentage of RetestAll time the technique takes across all versions for a project. Once again, we do not show all Border Analysis variants because they have very similar times, so we only show for dynamic Border Analysis and static Border Analysis.

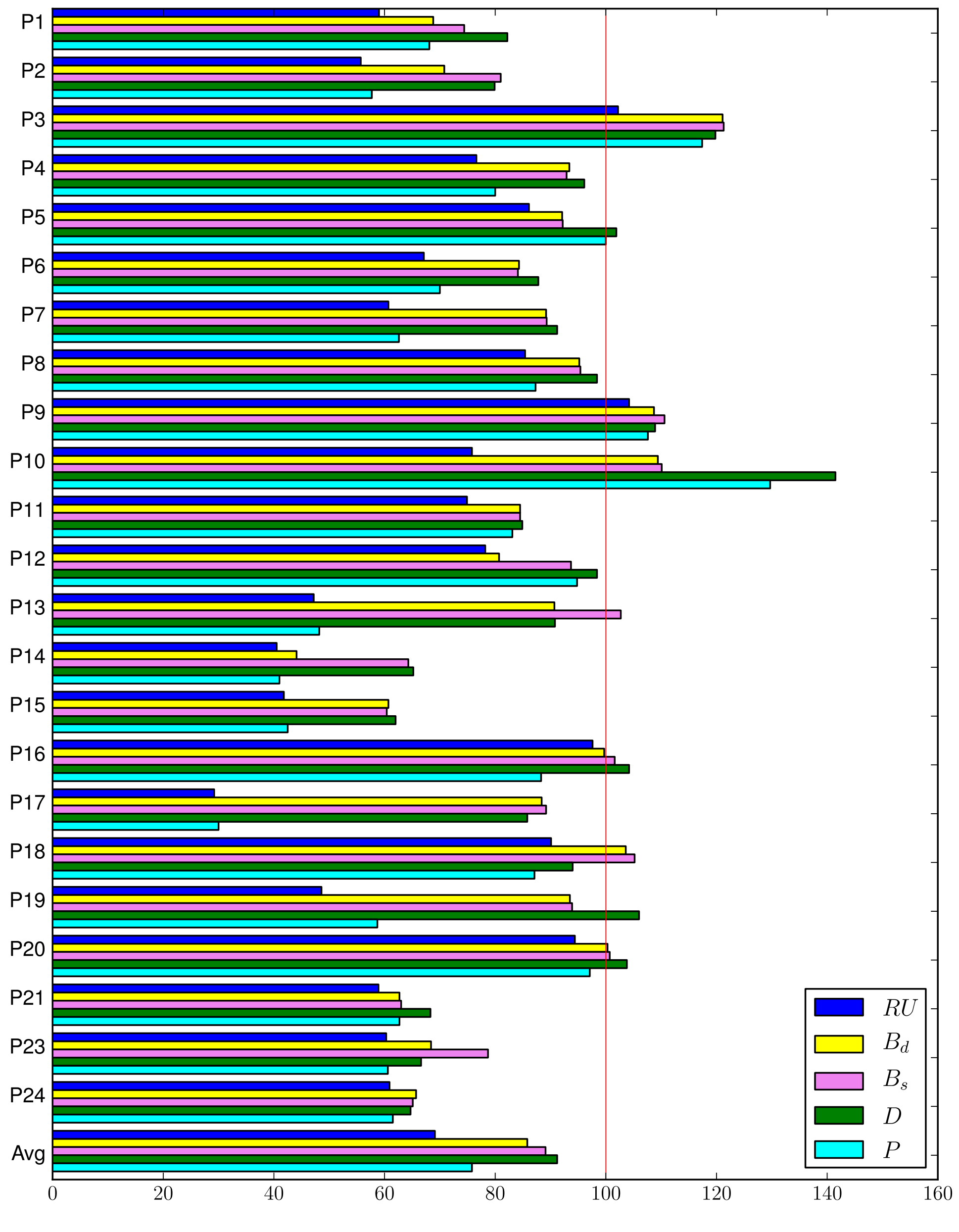

Fig. 6. Average percentage of RetestAll time for RTS in "offline mode"

In the case of Figure 5, for the sake of presentation, we limit the figure to show only the bars up to 200%. For the three bars that go beyond this limit, namely for Dynamic Analysis of projects P3, P5, and P10, we also show the actual percentage above the bars. The red line in Figures 5 and 6 represent the 100% mark to help show how well a technique performs with respect to RetestAll; bars that go over this line indicate that a technique performs worse than RetestAll.

