More Precise Regression Test Selection
via Reasoning about Semantics-Modifying Changes

Yu Liu★, Jiyang Zhang★, Pengyu Nie★, Milos Gligoric★, Owolabi Legunsen†

★The University of Texas at Austin; †Cornell University

USA

{yuki.liu,jiyang.zhang,pynie,gligoric}@utexas.edu,legunsen@cornell.edu

ABSTRACT

Regression test selection (RTS) speeds up regression testing by only re-running tests that might be affected by code changes. Ideal RTS *safely* selects all affected tests and *precisely* selects only affected tests. But, aiming for this ideal is often slower than re-running all tests. So, recent RTS techniques use program analysis to trade precision for speed, i.e., lower regression testing time, or even use machine learning to trade safety for speed. We seek to make recent analysis-based RTS techniques more precise, to further speed up regression testing. Independent studies suggest that these techniques reached a "performance wall" in the speed-ups that they provide.

We manually inspect code changes to discover those that do not require re-running tests that are only affected by such changes. We categorize 29 kinds of changes that we find from five projects into 13 findings, 11 of which are semantics-modifying. We enhance two RTS techniques—Ekstazi and STARTS—to reason about our findings. Using 1,150 versions of 23 projects, we evaluate the impact on safety and precision of leveraging such changes. We also evaluate if our findings from a few projects can speed up regression testing in other projects. The results show that our enhancements are effective and they can generalize. On average, they result in selecting 41.7% and 31.8% fewer tests, and take 33.7% and 28.7% less time than Ekstazi and STARTS, respectively, with no loss in safety.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software evolution.*

KEYWORDS

Regression test selection, regression testing, semantics-modifying changes, change-impact analysis

1 INTRODUCTION

Regression testing is the dominant quality assurance approach today; it commonly re-runs all tests (*RetestAll*) to check that code changes do not introduce bugs. But, RetestAll costs are growing rapidly with increasing rates of updates and growth in code size [31, 70]. So, without cost-reducing automated techniques, developers may test less, or use manual *ad hoc* approaches that miss bugs [29].

Regression test selection (RTS) reduces regression testing costs by only re-running tests that are *affected* by changes. Affected tests are computed as those that transitively depend on changed code. Researchers studied RTS for decades [10, 11, 14, 17, 18, 21, 22, 27, 28, 30, 32–34, 37, 39, 41, 42, 50, 54–58, 60–62, 64, 65, 68, 69, 74] and recent techniques [5, 8, 9, 19, 26, 40, 53, 66] are being adopted.

Ideally, RTS would *safely* select <u>all</u> affected tests and *precisely* select <u>only</u> affected tests. But, RTS techniques that aim for safety and precision are often slower than RetestAll [25, 27, 50].

Recent RTS techniques that are being adopted make two kinds of trade-offs. First, some techniques based on program analysis trade precision for speed, i.e., lower end-to-end regression testing time, when selecting affected tests. The rationale is that developers likely prefer *safe but imprecise* RTS that is faster than RetestAll to *unsafe* RTS, or *safe and precise* RTS that is slower than RetestAll. Second, some techniques trade safety for speed, typically by training machine learning (ML) models to only select tests that may fail (i.e., not all affected tests) after a change [7, 44, 45]. The rationale is that a failing test suffices to initiate debugging.

We seek to speed up recent analysis-based RTS techniques because they seem to have reached a "performance wall"—a limit on how much they can speed up regression testing. Independent studies showed similar average ratios of selected tests and average time reduction on different projects and sets of project revisions [10, 39, 63, 67, 73]. So, the next generation of RTS techniques should break this performance wall to improve on the gains of the existing ones. Sections 7 and 8 position our work relative to ML-based RTS. There, we show that our improved analysis-based RTS performs better than the state-of-the-art ML-based RTS.

The technical challenge that we address in this paper is how to speed up regression testing by making analysis-based RTS more precise without sacrificing safety. We do so based on the idea that some semantics-*modifying* code changes do not require re-running all tests that analysis-based RTS selects. For example, suppose that the only change to a Java class that has no ancestors (except `java.lang.Object`) or descendants is the deletion of a method. If the resulting code compiles, then a class-level RTS technique need not re-run test classes that are only affected by this deletion. Our approach therefore generalizes related work like Reks [67],

which improves RTS precision by not re-running tests that are only affected by semantics-*preserving* changes, i.e., refactorings.

