An Extensive Study of Static Regression Test Selection in Modern Software Evolution

Owolabi Legunsen[1], Farah Hariri[1], August Shi[1], Yafeng Lu[2],
Lingming Zhang[2], and Darko Marinov[1]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
{legunse2,hariri2,awshi2,marinov}@illinois.edu
[2]Department of Computer Science, The University of Texas at Dallas
{yxl131230,lingming.zhang}@utdallas.edu

ABSTRACT

Regression test selection (RTS) aims to reduce regression testing time by only re-running the tests affected by code changes. Prior research on RTS can be broadly split into *dynamic* and *static* techniques. A recently developed dynamic RTS technique called Ekstazi is gaining some adoption in practice, and its evaluation shows that selecting tests at a coarser, class-level granularity provides better results than selecting tests at a finer, method-level granularity. As *dynamic* RTS is gaining adoption, it is timely to also evaluate *static* RTS techniques, some of which were proposed over three decades ago but not extensively evaluated on modern software projects.

This paper presents the first extensive study that evaluates the performance benefits of static RTS techniques and their safety; a technique is *safe* if it selects to run all tests that may be affected by code changes. We implemented two static RTS techniques, one class-level and one method-level, and compare several variants of these techniques. We also compare these static RTS techniques against Ekstazi, a state-of-the-art, class-level, dynamic RTS technique. The experimental results on 985 revisions of 22 open-source projects show that the class-level static RTS technique is comparable to Ekstazi, with similar performance benefits, but at the risk of being unsafe sometimes. In contrast, the method-level static RTS technique performs rather poorly.

CCS Concepts

•**Software and its engineering** → **Software evolution;** *Automated static analysis;* •**Software defect analysis** → **Software testing and debugging;**

Keywords

regression test selection, static analysis, class firewall

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950361

1. INTRODUCTION

Modern software projects evolve rapidly as developers add new features, fix bugs, or perform refactorings. To ensure that project evolution does not break existing functionality, developers commonly perform regression testing. However, frequent re-running of full regression test suites can be extremely time consuming. Some test suites require weeks to run [32], but waiting even a few minutes for test results can be detrimental to developers' workflow. In addition to reducing developers' productivity, slow regression testing can consume a lot of computing resources. For example, engineers at Google have observed a quadratic increase in their total test-running times [10, 15, 16], showing that regression testing is challenging, even for a company with a lot of computing resources. As a result, a large body of research has been dedicated to reducing the costs of regression testing, using approaches such as regression test selection [13, 20, 27, 28, 31, 41, 45], regression test-suite reduction [18, 34, 35, 47, 48], regression test-case prioritization [9, 19, 32, 43, 44], and test parallelization [6]. Yoo and Harman provide a thorough survey of regression testing approaches [42].

Regression test selection (RTS) is the most widely used approach to speeding up regression testing [10]. RTS aims to reduce regression testing efforts by only re-running the tests affected by code changes. An RTS technique is *safe* if it selects all tests whose behavior may be affected by code changes; not running any of those tests may cause developers to miss regressions. Prior research on RTS can be broadly split into *dynamic* and *static* techniques.

A typical dynamic RTS technique requires two types of information: (1) changes between two code revisions, and (2) test dependencies dynamically computed while running the tests on the old code revision. Given these inputs, the technique analyzes how the code changes interact with the dependencies to determine a subset of tests that may reach (and thus get affected by) the code changes. Dynamic and safe RTS has been drawing attention in the literature since at least 1993 [30, 42], with some newer techniques such as DejaVOO [27], FaultTracer [45], and Ekstazi [13]. Different dynamic RTS techniques differ in precision[1] and analysis overhead. Techniques that collect finer-granularity dependencies may be more precise, selecting fewer tests to be

[1]Safe techniques always select all tests affected by code changes but could also select some non-affected tests.

run, but can incur higher analysis overhead; in contrast, techniques that collect coarser-granularity test dependencies may be less precise but can have lower analysis overhead.

The state-of-the-art dynamic RTS technique, Ekstazi [11–13], tracks changes and dependencies at the granularity level of files; for Java code, these files include bytecode classes. Ekstazi computes (1) classes which changed between revisions and (2) classes that each test class required while running on the old code revision. Ekstazi selects to run on the new code revision only tests that depend on at least one of the changed classes. Prior experiments with Ekstazi showed that using coarse-granularity test dependencies (at the class level) can substantially save the end-to-end testing time (that includes time to analyze changes, run selected tests, and update dependencies) [13]. Due to this, several open-source projects (e.g., Apache Camel [1], Apache Commons Math [2], and Apache CXF [3]) have already incorporated Ekstazi into their build systems [13].

Despite the recent advances in dynamic RTS, its reliance on dependencies collected dynamically could limit its application in practice, making it important and timely to reconsider static RTS. First, when performing RTS, dynamic test dependencies for the old revision may not always be available, e.g., on the first application of RTS to the project (the project may have earlier revisions). Second, dynamic test dependencies for large projects may be time-consuming to collect. Third, for real-time systems, dynamic RTS may not be applicable, because code instrumentation for obtaining dependencies may cause timeouts or interrupt normal test run. Finally, for programs with non-determinism (e.g., due to randomness or concurrency), dependencies collected dynamically may not cover all possible traces, leading to dynamic RTS being unsafe.

In contrast to dynamic RTS, which collects test dependencies dynamically, static RTS [5, 8, 22, 33] uses static program analysis to infer, ideally, an over-approximation of the test dependencies to enable safe test selection. However, although static RTS techniques for object-oriented languages have been proposed over three decades ago [8, 22], to our knowledge, these techniques have not been studied extensively on modern, real-world projects. In particular, it is not clear *a priori* which granularity level would be better for *static* RTS, although (i) the most recent work [13] on *dynamic* RTS shows that class level provides better results than method level [45] and (ii) due to the growing and relatively larger scale of modern software systems, other recent state-of-the-art dynamic RTS techniques [12, 13, 28, 45] use coarser granularity (e.g., classes and methods) rather than finer granularity (e.g., statements or CFG edges) [20, 30, 31], which are more expensive to collect.

To investigate the safety, precision, and overhead of static RTS, we implemented one class-level static RTS technique and one method-level static RTS technique. The class-level static RTS technique (*ClassSRTS*) is our implementation of the previously proposed *class firewall* [22] technique; it finds class-level dependencies by reasoning about inheritance and reference relationships in a class dependency graph. ClassSRTS selects to re-run any test class that transitively depends on any changed class in the dependency graph. The method-level static RTS technique (*MethSRTS*) utilizes call-graph analysis [5, 17, 39]; it constructs a call graph with all test *methods* as entry points and selects to re-run test *classes* that can transitively reach any changed class through a traversal of the call graph. Our ClassSRTS and MethSRTS implementations are based on the ASM bytecode manipulation and analysis framework [4] and the T.J. Watson Libraries for Analysis (WALA) [40], respectively.

We evaluated these two static RTS techniques on 985 revisions of 22 open-source Java projects. We considered two variants of ClassSRTS and eight variants of MethSRTS, and we compared them against Ekstazi. The results show that ClassSRTS has comparable performance as Ekstazi, but ClassSRTS is occasionally unsafe. In contrast, MethSRTS performs rather poorly: it does not provide performance benefits and is more frequently unsafe. The latter result was somewhat surprising as one may expect finer-grain analysis at the method level to be safer and more precise (but potentially slower) than the coarser-grain analysis at the class level. In conclusion, we recommend that researchers continue improving static RTS techniques at the coarser granularity, which already shows promising results (at least at the level of classes if not modules or projects).

2. BACKGROUND

We introduce the two static RTS techniques based on program analysis at different granularity levels: ClassSRTS performs class-level analysis [22] (Section 2.1), and MethSRTS performs method-level analysis [5,17,29,33,39] (Section 2.2). Although they perform analyses at different levels, we implemented both techniques to report selected test *classes* to aid comparison. In the rest of the paper, when we refer to a *test*, we mean a test class. Recent surveys on regression testing [42] and change-impact analysis [24,26] provide more details about static and dynamic RTS.

2.1 Class-Level Static RTS (ClassSRTS)

Leung et al. [25] first introduced the notion of *firewall* to assist testers in focusing on code modules that may be affected by program changes. Kung et al. [22] further introduced *class firewall* to account for the characteristics of object-oriented languages, e.g., inheritance. Given a set of changed classes, a class firewall computes the set of classes that may be affected by the changes, conceptually building a "firewall" around the changed classes. The original class firewall technique was proposed for the object-relation graph in C++ [22], and Orso et al. [27] generalized it to the *intertype relation graph* (IRG) to additionally consider interfaces in Java. Subsequently, we use *types* to denote classes and interfaces. To the best of our knowledge, using the IRG and the class firewall is the only proposed technique to perform class-level static RTS in Java.