Table 3. Breakdowns of time for RTS techniques. A is analysis time, E is test execution time, G is graph computation time, and C is project compilation time

| ID | *RU* | | | | $B_d$ | | | | *D* | | | | *P* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | E | G | C | A | E | G | C | A | E | G | C | A | E | G | C |
| P1 | 0.9 | 21.9 | 3.3 | 73.8 | 2.6 | 27.7 | 4.0 | 65.7 | 2.6 | 18.5 | 17.1 | 61.8 | 2.9 | 23.7 | 4.7 | 68.7 |
| P2 | 1.2 | 11.3 | 4.1 | 83.3 | 1.2 | 24.7 | 4.2 | 69.9 | 1.1 | 18.7 | 18.2 | 62.0 | 1.5 | 10.9 | 5.8 | 81.8 |
| P3 | 1.1 | 13.6 | 4.7 | 80.6 | 8.1 | 10.5 | 27.1 | 54.3 | 7.2 | 6.7 | 43.3 | 42.8 | 9.3 | 9.0 | 24.7 | 56.9 |
| P4 | 1.2 | 28.3 | 4.8 | 65.6 | 1.7 | 40.0 | 5.7 | 52.6 | 1.9 | 28.5 | 25.0 | 44.5 | 2.1 | 26.4 | 10.0 | 61.6 |
| P5 | 2.1 | 40.7 | 5.3 | 51.9 | 9.0 | 34.5 | 12.6 | 43.8 | 7.0 | 14.6 | 50.7 | 27.7 | 10.7 | 34.0 | 13.1 | 42.1 |
| P6 | 1.4 | 21.4 | 3.8 | 73.5 | 1.4 | 33.1 | 3.6 | 61.9 | 2.1 | 26.3 | 15.6 | 55.9 | 1.7 | 20.4 | 6.8 | 71.2 |
| P7 | 1.6 | 17.2 | 4.6 | 76.6 | 1.5 | 38.8 | 4.3 | 55.4 | 2.0 | 30.1 | 18.0 | 49.9 | 1.9 | 16.2 | 10.6 | 71.3 |
| P8 | 0.7 | 45.6 | 3.3 | 50.4 | 0.6 | 51.7 | 3.4 | 44.3 | 0.6 | 40.9 | 16.7 | 41.8 | 0.7 | 44.4 | 4.6 | 50.2 |
| P9 | 0.9 | 7.1 | 4.6 | 87.4 | 2.3 | 8.9 | 7.3 | 81.4 | 1.3 | 8.2 | 13.5 | 77.0 | 1.3 | 7.1 | 6.9 | 84.7 |
| P10 | 1.8 | 40.8 | 6.4 | 51.0 | 9.1 | 40.1 | 21.8 | 28.9 | 7.5 | 23.4 | 53.3 | 15.8 | 10.0 | 37.3 | 29.1 | 23.6 |
| P11 | 1.8 | 28.3 | 3.9 | 66.0 | 5.0 | 30.6 | 11.9 | 52.5 | 4.9 | 16.9 | 31.0 | 47.2 | 5.1 | 26.7 | 9.0 | 59.3 |
| P12 | 1.9 | 35.2 | 5.4 | 57.5 | 5.2 | 32.3 | 9.0 | 53.5 | 4.3 | 23.8 | 30.6 | 41.3 | 4.9 | 35.2 | 9.5 | 50.4 |
| P13 | 1.0 | 34.6 | 4.8 | 59.5 | 0.8 | 69.8 | 5.5 | 24.0 | 1.1 | 50.9 | 17.4 | 30.6 | 1.4 | 33.2 | 8.8 | 56.6 |
| P14 | 0.9 | 40.5 | 2.6 | 56.0 | 0.9 | 43.8 | 2.8 | 52.5 | 0.8 | 49.8 | 7.8 | 41.6 | 1.0 | 40.1 | 3.5 | 55.4 |
| P15 | 2.4 | 19.6 | 1.8 | 76.3 | 2.1 | 32.1 | 1.4 | 64.4 | 4.1 | 28.3 | 6.2 | 61.4 | 2.4 | 17.7 | 6.7 | 73.3 |
| P16 | 0.1 | 90.8 | 1.0 | 8.1 | 1.1 | 88.0 | 2.7 | 8.1 | 1.4 | 81.9 | 10.6 | 6.0 | 3.3 | 74.5 | 4.5 | 17.7 |
| P17 | 0.6 | 65.3 | 3.0 | 31.2 | 1.0 | 83.5 | 1.6 | 14.0 | 1.2 | 75.6 | 7.1 | 16.0 | 1.6 | 61.9 | 11.3 | 25.1 |
| P18 | 0.6 | 81.5 | 2.9 | 14.9 | 1.1 | 84.1 | 4.4 | 10.4 | 1.6 | 69.1 | 19.3 | 9.9 | 1.4 | 74.1 | 11.9 | 12.6 |
| P19 | 1.3 | 45.4 | 3.8 | 49.6 | 3.5 | 70.9 | 7.9 | 17.6 | 4.6 | 46.7 | 34.5 | 14.2 | 5.9 | 29.6 | 41.8 | 22.7 |
| P20 | 0.4 | 87.9 | 1.0 | 10.8 | 1.8 | 86.3 | 3.7 | 8.2 | 2.3 | 53.4 | 7.3 | 37.0 | 2.1 | 74.9 | 13.8 | 9.2 |
| P21 | 2.4 | 49.0 | 1.4 | 47.1 | 5.8 | 50.4 | 4.0 | 39.7 | 11.6 | 41.6 | 12.2 | 34.5 | 7.8 | 42.9 | 9.8 | 39.4 |
| P22 | 2.3 | 29.2 | 6.8 | 61.7 | 7.1 | 26.3 | 17.7 | 48.9 | 4.6 | 19.7 | 21.2 | 54.5 | 5.1 | 20.9 | 13.6 | 60.4 |
| P23 | 0.4 | 64.7 | 0.3 | 34.6 | 0.5 | 66.9 | 0.3 | 32.3 | 0.6 | 61.4 | 1.9 | 36.1 | 0.6 | 64.7 | 0.3 | 34.5 |
| P24 | 0.3 | 8.5 | 0.3 | 91.0 | 0.8 | 9.8 | 0.5 | 88.9 | 1.5 | 9.2 | 2.2 | 87.1 | 0.9 | 8.4 | 0.8 | 90.0 |
| Avg | **1.2** | **38.7** | **3.5** | **56.6** | **3.1** | **45.2** | **7.0** | **44.7** | **3.3** | **35.2** | **20.0** | **41.5** | **3.6** | **34.8** | **10.9** | **50.8** |