We hypothesize that identifying and reasoning about the kinds of changes that we target can further reduce end-to-end regression testing time at the cost of increased RTS analysis time. End-to-end regression testing time with RTS consists of *collection time* to find test dependencies, *analysis time* to find affected tests, and *execution time* to run selected tests.

To identify kinds of code changes for which analysis-based RTS may safely skip to re-run some tests, we manually inspect developer changes in 250 revisions of 5 open-source projects. We find 29 such kinds of changes and organize them into 13 findings, 11 of which are semantics-modifying. We do not claim to have discovered an exhaustive list of kinds of changes that can be used to make RTS more precise. Rather, we use those that we discover to investigate if RTS that reasons about them is effective for speeding up end-to-end regression testing time, and if information discovered from some projects help make RTS more precise in other projects. Future work can pursue the discovery of more of such kinds of changes.

We enhance Ekstazi (a dynamic class-level RTS technique) [26] and STARTS (a static class-level RTS technique) [40] to reason about the kinds of changes that we find. We call the enhanced techniques FineEkstazi and FineSTARTS; they enable us to evaluate the impact of our work on dynamic and static RTS paradigms. Finding and leveraging these kinds of changes is a one-time cost paid by RTS tool developers, not by the users of FineEkstazi and FineSTARTS.

We evaluate our enhanced RTS techniques on other projects and revisions than those from which we discover the kinds of changes that we use. We run FineEkstazi and FineSTARTS on 50 revisions each in 23 open source projects (total: 1,150 revisions). Then, we compare the number of selected tests, the end-to-end times, and the analyses times of FineEkstazi and FineSTARTS with those of Ekstazi and STARTS. We also evaluate the safety of FineEkstazi and FineSTARTS relative to Ekstazi and STARTS by using RTSCheck [75] to check all four implementations. Finally, to assess the prevalence of the kinds of changes that we leverage, we manually compare the kinds of changes in 250 of these 1,150 revisions with those that we used during the discovery process.

The results show that reasoning about these kinds of changes is effective for improving RTS precision and speeding up regression testing. FineEkstazi reduces the number of selected tests and the end-to-end regression testing time by as much as 80.8% (average: 41.7%) and 55.4% (average: 33.7%), compared to Ekstazi. The comparative numbers for FineSTARTS's improvement over STARTS are 71.2% (average: 31.8%) and 60.6% (average: 28.7%). FineEkstazi is also faster than HyRTS [73], which is a dynamic RTS tool that is more precise than Ekstazi and STARTS. But, future work could enhance HyRTS to reason about the kinds of changes that we find.

FineEkstazi and FineSTARTS make Ekstazi and STARTS more precise and speed up regression testing without sacrificing safety. Our analysis of the RTSCheck results shows that FineEkstazi and FineSTARTS do not incur any new safety violations compared to Ekstazi and STARTS. Lastly, our manual checks show that 250 (of 1,150) revisions contain several but not all of the kinds of changes that we find. Changes in the manually checked revisions map to 10 out of 13 kinds of changes. So, while the kinds of changes varied

```
1  class A {
2    public int m(int x, int y){ return x - y; }
3    public int m(int x, int y){ return x / y; }
4  }
5  class B {
6    public int m(int x, int y){ return x + y; }
7  }
8  class C {
9    public int m(int x, int y) throws Exception {
10     Object a = Class.forName("A").newInstance();
11     Method m = a.getClass().getMethod("m", ...);
12     return m.invoke(a, x, y);
13   }
14 }
```

```
1  class T1{
2    @Test void t1(){assertEqual(2, new A().m(5, 3);}
3  }
4  class T2{
5    @Test void t2(){assertEquals(8, new B().m(5, 3);}
6  }
7  class T3{
8    @Test void t3()
9      throws Exception{assertEquals(4, new C().m(7, 3);}
10   @Test void t4(){assertEquals(10, new B().m(7, 3);}
11 }
```

Figure 1: Code and tests for illustrating Ekstazi and STARTS.
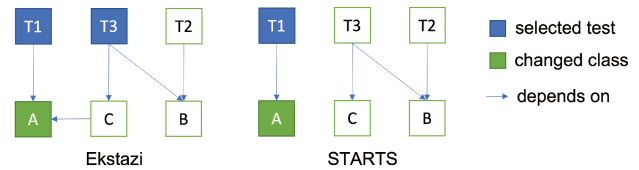

Figure 2: Ekstazi and STARTS dependency graphs for Fig. 1.

across programs' revision histories, the kinds of changes that we find apply beyond the projects from which we obtained them.