An IRG represents the use and inheritance relations between types in a program, as defined by Orso et al. [27]:

Definition 2.1 (intertype relation graph). *An intertype relation graph, IRG, of a given program is a triple* $\langle N, E_I, E_U \rangle$ *where:*

- N *is the set of nodes representing all types in the program;*
- $E_I \subseteq N \times N$ *is the set of inheritance edges; there exists an edge* $\langle n_1, n_2 \rangle \in E_I$ *if type* n_1 *inherits from* n_2*, and a class implementing an interface is in the inheritance relation;*
- $E_U \subseteq N \times N$ *is the set of use edges; there exists an edge* $\langle n_1, n_2 \rangle \in E_U$ *if type* n_1 *directly references* n_2*, and aggregations and associations are in the use relations.*

Based on this definition of IRG, the class firewall is defined as the types that can (transitively) reach some changed type through use or inheritance edges:

Definition 2.2 (Class Firewall). *The class firewall corresponding to a given set of changed types $\tau \subseteq N$ is computed over the IRG $\langle N, E_I, E_U \rangle$ using the transitive closure of the dependence relation $D = (E_I \cup E_U)^{-1}$; $firewall(\tau) = \tau \circ D^*$, where $^{-1}$ denotes the inverse relation, * denotes the reflexive and transitive closure, and $\circ$ is the relational product.*

ClassSRTS takes as input the two program revisions and the regression test suite T that consists of the tests for the new revision. The output is the subset of tests $T_s \subseteq T$ that may be affected by the changes. ClassSRTS first builds an IRG, computes the changed types between two program revisions, and adds the transitive closure of each changed type to the class firewall. Finally, ClassSRTS returns all tests in the class firewall as the selected test set T_s. Note that ClassSRTS need not include supertypes of the changed types (but must include all subtypes) in the transitive closure because a test cannot be affected statically by the changes even if the test reaches supertype(s) of the changed types unless the test also reaches a changed type or (one of) its subtypes.

2.2 Method-Level Static RTS (MethSRTS)

A program call graph (CG) represents invocation relationships among program methods [39]. Intuitively, starting from each root method (e.g., the `main` method), the call-graph construction finds all methods that can be (transitively) invoked from the root method. A call graph is defined as follows:

Definition 2.3 (Call Graph). *A call graph CG of a given program is a pair $\langle N, E \rangle$, where:*

- *N is the set of all methods in the program under analysis;*
- *$E \subseteq N \times N$ are the method invocation edges.*

MethSRTS takes as input the two program revisions and the regression test suite. The output is T_s, the subset of tests that may be affected by the changes. A test is affected if any of its test methods is affected. (Our experiments show that MethSRTS is rather slow, not because it selects to run test classes instead of test methods, but because the analysis itself is slow.) Further, changes are computed at the *class* level rather than the *method* level because recent work [13] demonstrated that class-level changes do not require complex modeling of changes due to dynamic dispatch (e.g., using *lookup* changes [29, 33]), and can be extremely fast to compute. MethSRTS first builds a call graph for the old revision using as root methods all public methods (e.g., including the "`@Before`" and "`@After`" methods) in the test suite. Then, MethSRTS iterates over each test to check if any of its methods may be affected by the changed types τ, i.e., if it can (transitively) reach methods in τ. Finally, MethSRTS returns the selected set of tests.

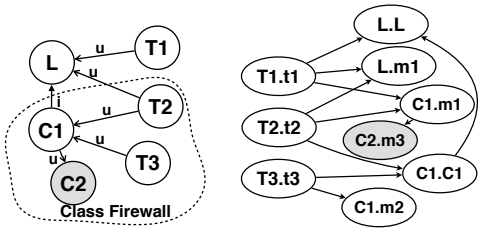
2.3 Example

Although MethSRTS performs analysis at the method level, a finer level of granularity than ClassSRTS, it does not necessarily deliver more precise RTS. To illustrate both techniques, consider the example in Figure 1. The class `L` is in a library, while classes `C1` and `C2` are in the code under test. `T1`, `T2`, and `T3` are three tests. Suppose `C2` is modified (marked in gray). A typical dynamic RTS technique (e.g., Ekstazi), will select to re-run only `T2`, the only test that executes the modified `C2`. We discuss the RTS results for ClassSRTS and MethSRTS.

```
1 //library code
2 class L {
3   void m1() {}
4 }
5
6 //source code
7 class C1 extends L {
8   void m1(){C2.m3()}
9   void m2(){}
10 }
11 class C2 {
12   static void m3(){}
13 }
```

```
1 //test code
2 class T1 {
3   void t1() {
4     L l = new L();
5     l.m1();
6   }
7 }
8 class T2 {
9   void t2() {
10    L l = new C1();
11    l.m1();
12  }
13 class T3 {
14   void t3() {
15    C1 c = new C1();
16    c.m2();
17  }
18 }
```

Figure 1: Example code



(a) IRG (b) Call graph

Figure 2: Example IRG and call graph

ClassSRTS: Figure 2(a) shows the IRG for the example. Edges labeled "`u`" and "`i`" are use and inheritance edges, respectively. The class firewall (enclosed in the dashed area) consists of all classes that can potentially reach the modified class, `C2`. Tests `T2` and `T3` are in the class firewall and thus selected. Note that `T3` is selected due to the imprecision of the class-level analysis: although `T3` uses `C1`, it does not invoke any method of `C2`.

MethSRTS: Figure 2(b) shows the call graph for the example. The method `C2.m3` in the modified class `C2` is marked in gray. From the call graph, tests `T1` and `T2` can potentially reach `C2.m3` and are thus selected. Although MethSRTS can be more precise than ClassSRTS in determining that `T3` cannot invoke `C2.m3`, MethSRTS incurs another imprecision—even advanced call-graph analyses cannot precisely determine receiver object types in all cases. For example, `T1` invokes `m1` with the static receiver type `L`; a naive call-graph analysis (e.g., CHA) will treat all subclasses of `L` as potential runtime receiver types. Therefore, MethSRTS may imprecisely select `T1` that could potentially invoke `C1.m1` because `C1` extends `L`. In contrast, ClassSRTS does not have this issue; any possible runtime object type has to be referenced by the test to instantiate it.

2.4 Analysis Scope

Modern software projects use external libraries extensively which can slow down static analysis in general and MethSRTS in particular. For example, in one revision pair of the project `invokebinder` (p1, Table 5), we found that using RTA call-graph analysis takes 370.8% of RetestAll time when third-party libraries are excluded from the analysis, but it becomes 81304.2% of RetestAll time when libraries

are included—orders of magnitude difference! Therefore, we consider the impacts of excluding third-party libraries for ClassSRTS and MethSRTS as follows.

THEOREM 2.1. *Excluding third-party libraries* cannot *introduce a new type of safety issue for ClassSRTS.*

PROOF. Excluding third-party libraries removes all types in the library from the IRG. If the exclusion induced a safety issue, there must be a path from a test `T` to a changed type `C` that contains a library type `L`; otherwise, the exclusion will not impact the selection results. This implies a path from `T` to `L` and a path from from `L` to `C`. However, third-party libraries are built before the code under test, and therefore `L` cannot statically inherit from or use `C`[2]. Therefore, the path from `L` to `C` does not exist. Contradiction. □

THEOREM 2.2. *Excluding third-party libraries* can *introduce a new type of safety issue for MethSRTS.*

PROOF. As shown in Figure 2, by including library code MethSRTS can select the truly affected test `T2`. However, if the library is excluded, Line 11 in Figure 2(b) will be ignored by the call-graph analysis because the static receiver type is `L`. Then, `T2` only reaches `C1.C1()` that does not invoke any methods from the code under test. Thus, `T2` does not reach any method in the modified class `C2`, and would not be selected. Therefore, excluding third-party libraries can introduce a new type of safety issue for MethSRTS. □

As a result, we exclude libraries for ClassSRTS (making the analysis run faster while selecting the same set of tests), but we evaluate MethSRTS both with and without library exclusion to investigate the cost/safety tradeoff.

3. IMPLEMENTATION

In this section, we describe the implementation details of our ClassSRTS and MethSRTS techniques. Details of Ekstazi, the dynamic RTS tool used in our evaluation, can be found elsewhere [12, 13].