For small projects, we find that SRTS does *not* save as much time compared to big projects relative to RetestAll and that small projects do not benefit as much from RTS as big projects. The savings from SRTS are greater for the big projects. For example, considering only the small projects, the average "offline" percentage of RetestAll time taken by dynamic Border Analysis is 92.5% ± 15.2% (once again, we express variance between projects in the form of $\mu \pm \sigma$), while for big projects the average "offline" percentage is 80.2% ± 23.1%.

Overall, considering all projects, the purely static reflection-aware SRTS techniques (the Border Analysis variants) are on average faster than RetestAll, but not by much, e.g., 85.8% ± 21.7% of RetestAll time for dynamic Border Analysis (in the "offline" mode). Note that these time savings are all for unit tests in the projects, as Ekstazi and STARTS are currently implemented to only handle unit tests run through Maven Surefire. If these tools support RTS for integration tests, which are generally run through Maven Failsafe, we believe the time savings can be even greater. Integration tests generally run longer and our results show that bigger projects with longer-running tests have better time savings (five of our subjects have Failsafe integration tests). Per-test Analysis performs the best out of all reflection-aware techniques, running on average 75.8% ± 24.5% of RetestAll time due to selecting fewer tests. RU Analysis is the fastest overall, taking on average 69.1% ± 19.7% of RetestAll time (compared with 65.5% ± 25.9% for Ekstazi), but is reflection-unaware. Therefore, while RU Analysis remains similar to Ekstazi in terms of end-to-end time, the current approaches for adding reflection-awareness to static RTS makes it less useful in practice.

*5.5.1 Breakdown of Time.* Table 3 shows the breakdown of the time it takes for each RTS technique to run. For space reasons, we only show breakdowns for dynamic Border Analysis as a representative of the Border Analysis variants, as they all have similar breakdown times. The table shows the percentage of the time it takes for each technique in the A phase, which is the analysis time to decide what tests to select, the E phase, which is the execution time for running the selected tests, the G phase, which is the IRG computation time, including collecting and updating dependencies as well as performing the reachability analysis on the IRG, and the C phase, which is the remaining time in the end-to-end total time, mainly used for compilation. The total sum of the times in these phases sum up to the "online" mode time presented in Section 5.5, and the "offline" mode time there is the sum of the times in all phases except the G phase.