This paper makes the following contributions:

★ **Idea.** We use reasoning about semantics-modifying changes to make RTS more precise without sacrificing safety.
★ **Kinds of changes.** We produce an initial set of kinds of changes that future work can build on to make RTS even more precise.
★ **Tools.** We develop FineEkstazi and FineSTARTS as manual-analysis driven enhancements of Ekstazi and STARTS.
★ **Evaluation.** We find that reasoning about these kinds of changes speeds up regression testing and can generalize across projects.

Our data is at https://github.com/EngineeringSoftware/FineRTS.

2 BACKGROUND AND EXAMPLES

We use examples to describe Ekstazi, STARTS, and semantics-modifying changes for which it is safe to not re-run some tests.

Recent RTS techniques. Ekstazi and STARTS are recent analysis-based class-level RTS techniques; they find changed *classes* and affected *test classes* based on a *class*-level dependency graph. Ekstazi creates its dependency graph dynamically, but STARTS builds its dependency graph statically. Figures 1 and 2 to illustrate their similarities and differences. Figure 1 shows classes A, B, C in a synthetic code under test (CUT), and test classes T1, T2, and T3, which check them. Two versions of the CUT are shown in A.m, which used to compute difference (old version in red) but now computes division (new

```
1  public class Base64Test {
2-   private static final String[] BASE64_IMPOSSIBLE_CASES = {
3+   static final String[] BASE64_IMPOSSIBLE_CASES = {
4      "ZE==", "ZmC=", "Zm9vYE==", "Zm9vYmC=", "AB", };
```

Figure 3: Changing a field's access modifier.

```
1  public class Percentile extends AbstractUnivariateStatistic ...
2-   public Percentile(final double quantile) throws
3-    MathIllegalArgumentException {
4+   public Percentile(final double quantile) {
```

Figure 4: Removing `throws` clause from method signature.

```
1  public abstract class Email
2+   public String getHeader(final String header)
3+   {  return headers.get(header); }
```

Figure 5: Adding a method to a class.

version in green). Figure 2 shows the dependency graphs that Ek-stazi and STARTS compute and the tests that they select. STARTS is unsafe as it does not detect that T3 depends on the changed class A because C.m uses reflection to invoke A.m. STARTS can also be less precise due to dynamic dispatch. Legunsen et al. [39, 60] found that Ekstazi and STARTS have similar end-to-end times and reflection rarely makes STARTS unsafe in practice.

Some kinds of change that we use. We give several examples of semantics-modifying changes for which Ekstazi and STARTS re-run affected tests, and illustrate why it is safe to not re-run tests that are only affected by such changes. The examples are simplified from changes in open-source projects; we show only relevant code.

The change in Figure 3 (from the Apache codec project [1], revision a6b2f1) removes the `private` access modifier on a static field and the project's code still compiles. Ekstazi and STARTS re-run `Base64Test` (it depends on itself), but doing so is needless: other classes that access the field must use reflection to do so, but reflection is not used here. So, if the only changes are to access modifiers, code still compiles, and the project does not use reflection to access the changed fields, then it is safe to not re-run tests that are only affected by such changes.

The change in Figure 4 (from the Apache math project [3], revision 802058f) only deletes a `throws` clause. Ekstazi and STARTS needlessly re-run 15 and 22 test classes. No class uses reflection to check method signatures, so tests that are only affected by this change will behave the same before and after the change. It is safe to not re-run such tests.

As a final example, in Figure 5 (from the Apache email project [2], revision 78b9fdf) a new method is added. Ekstazi and STARTS re-run eight and nine test classes unnecessarily, together with a test class that was changed to depend on the new method. It is safe to not re-run those eight and nine test classes: they do not transitively depend on the new method.