Change computation: Finding syntactically changed source files can be done easily using the `diff` utility or version-control systems, but syntactic changes (e.g., simple reformatting) may not translate to bytecode changes [13]. Therefore, we compute changes at the bytecode level, leveraging the comparison utility from Ekstazi. More specifically, given two program revisions, the comparison first detects the bytecode files that differ between the revisions, and then invokes the Ekstazi API to compute *smart checksums* [13] (by removing debug-related information) of those bytecode files to further filter out the files where only debug-related information changed. Using the Ekstazi change computation also enables a fairer evaluation and comparison of tools.

Graph construction: For ClassSRTS, we used the ASM bytecode manipulation framework (version 5.0) [4] to construct the IRG. Our tool uses ASM to parse the bytecode of each (changed) classfile, traversing all the fields, methods, signatures, and annotations to collect all types that are referenced/used by the type in the classfile. It also collects all types that the type in the classfile extends/implements. Importantly, it incrementally updates the IRG computed from a prior revision by analyzing only the classfiles that changed.

For MethSRTS, we used the call-graph analyses from the IBM WALA framework [40]. We evaluated four widely used call-graph anlayses: CHA (Class Hierarchy Analysis), RTA (Rapid Type Analysis), 0-CFA (Control-Flow Analysis), and 0-1-CFA, in the ascending order of precision [17, 39]. The analyses effectively differ in how they approximate the runtime types of receiver objects, e.g., CHA does not approximate the runtime types at all, while 0-CFA uses one set of types to approximate the runtime types. In general, a more precise call-graph analysis may incur a higher overhead. Furthermore, WALA also allows excluding the library code from the analysis to speed it up. Therefore, we studied these four analyses both with and without library exclusion to investigate the cost/safety trade-offs for MethSRTS. We used *0-CFA with library exclusion* as the default MethSRTS variant because (1) it is recommended by the WALA tutorial [40] for general call-graph analysis applications, and (2) our results show it to perform well among all the eight variants for the specific application of call-graph analysis to RTS. Both tools construct the appropriate graph (IRG or call graph) on the *old* program revision and serialize the constructed graph to the disk for the new revision.

Graph traversal: Given an appropriate graph (IRG or call graph), and a set of changed nodes, each tool needs to find the tests that can reach the changed nodes. Our tools always traverse the graph representing the old revision. For ClassSRTS, we evaluate two modes, offline and online, that traverse the (old) graph at different points. The *offline* mode computes transitive closure of the entire graph in advance (before the new revision and the changes are known), and produces a mapping from test to dependencies; the time to compute the closure *is not* counted in the end-to-end time. The selection then simply checks what test has some changes among its dependencies. The *online* mode computes only nodes transitively reachable from the changes once it knows what those changes are; the time to compute reachability *is* counted in the end-to-end time. Both modes also incrementally update the old graph to produce a new graph for the next revision; the time to perform the update *is not* counted for offline, while it *is* counted for online. For MethSRTS, all eight variants use the online mode. We did not try the offline mode for MethSRTS because (1) MethSRTS performs poorly in terms of safety and precision and is not worth further cost analysis, and (2) MethSRTS selects many more tests than Ekstazi/ClassSRTS so its offline mode can be predicted to be inferior to both others. To implement graph traversals, we use JGraphT [21].

4. EVALUATION AND RESULTS

We present our experimental setup and the results of evaluating static RTS techniques in terms of number of selected tests, time overhead, precision, and safety. We evaluate two variants of ClassSRTS and eight variants of MethSRTS, and we compare them against two baselines: *RetestAll* (which just runs all tests) and Ekstazi. Finally, we present some examples of the safety and precision issues of static RTS.

4.1 Experimental Setup

To evaluate static RTS, we use 22 open-source projects, listed in Table 1. We chose these projects among single-

[2]Note that even when there are callbacks from the library code, the static reference to the receiver type of the callback is usually referenced by some class from the code under test to pass to the library code; also note that the safety issues of static RTS caused by reflection (shown in Section 4.3) exist even with library code analysis.

Table 1: Projects used in study

ID	Project Name	SHA	kLOC	Revs	Tests	T[s]
p1	invokebinder	8611721	2.0	66	2.2	1.7
p2	logback-encoder	4fe0f4a	3.2	43	18.7	3.4
p3	compile-testing	8d5229e	3.0	30	7.6	3.7
p4	commons-cli	3ba638a	5.9	50	23.0	3.8
p5	commons-dbutils	1429538	5.4	33	23.2	4.1
p6	commons-fileupload	1460343	4.3	54	12.0	4.8
p7	commons-validator	bcb1ec4	11.9	19	61.0	4.8
p8	asterisk-java	08dda72	34.5	59	38.1	6.1
p9	commons-codec	50a1d17	17.0	63	47.5	6.5
p10	commons-compress	ec07514	32.5	12	89.4	9.4
p11	commons-email	1607174	6.5	23	17.0	12.3
p12	commons-collections	1543740	54.3	66	149.6	19.9
p13	commons-lang	bcb33ec	69.0	61	133.8	21.6
p14	commons-imaging	b1fdec9	37.1	87	58.9	28.9
p15	commons-dbcp	1587107	18.7	31	27.2	68.9
p16	b.HikariCP	19e0c5d	9.4	49	21.0	80.2
p17	commons-io	1686461	27.7	49	93.9	91.4
p18	addthis.stream-lib	4dc3705	8.3	5	24.0	104.8
p19	commons-math	79c4719	185.4	57	450.2	109.3
p20	OpenTripPlanner	aa21c92	79.3	20	135.8	277.9
p21	commons-pool2	1622091	12.8	51	19.5	294.6
p22	jankotek.mapdb	eac22b7	67.9	57	144.2	515.9
	Average		32.3	44.8	80.2	75.8

module Maven projects with JUnit 4 tests from (1) the original Ekstazi paper [13], (2) one of our previous studies [23], and (3) popular GitHub Java projects with longer-running tests (which we deliberately chose because they are more likely to benefit from RTS). Table 1 shows basic statistics for the projects: SHA is the initial revision of the project on which our experiments started, kLOC is the number of thousands of lines of code in the project, Revs is the number of revision pairs used in our evaluation, Tests is the number of tests in the project, and T[s] is the time in seconds to run all the tests in each project. kLOC, Tests, and T[s] are averages across all revisions that we used.

We used Ekstazi off-the-shelf to compare with static RTS techniques. To select the revisions, we followed the methodology from the original Ekstazi paper [13]. For each project, we started with the 100 revisions immediately preceding SHA and then chose the subset of those 100 revisions (1) that compiled, (2) for which `mvn test` ran successfully, and (3) for which Ekstazi ran successfully with `mvn ekstazi:ekstazi`. Starting from the oldest revision in our list, we ran each RTS technique on the successive pairs of revisions, simulating what an end user would have experienced when using an RTS tool. We measure the number of tests selected by each technique, the time taken to run the selected tests, and the time taken by RTS techniques to perform the selection. The end-to-end time includes the time to collect dependencies, analyze what tests to run, and to actually run the selected tests for the online variants of Ekstazi and the static RTS techniques. For all the offline variants, the time to collect dependencies is not included. We ran all experiments on a 3.40 GHz Intel Xeon E3-1240 V2 machine with 16GB of RAM, running Ubuntu Linux 14.04.4 LTS and Oracle Java 64-Bit Server version 1.8.0_91.

4.2 Research Questions

Our study aims to answer these research questions:

- **RQ1:** How do static RTS techniques compare among themselves and with RetestAll and dynamic RTS in terms of the number of tests selected to run?
- **RQ2:** How do static RTS techniques compare among themselves and with RetestAll and dynamic RTS in terms of runtime?
- **RQ3:** How do static RTS techniques compare with a safe class-level dynamic RTS in terms of precision and safety?
- **RQ4:** How do different variants of the MethSRTS influence the cost/safety trade-offs?