In Table 3, the A phase is quite fast in general across all techniques. The G phase is also relatively quick, though it is larger for the hybrid static-dynamic techniques—Dynamic Analysis and Per-test Analysis. The fact that a larger percentage of the time is in the G phase for the hybrid static-dynamic techniques suggests that these techniques have larger graphs to process, or for Per-test Analysis, it suggests that the time to process a separate graph per test is fairly expensive. However, the E percentage for running the selected tests is still larger than the percentages for the other two phases, particularly prominent for the big, longer-running projects. The large percentage of time for executing selected tests suggests that imprecision is still a major factor; these techniques could be sped up if they can (quickly) select fewer tests to run. As discussed more in Section 6, one way to improve the precision of reflection-aware RTS techniques is to consider a hybrid class-and-method static RTS approach, as was recently done in the HyRTS approach for dynamic RTS [Zhang 2018].

One final thing to note is how the C phase actually takes up a large percentage of the end-to-end times. In particular, we see for the small projects that the C phase takes up the *largest* percentage of the time, even greater than the time to execute selected tests. Even for the big projects, the C phase time is comparable to the time for executing tests. Since such a large percentage of the time is dedicated to compilation, improving on compilation may be key to improving the overall end-to-end time developers see for regression testing.

## 5.6 RQ5: Dependency Graph Sizes

As an internal measure of the complexity of the projects and the different SRTS techniques, we compute the number of nodes and edges in the IRGs constructed for each SRTS technique. The size of the graphs can affect the time needed by each technique to determine what tests should be selected after code changes. The IRGs constructed by RU Analysis have, on average across all projects, 5054.1 nodes and 50891.7 edges. RU Analysis graphs contain nodes from the program's classes but not classes from the third-party libraries that the program depends on. Adding reflective edges to this base IRG increases the number of nodes reachable from the tests in the IRG, causing reflection-aware SRTS techniques to explore more edges and reach more classes. Dynamic Analysis adds the largest number of extra nodes and edges, having 227882.8 and 2602026.2, respectively, on average. These large numbers are expected, because Dynamic Analysis tracks the internals of the JSL and therefore finds many classes that are reachable through reflection. Border Analysis IRGs on average range from 45296.4 nodes and 499909.7 edges (four-method Border Analysis) to 45357.4 nodes and 517270.6 edges (static Border Analysis). 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 IRGs select fewer/more tests to run.

IRG sizes for the different Border Analysis variants are essentially the same, a trend that is also correlated with the fact that all the variants have about the same test-selection rates and end-to-end running times. However, all Border Analysis variants still have many more nodes than RU Analysis. This is due to optimizations in the implementation. For RU Analysis, only the classes statically reachable by the tests need to be checked after every change to see what tests should be selected; no other changes can lead to more affected tests in RU Analysis. With Border Analysis, because there can be edges to "*", *all classes*, not just those reachable by the tests, need to be tracked, so the graph size includes many more nodes. The effect of tracking all classes, not just those reachable by the tests, is more pronounced in multi-module Maven projects where graphs for some modules often need to be built and analyzed after code changes for Border Analysis but not RU Analysis.

## 6 DISCUSSION AND FUTURE WORK

**Safety vs. Cost Tradeoffs of Reflection-Aware SRTS:** Overall, our evaluation shows negative results—current techniques to make SRTS safe with respect to reflection come at very high costs that make SRTS less practical. These techniques either are still unsafe or make SRTS too costly for saving time in regression testing. For developers who want SRTS safe with respect to reflection, based on our negative results, future work would need to explore completely new kinds of techniques to make SRTS not just safe but also practical. However, our results show potential tradeoffs of different techniques in terms of safety and cost. If developers are willing to accept some unsafety to lower the cost, they may choose the technique that best suits their purposes.

While RTS research has traditionally focused only on safe techniques, several teams in industry have recently proposed *unsafe* RTS techniques that give acceptable tradeoffs in their settings. For example, Machalica et al. [2019] at Facebook recently proposed an unsafe RTS technique that selects to run only tests that a machine-learning model, trained on historical test failures and historical code changes, predicts as likely to fail for the current code changes. Çelik et al. [2018] also developed an unsafe static RTS technique in their recent collaboration with Samsung because real-time constraints of embedded software being tested prevented using dynamic RTS. Memon et al. [2017] evaluated at Google an unsafe RTS technique that selects tests based only on dependencies close to the code changes and not based on all transitive dependencies. Also at Google, Elbaum et al. [2014] proposed an unsafe RTS technique that selects to rerun only tests that have failed in a predefined failure window, tests that have not been executed in a predefined execution window, or tests that are new. These examples show that developers may be willing to accept unsafe RTS techniques if they offer a good tradeoff between cost and detection of regression bugs. Interesting future work would be to empirically compare the unsafe SRTS techniques from this paper with the other emerging unsafe RTS techniques.