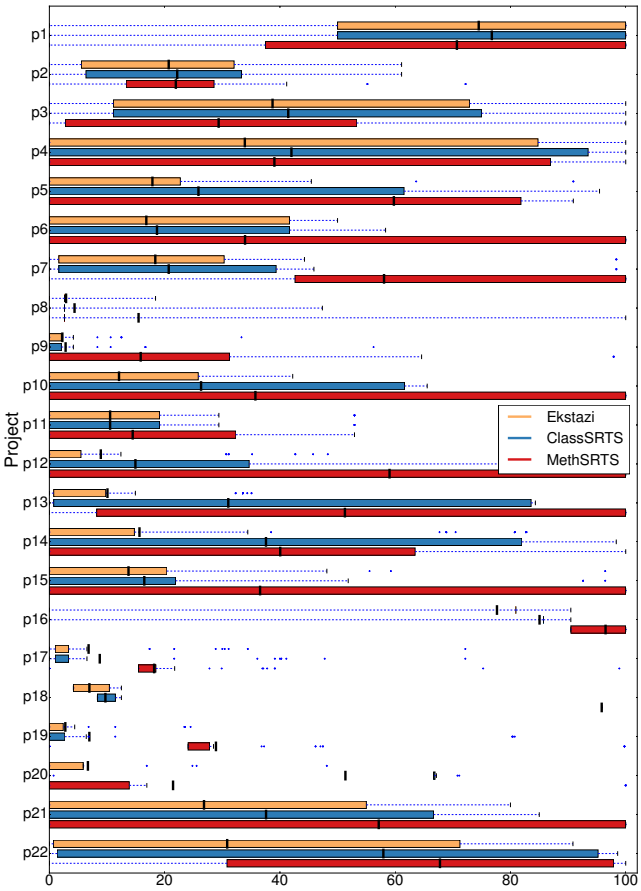


Figure 3: Percentage of tests selected

4.2.1 RQ1: Tests Selected

Figure 3 shows the percentage of tests selected by static and dynamic RTS relative to RetestAll (the black line in the middle of each boxplot is the mean). On average, Ekstazi, ClassSRTS, and MethSRTS select to run 20.6%, 29.4%, and 43.8% of all tests, respectively. As expected, ClassSRTS and MethSRTS both tend to select a higher percentage of tests than Ekstazi. We further discuss exactly how precise and safe ClassSRTS and MethSRTS are with respect to Ekstazi in Section 4.2.3. Surprisingly, the coarser-granularity ClassSRTS technique selects *fewer* tests than the finer-granularity MethSRTS technique. One reason was discussed in Section 2.3, and more concrete cases are analyzed in Section 4.3. Overall, although inferior to the state-of-the-art dynamic RTS in terms of the number of tests selected, static RTS still selects only a fraction of all tests.

4.2.2 RQ2: Time Overhead

While the number of selected tests is an important internal metric in RTS, the time taken for testing is the relevant external metric because a developer using RTS perceives it based on this time. We measure time from the point of view of a developer who commits/pushes some code changes and then waits for test results before proceeding with other tasks. Specifically, we follow the original Ekstazi experiments [13]

Table 2: Summary of end-to-end testing time relative to RetestAll

	Ekstazi (offline)			Ekstazi (online)			ClassSRTS (offline)			ClassSRTS (online)			MethSRTS		
Project	min	max	avg	min	max	avg	min	max	avg	min	max	avg	min	max	avg
p1	92.4	132.7	115.5	88.1	147.3	126.2	72.4	113.8	102.9	75.7	130.9	114.8	215.1	520.8	390.5
p2	82.9	122.3	105.4	85.4	146.2	121.6	64.4	114.0	90.9	65.8	127.3	99.5	190.7	270.9	220.6
p3	74.9	122.6	103.0	76.4	139.0	113.9	65.0	109.6	91.8	64.3	124.5	97.8	130.7	284.4	227.7
p4	92.4	119.4	106.7	97.0	131.2	114.3	83.0	117.7	99.3	83.0	125.0	102.2	167.1	250.3	202.5
p5	81.2	112.8	101.4	87.2	129.1	106.2	73.7	116.7	94.5	80.7	116.9	97.7	156.9	282.9	231.7
p6	73.6	124.0	93.4	70.2	148.3	98.3	61.3	114.6	85.7	62.0	125.3	87.6	123.0	230.1	175.9
p7	71.5	107.0	88.9	75.3	114.4	95.0	62.2	97.9	77.5	61.2	106.2	82.9	131.9	15688.0	9305.2
p8	31.6	109.6	44.4	31.7	117.2	49.0	25.5	104.5	42.9	24.8	330.2	54.1	85.2	458.6	352.8
p9	41.7	93.2	56.4	45.4	96.0	58.2	35.6	80.7	49.6	36.0	91.9	51.3	92.6	258.3	166.9
p10	43.6	91.3	54.1	46.1	108.9	57.9	34.7	105.0	64.2	35.6	110.2	65.3	98.2	253.1	165.8
p11	29.5	99.4	49.9	28.3	103.2	50.1	24.4	105.9	44.8	23.5	105.6	47.1	51.7	155.6	86.5
p12	28.4	101.8	44.0	26.6	109.8	45.5	25.7	102.4	51.3	25.5	252.4	55.8	61.3	4675.5	2670.4
p13	29.4	74.7	44.7	30.6	83.0	47.6	25.2	99.2	55.6	25.9	103.5	57.1	58.7	444.2	316.8
p14	20.4	104.2	46.1	23.1	111.9	48.9	19.0	107.2	54.5	19.3	107.4	55.7	39.3	136.2	81.7
p15	6.5	104.7	26.4	6.7	107.5	27.0	5.6	104.9	26.0	5.6	103.4	26.1	12.4	123.8	57.3
p16	35.1	99.8	94.5	10.9	105.3	97.0	2.8	98.1	68.4	3.0	98.8	66.9	7.8	120.5	80.0
p17	4.1	96.6	19.9	4.1	109.6	21.3	3.4	94.9	22.0	3.4	96.4	22.3	9.7	113.4	46.2
p18	31.6	45.1	35.3	31.9	46.1	36.1	29.9	43.6	34.7	30.4	44.5	34.6	103.6	109.5	107.4
p19	10.9	69.4	17.8	11.3	96.7	20.6	9.6	97.5	19.4	9.7	98.0	19.9	21.9	187.1	112.1
p20	47.8	90.9	67.1	47.9	100.7	71.0	47.0	99.3	85.9	47.1	100.5	86.2	85.8	149.7	112.5
p21	1.1	101.5	52.4	1.1	101.8	52.5	0.9	100.7	59.8	0.8	100.8	59.7	2.2	102.6	68.2
p22	0.9	102.4	41.1	0.9	104.3	42.0	0.8	87.5	52.7	0.7	89.3	53.2	1.6	177.6	117.4
Average	42.3	101.2	**64.0**	42.1	111.7	**68.2**	35.1	100.7	**62.5**	35.6	122.2	**65.3**	84.0	1136.0	**695.3**

Table 3: Safety and precision violations of static RTS compared to Ekstazi

	Safety Violation %								Precision Violation %							
	ClassSRTS				MethSRTS				ClassSRTS				MethSRTS			
Project	revs	min	max	avg	revs	min	max	avg	revs	min	max	avg	revs	min	max	avg
p1	0.0	n/a	n/a	n/a	13.6	33.3	100.0	48.1	4.5	33.3	66.7	50.0	6.1	33.3	66.7	45.8
p2	0.0	n/a	n/a	n/a	76.7	11.1	100.0	40.6	9.3	6.7	75.0	42.3	67.4	15.4	80.0	57.5
p3	0.0	n/a	n/a	n/a	43.3	11.1	100.0	41.9	20.0	20.0	33.3	27.5	0.0	n/a	n/a	n/a
p4	0.0	n/a	n/a	n/a	4.0	16.7	16.7	16.7	16.0	5.9	90.9	61.7	42.0	4.3	80.0	36.4
p5	0.0	n/a	n/a	n/a	30.3	3.2	10.0	6.8	30.3	4.8	100.0	51.1	66.7	33.3	100.0	78.6
p6	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	22.2	14.3	16.7	16.3	29.6	50.0	58.3	57.8
p7	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	36.8	3.6	25.0	16.0	79.0	1.6	96.3	75.6
p8	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	16.9	42.9	100.0	72.9	13.6	81.6	97.4	92.8
p9	0.0	n/a	n/a	n/a	1.6	6.7	6.7	6.7	6.3	25.0	40.7	28.9	44.4	48.4	93.3	87.5
p10	0.0	n/a	n/a	n/a	8.3	50.0	50.0	50.0	41.7	28.3	64.8	53.9	50.0	71.9	100.0	78.9
p11	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	26.1	20.0	71.4	42.3
p12	0.0	n/a	n/a	n/a	6.1	0.8	100.0	25.9	39.4	0.8	98.0	50.0	63.6	7.0	99.3	85.3
p13	0.0	n/a	n/a	n/a	16.4	5.9	15.4	11.3	49.2	6.7	96.3	54.9	83.6	55.6	97.0	80.2
p14	0.0	n/a	n/a	n/a	5.8	2.8	46.1	20.6	39.1	14.0	98.0	62.4	72.4	4.4	97.3	68.3
p15	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	16.1	7.1	40.0	26.2	41.9	3.6	91.7	65.8
p16	2.0	100.0	100.0	100.0	0.0	n/a	n/a	n/a	77.5	5.6	100.0	13.1	93.9	9.5	100.0	20.9
p17	0.0	n/a	n/a	n/a	12.2	2.7	24.2	8.4	14.3	20.0	56.8	33.3	75.5	4.1	95.0	76.0
p18	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	80.0	50.0	50.0	50.0	100.0	87.0	95.7	93.9
p19	1.8	50.0	50.0	50.0	5.3	2.3	7.0	3.9	17.5	35.5	100.0	61.2	79.0	50.2	100.0	92.6
p20	0.0	n/a	n/a	n/a	5.0	55.8	55.8	55.8	75.0	63.0	100.0	91.0	25.0	20.9	94.1	70.6
p21	0.0	n/a	n/a	n/a	0.0	n/a	n/a	n/a	51.0	15.4	92.3	31.9	68.6	15.4	88.9	55.0
p22	0.0	n/a	n/a	n/a	5.3	0.7	13.5	6.5	63.2	7.9	100.0	50.1	96.5	7.9	100.0	65.7
Average	0.2	6.8	6.8	**6.8**	10.6	9.2	29.3	**15.6**	33.0	18.7	70.2	**42.9**	55.7	28.4	86.5	**64.9**

and compare static and dynamic RTS in terms of the *end-to-end time* that includes time to (1) analyze what tests should be run, (2) run those tests, and (3) collect dependencies for future RTS[3]. For any RTS technique to be beneficial, this end-to-end time must be less than RetestAll time.

Table 2 summarizes the end-to-end times for static and dynamic RTS relative to RetestAll. Columns Ekstazi (offline), Ekstazi (online), ClassSRTS (offline), ClassSRTS (online), and MethSRTS represent the time for Ekstazi in offline and online modes, ClassSRTS in offline and online modes, and MethSRTS, respectively. The difference between the offline and online modes of ClassSRTS is explained in Section 3. For Ekstazi, the online mode collects test dependencies during test runs on the new revision, while the offline mode collects them in a separate phase (hence, the results of test runs can be obtained faster, but the overall machine time used is higher) [13]. For each technique, we show the minimum, maximum, and average time relative to RetestAll. The last row shows the average for each column.

[3]Note that the dependency collection time is not counted in the end-to-end time of the offline mode.

The results show that ClassSRTS and Ekstazi both provide benefits over RetestAll (with the average end-to-end time across all projects being 62.5% to 68.2% of the RetestAll time). Comparing ClassSRTS and Ekstazi, we find them to be fairly similar in the respective modes. Based on the average time, ClassSRTS slightly outperforms Ekstazi (62.5% to 64.0% in the fastest, offline modes), but we do note that Ekstazi is implemented as a plugin for the Maven build system [12], while our implementation for ClassSRTS is not yet available as a Maven plugin. Therefore, Ekstazi analysis involves some overhead from Maven. (Both ClassSRTS and Ekstazi actually run tests through Maven.) In brief, we find ClassSRTS and Ekstazi to be equally good based on these experiments. In the future, we plan to perform a deeper comparison, especially to determine how much the ClassSRTS analysis performed to select tests is faster than the dynamic analysis that Ekstazi uses, because Ekstazi tends to select fewer tests to run than ClassSRTS, as seen in Section 4.2.1. For both ClassSRTS and Ekstazi, comparing the offline and online modes shows that they do not differ much, which means that the end-to-end time is, on average, dominated by the time taken to run the selected tests, rather

than by the times to either analyze what tests to run in the case of ClassSRTS or to compute test dependencies in the case of Ekstazi. Finally, we observe that MethSRTS is substantially slower, not only much worse than the other RTS techniques but even worse than RetestAll. Overall, we find that static and dynamic RTS at the class-level have comparable performance, while MethSRTS is effectively useless.

We also point out that RTS techniques provide more benefits for projects with longer-running tests. Counting the number of projects in which some mode on average performs worse than RetestAll (i.e., the `avg` is over 100%), we see that ClassSRTS performs worse in two projects, while Ekstazi performs worse in five projects. However, most of those projects have shorter-running tests, with all five of these projects having tests that run in less than 5 `sec`. For such projects, one can simply use RetestAll and not attempt any RTS. In fact, we could have even removed from our evaluation such projects with short-running tests, but we preferred to keep them to highlight that RTS is not appropriate in all cases. An important point is that RTS techniques should be not only as fast as possible but also as safe as possible—any RTS technique can be simply made faster by not selecting to run some tests, but then it risks missing regressions.

4.2.3 RQ3: Safety and Precision

We compare the safety and precision of ClassSRTS and MethSRTS with respect to Ekstazi, because Ekstazi is a fairly safe and precise dynamic RTS technique [13]. Recall that a safe RTS technique selects to run *all* tests that could change their behavior due to the code changes between two revisions, and a precise RTS technique selects to run *only* the tests that could change their behavior.

Table 3 shows a summary of safety and precision violations, computed as follows. Let E be the set of tests selected by Ekstazi and T be the set of tests selected by another technique on some revision. Safety, respectively precision, violations are computed[4] as $|E \setminus T|/|E \cup T|$, respectively $|T \setminus E|/|E \cup T|$, to measure how much a technique is less safe, respectively precise, than Ekstazi; lower percentages are better. For each of the four combinations (of two types of violations and two static RTS techniques), we tabulate four metrics: `revs` is the percentage of all revisions in which the technique was unsafe/imprecise (i.e., the percentage was not 0), and `min`, `max`, and `avg` are the minimum, maximum, and average, respectively, percentages of tests missed (for safety) or selected extra (for precision). Intuitively, `avg` captures how bad the safety/precision violations are when they happen in a project. We use "n/a" when there were no safety/precision violations for the project. The last row shows the average for each column; we treat "n/a" as 0 when computing the overall averages.

Table 3 shows several interesting results. One surprising result for us is that ClassSRTS was rarely unsafe. Only 2 projects had safety violations, averaging 0.2% across all revisions of all projects evaluated. (Some example safety violations are discussed in Section 4.3.) Moreover, we found that MethSRTS is both less safe and less precise than ClassSRTS, i.e., method-level static RTS is much less effective than class-level static RTS. On average, across all 22 projects, the percentages of revisions in which ClassSRTS incurs safety violations and precision violations are 0.2% and 33.0%, respectively. For MethSRTS, these percentages increase to 10.6% and 55.7%, respectively (10.4 and 22.7 percentage points more unsafe and imprecise, respectively, than ClassSRTS). ClassSRTS is also more effective than MethSRTS in terms of number of projects with safety violations: 2 projects for ClassSRTS vs. 14 projects for MethSRTS. (The number of projects with a precision violation is the same, 21, for both techniques.) Comparing the `min`, `max`, and `avg` values shows the same trend, i.e., when there is a safety or precision violation, MethSRTS is more unsafe or imprecise.

[4]We consider the union of tests selected by both Ekstazi and the technique to avoid division by zero in cases where Ekstazi does not select any test but a static RTS technique selects some tests.

4.2.4 RQ4: MethSRTS Variants

Impacts of call-graph analyses: Table 4 summarizes the results comparing different call-graph analyses for MethSRTS. For each analysis, columns 2–5 present the average percentage of tests selected for all revisions in each project, while columns 6–9/10–13/14–17 present the safety/precision/overhead information. From the table, we observe three things. First, in general, more precise call-graph analyses tend to select fewer tests. For example, on average, CHA selects 48.8% of tests, while 0-1-CFA selects 43.4% of tests. This is because some changed nodes may be reachable from tests in imprecise call graphs but not reachable in precise call graphs. Second, there is no clear trend for safety issues because the precision of call-graph analyses does not directly impact the soundness of the constructed call graphs; instead, the safety issues are usually due to reflection and library exclusion. Third, although 0-1-CFA has a much higher time overhead, it has similar precision with 0-CFA while being the most unsafe of all the four variants. 0-1-CFA is very expensive because it spends a lot of time to approximate possible runtime types of the receiver object for each method invocation [39]. Comparing 0-CFA and 0-1-CFA, 0-CFA has much fewer safety issues and lower overhead; comparing CHA and RTA, RTA has much fewer safety issues and lower overhead. Therefore, 0-CFA (also suggested by WALA developers) and RTA seem to be the most suitable for MethSRTS.