**Other Sources of SRTS Unsafety:** Although we only investigate safety of SRTS due to reflection in this paper, there are many other potential sources of SRTS unsafety, including changes to non-code files (e.g., resource files loaded via file I/O operations), native methods, projects with multiple programming-languages/runtimes, distributed or service-oriented applications, etc. In addition, order-dependent tests [Lam et al. 2015; Zhang et al. 2014] are an issue with RTS in general, not just SRTS. For example, a dynamic RTS technique like Ekstazi would also have issues with order-dependent tests because dependencies may be wrongly associated with tests depending on the order in which the tests are run. Order-dependent tests are a problem for Per-test Analysis presented in Section 5.2. Recent work started addressing some of these sources of unsafety for dynamic RTS [Çelik et al. 2017]; future work could explore handling these safety issues for SRTS.

**Bad Testing Practices:** Some projects have problems in test suites that stem from bad testing practices, such as order-dependent tests or tests that pollute shared state [Gyori et al. 2015]. These problems can hinder RTS in general and SRTS in particular. For example, if a test suite has no order-dependent tests, the safety problems that we observe for Per-test Analysis would not manifest. Our specific experiences with SRTS and general recommendations from others [Greiler et al. 2013; Huo and Clause 2014; Palomba and Zaidman 2017; Spadini et al. 2018; Tufano et al. 2015] about negative effects of these bad testing practices prompt us to strongly recommend developers to avoid such bad practices in the future. If test suites already have order-dependent tests, developers should use existing techniques to detect and/or remove such tests [Bell and Kaiser 2014; Bell et al. 2015; Gambi et al. 2018; Gyori et al. 2015; Lam et al. 2019, 2015; Shi et al. 2019; Zhang et al. 2014].

**Combining Class-Level and Method-Level SRTS:** Zhang [2018] proposed HyRTS, which combines class-level and method-level granularity for dynamic RTS. A similar approach for static RTS may help improve precision of SRTS and alleviate some of the costs of handling reflection for reflection-aware SRTS. Specifically, it would be interesting to explore how the current RU Analysis for SRTS can also be improved by incorporating analysis at a more fine-grained level. Our prior work [Legunsen et al. 2016] found that SRTS based on class dependencies was more effective than SRTS based on method-level dependencies, and hence we evaluated only class-level SRTS in this paper. However, SRTS based on hybrid class-and-method dependencies may help improve the precision of SRTS without making SRTS more unsafe than it already is. Analyzing reflection for methods poses additional challenges due to Java's rich reflection API. We leave investigating this hybrid approach for SRTS as future work.

## 7 THREATS TO VALIDITY

The main threat to internal validity lies in the implementation of our reflection-aware analyses. To reduce this threat, we build our analyses on top of a publicly available static RTS tool, STARTS. Furthermore, we use mature frameworks and libraries to build our reflection-aware analyses, such as the JSA string analysis framework. To further reduce this threat, we plan to contribute our analyses to the STARTS tool so that users can validate, refine, and build on our implementation.

The main threat to external validity lies in the subjects we use in our study. To reduce this threat, we use 1173 versions from 24 open-source projects, including 12 projects with longer test-running times from our prior static RTS study [Legunsen et al. 2016] and 12 additional projects that use reflection heavily. However, our experimental results may still not generalize to other projects. Further reducing this threat requires evaluating our techniques on more and larger projects. Another threat is that some of our subjects may contain flaky tests that can pass and fail on the same version of the code. However, the results we present are consistent over many runs of the experiments.

The main threat to construct validity lies in the metrics we use in evaluating the studied techniques. To reduce this threat, we rely on widely-used RTS metrics, namely test-selection rates and percentage of RetestAll time for testing. For measuring safety and precision, we compare against Ekstazi, a state-of-the-art dynamic RTS tool, as we previously did [Legunsen et al. 2016].

## 8 RELATED WORK

Dynamic RTS techniques have been intensively studied in the literature. Rothermel and Harrold [1993, 1997] proposed one of the first dynamic RTS techniques for C programs based on basic-block-level analysis. Harrold et al. [2001] and Orso et al. [2004] later proposed to handle object-oriented features and adapt 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. [2004] and Zhang et al. [2011] studied method-level dynamic RTS. Gligoric et al. [2015b] 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 costs with its coarse-grained analysis.