Impacts of library exclusion: To study the impacts of analyzing library code, for each project, we ran WALA with and without library exclusion on only *one* randomly selected revision pair. We do not run for all revisions, because analysis without library exclusion is quite slow. We set a timeout of 3 hours for each revision pair. The results are shown in Table 5, where "DNF" means that the run timed out. (Note that the corresponding times in tables 5 and 4 do not match because Table 5 is for only one revision pair for each project.) 0-CFA and 0-1-CFA timed out for *all* projects without library exclusion, so we do not show them. Additionally, the analyses failed due to memory constraints for three projects (p16, p18, and p20); we do not show these rows. From the results in Table 5, we observe two things. First, without library exclusion, the more expensive and precise RTA analysis does not pay off—RTA selects the same number of tests as the less expensive CHA in all cases, for projects where neither analyses timed out. Second, the analysis overhead relative to RetestAll is much higher without library exclusion than with library exclusion. For example, for `commons-math` (p19), CHA overhead is 2527.9% without library exclusion but 527.3% with library exclusion. This happens because, without library exclusion, a total of 33,770 classes need to be analyzed for this project, making

Table 4: Impacts of different call-graph analyses on MethSRTS

Project	Tests Selected %				Safety Violation %				Precision Violation %				Time %			
	CHA	RTA	0CFA	01CFA	CHA	RTA	0CFA	01CFA	CHA	RTA	0CFA	01CFA	CHA	RTA	0CFA	01CFA
p1	74.8	74.0	70.7	70.7	48.1	48.1	48.1	48.1	45.0	44.4	45.8	45.8	777.8	493.7	390.5	387.2
p2	27.8	21.9	21.9	21.9	40.5	40.6	40.6	40.6	61.0	57.5	57.5	57.5	515.2	280.6	220.6	222.1
p3	39.7	32.7	29.4	29.4	75.0	47.5	41.9	41.9	19.7	30.0	n/a	n/a	571.2	571.5	227.7	237.6
p4	44.7	40.8	39.0	38.7	16.7	16.7	16.7	15.0	39.8	35.8	36.4	36.1	441.8	257.5	202.5	203.3
p5	68.9	68.8	59.8	59.8	n/a	n/a	6.8	6.8	75.0	75.1	78.6	78.6	484.9	346.4	231.7	247.6
p6	35.8	34.0	34.0	34.0	n/a	n/a	n/a	n/a	60.3	57.8	57.8	57.8	263.5	193.3	175.9	176.0
p7	52.4	50.6	58.1	58.1	n/a	n/a	n/a	n/a	72.2	71.2	75.6	75.6	661.0	659.8	9305.2	121213.1
p8	25.4	23.7	15.5	10.3	n/a	n/a	n/a	71.4	94.8	94.5	92.8	93.0	470.5	403.8	352.8	427.5
p9	21.6	21.6	15.8	15.8	n/a	n/a	6.7	5.0	90.0	90.0	87.5	87.5	435.9	231.5	166.9	167.1
p10	35.8	35.8	35.8	35.8	50.0	50.0	50.0	50.0	78.9	78.9	78.9	78.9	378.5	237.0	165.8	169.6
p11	15.3	15.3	15.1	14.6	n/a	n/a	n/a	11.1	42.9	42.9	42.3	42.3	143.2	100.6	86.5	86.3
p12	61.0	61.0	59.0	59.0	50.8	50.8	25.9	25.9	85.4	85.4	85.3	85.3	496.5	395.1	2670.4	416.1
p13	61.2	61.2	51.3	51.1	13.9	13.9	11.3	11.6	82.5	82.5	80.2	79.8	880.1	713.6	316.8	235.6
p14	40.9	40.9	40.0	40.0	20.6	20.6	20.6	20.6	68.7	68.7	68.3	68.3	109.5	88.1	81.7	82.4
p15	36.6	36.6	36.6	36.6	n/a	n/a	n/a	n/a	65.8	65.8	65.8	65.8	69.1	61.1	57.3	59.0
p16	98.1	100.0	96.5	93.0	94.1	n/a	n/a	12.5	24.4	24.3	20.9	23.4	121.8	125.6	80.0	79.4
p17	37.7	37.2	18.2	18.2	n/a	n/a	8.4	8.4	86.1	85.5	76.0	76.0	113.5	85.8	46.2	47.3
p18	95.8	95.8	95.8	95.8	n/a	n/a	n/a	n/a	93.9	93.9	93.9	93.9	134.1	131.0	107.4	106.3
p19	36.0	35.8	28.9	28.9	n/a	n/a	3.9	3.9	94.4	94.4	92.6	92.5	356.3	267.2	112.1	107.7
p20	31.4	31.4	17.5	17.5	n/a	n/a	55.8	58.1	89.2	89.2	70.6	70.6	158.2	194.1	112.5	128.6
p21	61.4	61.4	57.1	57.1	n/a	n/a	n/a	n/a	58.6	58.6	55.0	55.0	73.3	70.7	68.2	67.8
p22	71.7	71.4	67.8	67.8	7.1	6.5	6.5	6.5	66.0	65.8	65.7	65.7	118.1	112.0	117.4	107.1
Average	48.8	47.8	43.8	43.4	18.9	13.4	15.6	19.9	67.9	67.8	64.9	65.0	353.4	273.6	695.3	5680.7

Table 5: Impacts of library exclusion on MethSRTS (only one revision pair, not all pairs, per project)

Project	Tests Selected %				Safety Violation %				Precision Violation %				Time %			
	exclusion		no exclusion		exclusion		no exclusion		exclusion		no exclusion		exclusion		no exclusion	
	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA	CHA	RTA
p1	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	450.0	370.8	5550.0	81304.2
p2	0.0	0.0	100.0	100.0	n/a	n/a	n/a	n/a	n/a	n/a	100.0	100.0	478.1	237.5	5475.0	97087.5
p3	33.3	0.0	66.7	66.7	50.0	100.0	n/a	n/a	n/a	n/a	n/a	n/a	678.1	609.4	5271.9	73028.1
p4	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	4.3	4.3	4.3	4.3	627.0	316.2	6573.0	71027.0
p5	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	81.8	81.8	81.8	81.8	474.5	333.3	4452.9	53649.0
p6	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	50.0	50.0	50.0	50.0	402.1	261.7	3542.6	47459.6
p7	44.3	44.3	44.3	44.3	n/a	n/a	n/a	n/a	96.3	96.3	96.3	96.3	600.0	585.5	4960.0	65352.7
p8	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	97.4	97.4	97.4	97.4	566.1	496.6	3819.1	54051.5
p9	97.8	97.8	97.8	97.8	n/a	n/a	n/a	n/a	88.9	88.9	88.9	88.9	501.9	226.0	4018.3	34651.0
p10	1.2	1.2	1.2	1.2	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	437.1	243.8	3788.6	34364.8
p11	41.2	41.2	41.2	41.2	n/a	n/a	n/a	n/a	42.9	42.9	42.9	42.9	204.6	122.9	1681.7	20593.1
p12	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	64.8	64.8	64.8	64.8	678.4	553.8	3670.8	32296.5
p13	98.5	98.5	98.5	DNF	n/a	n/a	n/a	DNF	48.1	48.1	48.1	DNF	958.1	752.1	6385.5	DNF
p14	13.5	13.5	13.5	13.5	n/a	n/a	n/a	n/a	71.4	71.4	71.4	71.4	80.8	61.5	595.3	7551.1
p15	96.3	96.3	96.3	96.3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	157.4	136.1	507.0	5935.8
p17	97.7	97.7	97.7	97.7	n/a	n/a	n/a	n/a	67.4	67.4	67.4	67.4	181.8	151.7	758.8	6142.2
p19	100.0	100.0	100.0	DNF	n/a	n/a	n/a	DNF	94.2	94.2	94.2	DNF	527.3	422.4	2527.9	DNF
p21	100.0	100.0	100.0	100.0	n/a	n/a	n/a	n/a	44.4	44.4	44.4	44.4	108.1	104.0	181.7	1106.3
p22	58.0	58.0	58.0	58.0	n/a	n/a	n/a	n/a	50.0	50.0	50.0	50.0	140.8	130.3	586.5	4919.1
Average	72.7	71.0	79.8	77.5	2.6	5.3	n/a	n/a	47.5	47.5	52.7	50.6	434.3	321.9	3386.7	36343.1

CHA very time consuming. The analysis time for RTA is even higher without library exclusion. In the same project, the analysis for RTA times out. RTA has such a high overhead because it spends significant time to compute the sets for approximating potential runtime types of the receiver object for every method invocation [39]. Overall, these results demonstrate that call-graph analyses without library exclusion are not practical for static RTS due to the high cost and precision issues. In fact, method-level static RTS unfortunately appears impractical in all configurations.