Static RTS techniques [Kung et al. 1995; Ren et al. 2003] have been proposed in the past but were not as well studied as dynamic RTS techniques, and their effectiveness and efficiency were largely unexplored before our prior study [Legunsen et al. 2016], where we evaluated the effectiveness of static RTS and compared it against dynamic RTS (i.e., 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; these cases were all due to reflection. In this work, we focus on 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 edges. While we find that making static RTS reflection-aware helps with safety issues, it comes with the cost of very high imprecision and not too much savings in testing time.

Zhu et al. [2019] recently proposed a framework, RTSCheck, that systematically checks for bugs in RTS tools. Specifically, they evaluated RTSCheck on both Ekstazi and STARTS, and they found several bugs. Similar to how we previously [Legunsen et al. 2016] compared different SRTS techniques, including STARTS, with Ekstazi, RTSCheck compared tests selected by different RTS tools with the goal of finding bugs in the tools. Furthermore, RTSCheck supports automatically generating programs and changes to more easily expose differences in tests selected, in addition to using developers' changes in open-source projects. Our safety and precision analysis also involves analyzing differences in tests selected due to developers' changes in open-source projects. However, our overall goal in this work is to reduce the safety problems of static RTS caused by reflection, and we also investigate differences in the dependencies computed by different RTS techniques. By analyzing differences in dependencies, we can discover *potential* safety violations without relying on having actual changes that trigger differences in selection.

Since the time we evaluated and released STARTS, there have been other projects that build on or use STARTS. Thus, there is a growing body of work that could benefit from safety (and precision) improvements of SRTS. The aforementioned work on RTSCheck [Zhu et al. 2019] used STARTS as one of the RTS tools checked. Chen and Zhang [2018] studied the use of STARTS for reducing the costs of mutation testing. We recently used STARTS as a central component of techniques for evolution-aware runtime verification [Legunsen et al. 2015]. We [Legunsen et al. 2019] also used STARTS for change-impact analysis when adapting runtime verification to the context of software evolution by focusing runtime verification and its users on affected parts of code. We compared the class-level RTS in STARTS with the module-level RTS commonly used in large software ecosystems, e.g., at Facebook, Google and Microsoft [Gyori et al. 2018]. We found that STARTS can reduce the test-selection rate in those ecosystems by 10x. Several researchers [Hadzi-Tanovic 2018; Karlsson 2019; Lundsten 2019; Yilmaz 2019] recently published theses where STARTS played a role. In particular, Hadzi-Tanovic [2018] published some of our initial results on reflection-aware static RTS. Karlsson [2019] evaluated the safety, performance, and precision impact on RU Analysis of limiting the transitive closure computation on the IRG to fixed depths, instead of using the full transitive closure. Lundsten [2019] described a preliminary comparison of STARTS with a machine-learning based RTS approach inspired by Machalica et al. [2019]. Yilmaz [2019] compared STARTS with an information-retrieval based RTS approach.

Many researchers studied the impact of reflection in modern programming languages on static analysis [Barros et al. 2015; Bodden et al. 2011; Li et al. 2016a,b, 2014, 2015b; Livshits et al. 2015; Smaragdakis et al. 2015; Thies and Bodden 2012]. However, none of the existing studies investigated the impacts of reflection in the context of RTS. In other words, we are the first to address problems caused by reflection in terms of making static RTS unsafe. The most related previous work [Bodden et al. 2011; Thies and Bodden 2012] studied how to perform static analysis and refactoring in the presence of reflection. Bodden et al. [2011] proposed instrumenting reflection sites to dynamically record when classes invoke reflection and the classes they depend on through reflection. We adopt the same in our Dynamic Analysis and Per-test Analysis, except we apply it to SRTS.

## 9 CONCLUSIONS

We evaluate five reflection-aware RTS techniques—Naïve Analysis, String Analysis, Border Analysis, Dynamic Analysis, and Per-test Analysis. We compare the end-to-end times of these techniques against reflection-unaware static RTS technique, RU Analysis. We also evaluate their relative safety against a dynamic RTS technique, Ekstazi. We obtain mainly negative results that provide valuable knowledge for other researchers looking into safer static RTS. Specifically, our experimental results demonstrate that making RTS safer with respect to reflection is costly and can make static RTS less practical. Border Analysis is the best purely static reflection-aware RTS technique, but its end-to-end time is only on average 85.8% of RetestAll time, compared with 69.1% for RU analysis. Per-test Analysis provides the best time savings among the reflection-aware techniques at 75.8% of RetestAll time, but Per-test Analysis is unsafe in the presence of test-order dependencies. We also perform the first detailed safety/precision analysis at the test dependency level, which shows that RU Analysis could be even more unsafe in practice. Overall, our results show the current challenges to making static RTS reflection-aware.