4.3 Qualitative Analysis

We discuss concrete cases of safety and precision violations where a static RTS technique does not select some test(s) that Ekstazi selects or selects some test(s) that Ekstazi does not select. An unsafe RTS technique is bad in a non-obvious way because it can be deceivingly fast but risk missing regressions; our analysis of safety violations identifies some cases where static RTS techniques were unsafe during our experiments. In contrast, an imprecise RTS technique is bad in an obvious way because the imprecision is reflected in the end-to-end time. These examples illustrate current limitations of static analysis for RTS and can provide insight into how to improve them in the future.

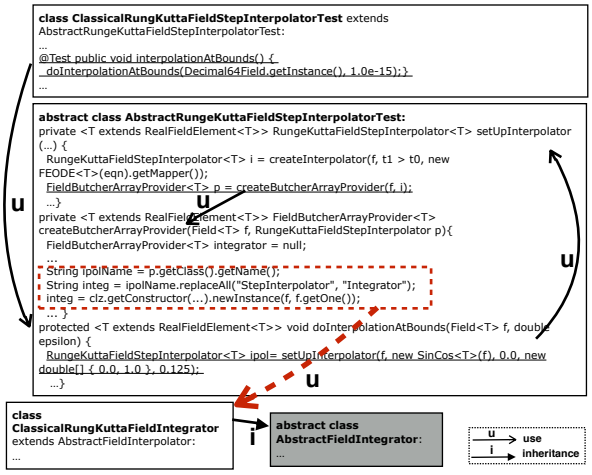


Figure 4: ClassSRTS safety violation due to reflection

Safety Violations of Static RTS.

Safety violations due to reflection: In `commons-math`, between revisions `2773215` and `c246b37`, ClassSRTS and MethSRTS miss to select nine tests that Ekstazi selects. The relevant change is to the abstract class `Ab-`

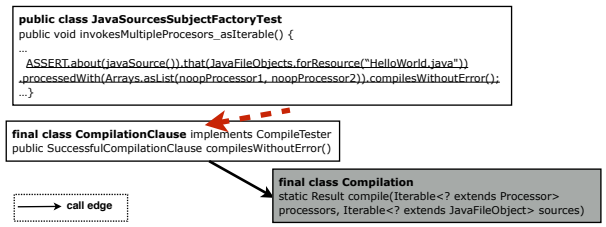


Figure 5: Safety violation due to third-party library exclusion (MethSRTS)

`stractFieldIntegrator`, extended by several `*FieldIntegrator` classes (e.g., `ClassicalRungeKuttaFieldIntegrator`) that are definitely affected by the change. All techniques do select multiple tests, such as `ClassicalRungeKuttaFieldIntegratorTest`. The nine additional tests that only Ekstazi selects are in `*FieldStepInterpolatorTest` classes (e.g., `ClassicalRungeKuttaFieldStepInterpolatorTest`) that extend `AbstractRungeKuttaFieldStepInterpolatorTest`. As shown in Figure 4, `ClassicalRungeKuttaFieldStepInterpolatorTest` invokes the method `doInterpolationAtBounds` in `AbstractRungeKuttaFieldStepInterpolatorTest` that invokes `setUpInterpolator`, which in turn invokes `createButcherArrayProvider` in the same class. The latter method (shown in the red dashed box area) gets `*FieldStepInterpolator` instances, replaces the string `"StepInterpolator"` with `"Integrator"` in the class name, and uses reflection to create the new `*FieldIntegrator` instance. In this case, ClassSRTS and MethSRTS are unsafe because they do not detect the potential use edge (red dashed arrow in the figure) due to reflection, and thus they miss to select the related tests. In contrast, Ekstazi tracks the precise class dependencies dynamically, even in the presence of reflection, and detects that the `*FieldStepInterpolatorTest` instances depend on the corresponding `*FieldIntegrator` instances.

Safety violation due to library exclusion: In `compile-testing`, between revisions `8d5229e` and `40c141b`, both Ekstazi and ClassSRTS select the same two tests, but MethSRTS with library exclusion is unsafe and selects no test. The relevant code change is to the class `Compilation`. As shown in Figure 5, test `JavaSourcesSubjectFactoryTest` can reach method `compile` in the changed class `Compilation` (marked in gray). The underlined statement in `JavaSourcesSubjectFactoryTest` invokes methods on `ASSERT`, a class in the third-party library `truth`. When third-party libraries are excluded from the call-graph analysis, MethSRTS does not analyze the underlined statement and fails to select the test.

Precision Violations of Static RTS.

Imprecision due to class-level analysis: In `asterisk-java`, between revisions `166f293` and `d6bfce1`, Ekstazi selects four tests, while ClassSRTS selects four additional tests due to the imprecision of analyzing at the class level. The setup method of `AsteriskAgentImplTest` calls `new AsteriskServerImpl()`. The method `onManagerEvent` in the class `AsteriskServerImpl` contains checks for the type of `event`, e.g., `if (event instanceof AgentCalledEvent)`. Therefore, ClassSRTS finds a transitive static dependency of `AsteriskAgentImplTest` on `AgentCalledEvent`, which changed between the mentioned revisions. However, the actual run of `AsteriskAgentImplTest` never invokes `onManagerEvent` in `AsteriskServerImpl` (the relevant conditional is never executed), and Ekstazi correctly does not track this dependency.

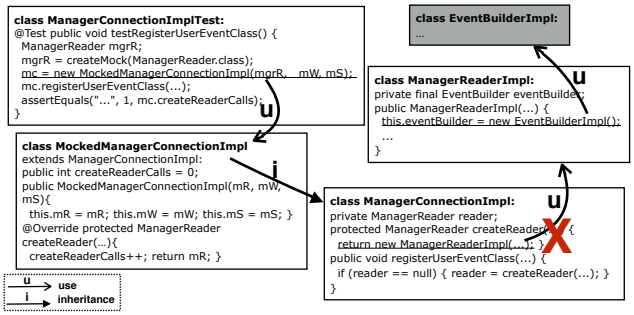


Figure 6: ClassSRTS precision violation due to dynamic dispatch

Imprecision due to dynamic dispatch: For the same `asterisk-java` revisions, Figure 6 shows simplified code for a case where ClassSRTS finds a false transitive dependency of the test `ManagerConnectionImplTest` on the changed class `EventBuilderImpl` (marked in gray) due to dynamic dispatch. `ManagerConnectionImplTest` uses `MockedManagerConnectionImpl` that extends `ManagerConnectionImpl` and overrides `createReader`. `ManagerConnectionImpl` uses class `ManagerReaderImpl` which in turn uses the changed class `EventBuilderImpl`. Therefore, ClassSRTS finds that `ManagerConnectionImplTest` depends on the changed class. However, during the test run, when `ManagerConnectionImplTest` invokes `ManagerConnectionImpl.registerUserEventClass` to check how many times the `createReader` is called, the overriding method `createReader` in `MockedManagerConnectionImpl` is executed instead of `createReader` in the parent class. Therefore, the use edge from `ManagerConnectionImpl.createReader` to `ManagerReaderImpl` is never executed and should not be considered (marked using a red cross), i.e., the edge exists statically but not dynamically as `ManagerConnectionImpl.createReader` is never executed. In brief, ClassSRTS has this imprecision due to dynamic dispatch.

Compared with ClassSRTS, MethSRTS is even less precise in identifying potential targets for dynamic dispatch. For example, between revisions `1460430` and `1475836` of `commons-fileupload`, while Ekstazi and ClassSRTS select 6 and 7 tests, respectively, MethSRTS selects all 12 tests. A simplified call graph for the test `FileItemHeadersTest` is shown in Figure 7. `FileItemHeadersTest` invokes `assertEquals` from the JUnit library, which in turn transitively invokes the method `printString` in the library class `FormatSpecifier`. When resolving the call site in the library method that invokes `toString` on a receiver of the declared type `Object`, even the most advanced 0-1-CFA cannot precisely determine the runtime type of the receiver. Therefore, all classes overriding `Object.toString` are potential targets. In this case, the invocation targets include `DiskFileItem.toString` which transitively invokes a method in the changed class `Streams` (marked in gray). This example, similar to the one in Section 2.3, shows how ClassSRTS is more precise because any possible runtime object type has to be referenced by the test to instantiate it.

4.4 Threats to Validity

Internal: Our static RTS prototypes and scripts for running experiments may contain bugs. (We thank an anonymous reviewer for pointing out one bug!) To mitigate risks, we use well-known libraries, e.g., ASM and WALA. We also wrote unit tests to check basic functionality, and we im-

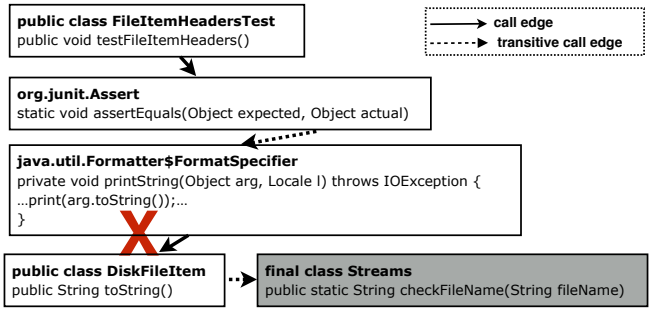


Figure 7: MethSRTS precision violation due to dynamic dispatch

plemented some sanity checks for numbers generated from scripts, e.g., we check that the offline and online variants of both ClassSRTS and Ekstazi select the same tests.

External: The projects in our study are a subset of all software and may not be representative, so our results may not generalize. To address this, we used 985 revisions from 22 open-source projects varying in size, application domain, number of tests, and test-suite running time. However, all our projects are single-module Maven projects. The results could differ for bigger, multi-module Maven projects, but we expect that these results could show RTS to be even better as we generally find RTS to be more effective for projects with longer-running tests.

Construct: We chose Ekstazi as the ground truth against which to evaluate static RTS techniques. Ekstazi is a state-of-the-art, publicly available tool for dynamic RTS, but may still not represent the ground truth for all RTS.

5. RELATED WORK

RTS [13,20,22,27,28,31,33,45] can reduce regression testing efforts and has been extensively studied for more than three decades. Recent studies [7, 14] show that developers often perform *manual* RTS even when their test-suite size is small, but our focus is on *automated* RTS. Most recently, the Ekstazi *dynamic* RTS technique [13] was extensively evaluated and is being adopted in open-source software. To the best of our knowledge, though, there was no prior such evaluation of static RTS techniques for modern software projects.

Dynamic RTS: Rothermel and Harrold [30, 31] investigated RTS for C programs. They dynamically collected coverage on the old revision to perform RTS using a control-flow graph (CFG) analysis. Harrold et al. [20] further extended this work to handle Java features and incomplete programs. Because CFG analysis can be very time consuming for large software, Orso et al. [27] proposed a two-phase analysis, a partitioning phase to filter out non-affected classes from an IRG and a selection phase that performs CFG analysis only on the classes kept by the first phase. They also evaluated a class-level RTS technique (called "HighLevel"), but it did not compute transitive dependencies on the use edges; moreover, it computed dependencies on the supertypes of the changed types and not only on the subtypes.

To improve the efficiency of dynamic RTS, a number of techniques at coarser-granularity levels (e.g., method- or class-level) rather than the finer-granularity CFG level were proposed. Ren et al. [28] and Zhang et al. [45, 46] applied change-impact analysis at the method-level, based on call graphs techniques, to improve RTS. Recently, Gligoric et al. [13] proposed Ekstazi, a dynamic RTS technique at the class/file level. Although Ekstazi performs coarser-level analysis and may select more tests than prior work, it was demonstrated to have a sufficiently lower end-to-end testing time to be adopted by some open-source projects. While these dynamic RTS techniques can be safe, they require dynamic test coverage information which may be absent, costly to collect, or require prohibitive instrumentation (e.g., for non-deterministic or real-time code). Therefore, our work studies static RTS techniques.

Static RTS: Kung et al. [22] first proposed static RTS based on class firewall, i.e., the statically computed set of classes that may be affected by a change. Ryder and Tip [33] proposed a call-graph-based static change-impact analysis technique and evaluated only one call-graph analysis (0-CFA) on 20 revisions of one project [29]. Badri et al. [5] further extended call-graph-based change-impact analysis with a CFG analysis to enable more precise impact analysis, but they did not evaluate it on RTS. Skoglund and Runeson [36, 37] performed a case study of class firewall, but they used dynamic coverage information together with class firewall to perform RTS, whereas we apply the class firewall purely statically. Although the literature contains many static RTS techniques, extensive studies of these techniques are lacking. In particular, prior to our study, evaluations on modern open-source projects were lacking, so it was not clear how static and dynamic RTS techniques compare in terms of safety, precision, and overhead. The static RTS techniques presented in this paper are representative of all prior work on class-firewall-based analyses [22] and call-graph-based analyses [33].

6. CONCLUSIONS

Regression testing is an important but costly software engineering task. To speed up regression testing, researchers have proposed many techniques for regression test selection (RTS). Dynamic RTS is showing promising results, with the Ekstazi tool being adopted in practice. This success of dynamic RTS provides motivation to (re-)evaluate static RTS, because static RTS could be more beneficial than dynamic RTS for systems with long-running tests, non-determinism, or real-time constraints. We evaluate several variants of class- and method-level static RTS. The results show that class-level static RTS is comparable to class-level dynamic RTS Ekstazi, but method-level static RTS is rather poor.

The key message of this paper is that static RTS at the class level shows promising results, and researchers should continue to improve static RTS. The promising results will also hopefully motivate development of robust static RTS tools that can be adopted in practice. The method-level analysis is currently not useful for RTS but may still be useful for other tasks, e.g., debugging regression failures [38].

7. ACKNOWLEDGMENTS

We thank Milos Gligoric for his help with the Ekstazi tool, and Alex Gyori, John Micco, and the anonymous reviewers for feedback on previous drafts of this paper. This research was partially supported by the National Science Foundation Grant Nos. CCF-1409423, CCF-1421503, CCF-1438982, CCF-1439957, and CCF-1566589. Darko Marinov and Lingming Zhang also gratefully acknowledge their Google Faculty Research Awards.

8. REFERENCES

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

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

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

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

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

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

[7] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *ESEC/FSE*, pages 179–190, 2015.

[8] S. A. Bohner and R. Arnold. *Software change impact analysis*. Wiley Publishers, 1996.

[9] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *TSE*, 32(9):733–752, 2006.

[10] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *FSE*, pages 235–245, 2014.

[11] M. Gligoric. *Regression Test Selection: Theory and Practice*. PhD thesis, UIUC CS Dept., July 2015.

[12] M. Gligoric, L. Eloussi, and D. Marinov. Ekstazi: Lightweight test selection. In *ICSE Demo*, pages 713–716, 2015.

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

[14] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *ASE*, pages 361–372, 2014.

[15] Testing at the speed and scale of Google, Jun 2011. http://goo.gl/2B5cyl.

[16] Tools for continuous integration at Google scale, Jan 2011. https://goo.gl/Gqj7uL.

[17] D. Grove and C. Chambers. A framework for call graph construction algorithms. *TOPLAS*, 23(6):685–746, 2001.

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

[19] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *TOSEM*, 24(2):10:1–10:31, 2014.

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

[21] JGraphT. http://jgrapht.org/.

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

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

[24] S. Lehnert. A review of software change impact analysis. Technical report, Ilmenau U. of Tech., 2011.

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

[26] B. Li, X. Sun, H. Leung, and S. Zhang. A survey of code-based change impact analysis techniques. *STVR*, 23(8):613–646, 2013.

[27] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–251, 2004.

[28] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *OOPSLA*, pages 432–448, 2004.

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

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

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

[32] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *ICSM*, pages 179–189, 1999.

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

[34] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing trade-offs in test-suite reduction. In *FSE*, pages 246–256, 2014.

[35] A. Shi, T. Yung, A. Gyori, and D. Marinov. Comparing and combining test-suite reduction and regression test selection. In *ESEC/FSE*, pages 237–247, 2015.

[36] M. Skoglund and P. Runeson. A case study of the class firewall regression test selection technique on a large scale distributed software system. In *ESEM*, pages 74–83, 2005.

[37] M. Skoglund and P. Runeson. Improving class firewall regression test selection by removing the class firewall. *JSEKE*, 17(3):359–378, 2007.

[38] S. H. Tan and A. Roychoudhury. relifix: Automated repair of software regressions. In *ICSE*, pages 471–482, 2015.

[39] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, 2000.

[40] IBM WALA. http://wala.sourceforge.net.

[41] G. Xu and A. Rountev. Regression test selection for aspectj software. In *ICSE*, pages 65–74, 2007.

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

[43] K. Zhai, B. Jiang, and W. K. Chan. Prioritizing test cases for regression testing of location-based services: Metrics, techniques, and case study. *TSC*, 7(1):54–67, 2014.

[44] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and

H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *ICSE*, pages 192–201, 2013.

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

[46] L. Zhang, M. Kim, and S. Khurshid. FaultTracer: A change impact and regression fault analysis tool for evolving Java programs. In *FSE Demo*, pages 1–4, 2012.

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

[48] H. Zhong, L. Zhang, and H. Mei. An experimental study of four typical test suite reduction techniques. *IST*, 50(6):534–546, 2008.