## ACKNOWLEDGMENTS

We thank David Craig, Alex Gyori, Farah Hariri, and Sasa Misailovic for discussions about this work. We are grateful to the anonymous reviewers for their comments and feedback; we used their words in framing the conclusions. This work was partially supported by NSF grants CCF-1421503, CCF-1566589, CNS-1646305, CNS-1740916, CCF-1763788, and CCF-1763906. We gratefully acknowledge support for research on regression testing from Microsoft and Qualcomm.

## REFERENCES

Apache Software Foundation. 2019a. Apache Camel. (2019). http://camel.apache.org/.

Apache Software Foundation. 2019b. Apache Commons Math. (2019). https://commons.apache.org/proper/commons-math/.

Apache Software Foundation. 2019c. Apache CXF. (2019). https://cxf.apache.org/.

Linda Badri, Mourad Badri, and Daniel St-Yves. 2005. Supporting predictive change impact analysis: A control call graph based technique. In *APSEC*. 167–175.

Paulo Barros, René Just, Suzanne Millstein, Paul Vines, Werner Dietl, Marcelo d'Amorim, and Michael D. Ernst. 2015. Static analysis of implicit control flow: Resolving Java reflection and Android intents. In *ASE*. 669–679.

Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*. 550–561.

Jonathan Bell, Gail Kaiser, Eric Melski, and Mohan Dattatreya. 2015. Efficient dependency detection for safe Java test acceleration. In *ESEC/FSE*. 770–781.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*. 241–250.

Ahmet Çelik, Young Chul Lee, and Milos Gligoric. 2018. Regression test selection for TizenRT. In *FSE Industry Track*. 845–850.

Ahmet Çelik, Marko Vasic, Aleksandar Milicevic, and Milos Gligoric. 2017. Regression test selection across JVM boundaries. In *ESEC/FSE*. 809–820.

Lingchao Chen and Lingming Zhang. 2018. Speeding up mutation testing via regression test selection: An extensive study. In *ICST*. 58–69.

Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. 1994. TestTube: A system for selective regression testing. In *ICSE*. 211–220.

Shigeru Chiba. 2000. Load-time structural reflection in Java. In *ECOOP*. 313–336.

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. 2003. Precise analysis of string expressions. In *SAS*. 1–18.

Nima Dini, Allison Sullivan, Milos Gligoric, and Gregg Rothermel. 2016. The effect of test suite type on regression test selection. In *ISSRE*. 47–58.

Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *FSE*. 235–245.

Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. 2016. CloudBuild: Microsoft's distributed and caching build service. In *ICSE SEIP*. 11–20.

Alessio Gambi, Jonathan Bell, and Andreas Zeller. 2018. Practical test dependency detection. In *ICST*. 1–11.

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015a. Ekstazi: Lightweight test selection. In *ICSE Demo*. 713–716.

Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015b. Practical regression test selection with dynamic file dependencies. In *ISSTA*. 211–222.

Neville Grech, George Kastrinis, and Yannis Smaragdakis. 2018. Efficient reflection string analysis via graph coloring. In *ECOOP*. 1–25.

Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *ICST*. 322–331.

José de Oliveira Guimarães. 1998. Reflection for statically typed languages. In *ECOOP*. 440–461.

Pooja Gupta, Mark Ivey, and John Penix. 2011. Testing at the speed and scale of Google. (Jun 2011). http://goo.gl/2B5cyl.

Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *ISSRE*. 112–122.

Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. 2015. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *ISSTA*. 223–233.

Milica Hadzi-Tanovic. 2018. *Reflection-aware static regression test selection.* Master's thesis. University of Illinois at Urbana-Champaign, USA.

Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software. In *OOPSLA*. 312–326.

Kim Herzig and Nachi Nagappan. 2015. Empirically detecting false test alarms using association rules. In *ICSE*. 39–48.

Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *ISSTA*. 621–631.

Henrik Karlsson. 2019. *Limiting transitive closure for static regression test selection approaches.* Master's thesis. KTH Royal Institute of Technology, Sweden.

Christian Kirkegaard, Anders Moller, and Michael I. Schwartzbach. 2004. Static analysis of XML transformations in Java. *TSE* 30, 3 (2004), 181–192.

David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class firewall, test order, and regression testing of object-oriented programs. *JOOP* 8, 2 (1995), 51–65.

Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST*. 312–322.

Wing Lam, Sai Zhang, and Michael D. Ernst. 2015. *When tests collide: Evaluating and coping with the impact of test dependence.* Technical Report. University of Washington CSE Dept.

Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for static analysis of Java reflection: Literature review and empirical study. In *ICSE*. 507–518.

Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An extensive study of static regression test selection in modern software evolution. In *FSE*. 583–594.

Owolabi Legunsen, Darko Marinov, and Grigore Roşu. 2015. Evolution-aware monitoring-oriented programming. In *ICSE NIER*. 615–618.

Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In *ASE*. 949–954.

Owolabi Legunsen, Yi Zhang, Milica Hadzi-Tanovic, Grigore Roşu, and Darko Marinov. 2019. Techniques for evolution-aware runtime verification. In *ICST*. 300–311.

Hareton K.N. Leung and Lee White. 1990. A study of integration testing and software regression at the integration level. In *ICSM*. 290–301.

Ding Li, Yingjun Lyu, Mian Wan, and William G.J. Halfond. 2015a. String analysis for Java and Android applications. In *ESEC/FSE*. 661–672.

Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016a. Droidra: Taming reflection to support whole-program analysis of Android apps. In *ISSTA*. 318–329.

Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. 2016b. Reflection-aware static analysis of Android apps. In *ASE*. 756–761.

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *ECOOP*. 27–53.

Yue Li, Tian Tan, and Jingling Xue. 2015b. Effective soundness-guided reflection analysis. In *SAS*. 162–180.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundness: A manifesto. *CACM* 58, 2 (2015), 44–46.

Erik Lundsten. 2019. *EALRTS: A predictive regression test selection tool.* Master's thesis. KTH Royal Institute of Technology, Sweden.

Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive test selection. In *ICSE SEIP*. 91–100.

Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *ICSE-SEIP*. 233–242.

Jesper Öqvist, Görel Hedin, and Boris Magnusson. 2016. Extraction-based regression test selection. In *PPPJ*. 1–10.

Oracle. 2018. jdeps. (2018). https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html.

Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling regression testing to large software systems. In *FSE*. 241–251.

OW2 Consortium. 2018. ASM. (2018). http://asm.ow2.org/.

Fabio Palomba and Andy Zaidman. 2017. Does refactoring of test smells induce fixing flaky tests?. In *ICSME*. 1–12.

Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. 2004. Chianti: A tool for change impact analysis of Java programs. In *ACM Sigplan Notices*, Vol. 39. 432–448.

Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, Ophelia Chesley, and Julian Dolby. 2003. *Chianti: A prototype change impact analysis tool for Java.* Technical Report DCS-TR-533. Rutgers University CS Dept.

Gregg Rothermel and Mary Jean Harrold. 1993. A safe, efficient algorithm for regression test selection. In *ICSM*. 358–367.

Gregg Rothermel and Mary Jean Harrold. 1997. A safe, efficient regression test selection technique. *TOSEM* 6, 2 (1997), 173–210.

August Shi, Wing Lam, Reed Oei, Tao Xie, and Darko Marinov. 2019. iFixFlakies: A framework for automatically fixing order-dependent flaky tests. In *ESEC/FSE*. 545–555.

August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *ESEC/FSE*. 237–247.

Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *APLAS*. 485–503.

Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *ICSME*. 1–12.

Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively prioritizing tests in development environment. In *ISSTA*. 97–106.

STARTS Team. 2018. STARTS webpage. (2018). https://github.com/TestingResearchIllinois/starts.

Andreas Thies and Eric Bodden. 2012. RefaFlex: Safer refactorings for reflective Java programs. In *ISSTA*. 1–11.

Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *ICSE*. 403–414.

Kaiyuan Wang, Chenguang Zhu, Ahmet Çelik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *ICSE*. 233–244.

Ugur Yilmaz. 2019. *A method for selecting regression test cases based on software changes and software faults.* Master's thesis. Hacettepe University, Turkey.

Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *STVR* 22, 2 (2012), 67–120.

Nathan York. 2011. Tools for continuous integration at Google scale. (Jan 2011). https://goo.gl/Gqj7uL.

Lingming Zhang. 2018. Hybrid regression test selection. In *ICSE*. 199–209.

Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *ICSM*. 23–32.

Sai Zhang, Darioush Jalali, Jochen Wuttke, Kivanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *ISSTA*. 385–396.

Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A framework for checking regression test selection tools. In *ICSE*. 430–441.