Our goal is to study how to find these kinds of changes and enhance analysis-based RTS to reason about them, to improve RTS precision. Section 3 describes our process for finding kinds of changes that can be used, and Section 4 explains how we enhance Ekstazi and STARTS to reason about these kinds of changes.

3 MANUAL ANALYSIS OF CHANGES

We describe how we manually find and categorize the kinds of changes that we use, and discuss how many of these kinds of changes the RTS tools in this paper use. To re-emphasize, we do not claim that the kinds of changes that we find are exhaustive. We only show that it is feasible to find these kinds of changes and to improve RTS precision (and speed up regression testing) by reasoning about them. Future work could find more kinds of changes.

3.1 Manual Analysis Process

We manually analyze the nature of changes in 50 revisions per project in 5 projects that are shown in Table 1 with the revisions that we start from. We follow four steps:

(1) Choosing revisions. Per project, we randomly choose a revision from 2019 and 50 contiguous subsequent revisions. Our rationale for choosing these projects and revisions is in Section 5.1.

(2) Inspection. We manually inspect all changes to Java files in all 250 revisions and record the changed program elements (e.g., class, method) and our decisions on if each change, by itself, is safe for RTS to ignore. Three co-authors performed the inspection; one of them did initial inspection and then met with the other two to discuss and find agreement over a period of 2 months. Some decisions are challenging and depend on context. For example, if an instance method is added, whether tests that are only affected by that change can be ignored depends on if a call to a method with the same signature exists in the same class hierarchy as the new method.

Table 1: Manually inspected projects

Name	SHA
beanutils	50a9457
codec	6cf3482
compress	80a388e
pool	41c4df1
fastjson	6b1ed5f

(3) Categorization. We organize our *findings* on the kinds of changes, and aggregate the number of Java files, revisions, and projects related to each kind of code change.

(4) Checking RTS behavior. For each kind of change, we confirm that Ekstazi or STARTS selects at least one test, as a sanity check on our decisions, and to provide initial data for evaluating our enhanced RTS techniques.

3.2 Findings from Manual Analysis

In Table 2, we organize the 29 kinds of changes that we observe during manual analysis into 13 findings that can be used to improve RTS precision. We group kinds of changes that are similar or that modify similar program elements, if they are likely to induce the same test-selection behavior in our enhanced RTS techniques. ID in Table 2 is a label that we use to refer to each finding; the caption in that table describes other columns.

Here, we discuss the Kind of Change rows in Table 2 that are not self-explanatory. Sorting fields or methods (F3) is a refactoring; RTS should not re-run tests. Reks [67] is the only refactoring-aware RTS technique that we know, and it *does not* handle this refactoring. Yet, F3 applies to many files that we analyze.

We explain some other semantics-modifying kinds of changes.
(1) "Import method from different package" (F7): the package from which a method is imported has changed.
(2) "Import field type from different package" (F7): the package from which a field's type is imported has changed.

Table 2: Findings from our formative study. KIND OF CHANGE: description; *#F* no. of source files with each kind of code change; *#S*: no. of revisions with each kind of change; *#P*: no. of inspected projects (out of 5) with each kind of change; REASONING: why the kind of change can be used to improve RTS precision.

ID		KIND OF CHANGE	#F	#S	#P	REASONING
F1	a	Add class	133	39	5	Evaluated RTS techniques already handle these properly.
F2	a	Add instance method	58	43	5	If no method with same signature is invoked on instances of the modified class, then tests that are only affected by such change can be safely skipped.
	b	Remove instance method	1	1	1	
	c	Remove static method	1	1	1	
	d	Add constructor	1	1	1	
	e	Add static method	5	5	3	
F3	a	Sort members	42	5	3	This is a refactoring for which tests need not be re-run.
F4	a	Add field	18	13	3	Tests that do not create instances of modified class will not change behavior.
	b	Remove field	3	3	2	
	c	Add static initializer block	1	1	1	
F5	a	Change anonymous class to lambda	18	2	2	If these are the only changes, affected tests do not need to be rerun.
F6	a	Rename class	9	3	2	Refactorings for which no tests do not need to be re-run. The compiler will catch improper renamings.
	b	Rename instance method	4	4	3	
	c	Rename static method	1	1	1	
F7	a	Import field type from different package	1	1	1	Affected tests can be re-run based on method-level reasoning.
	b	Modify field initialization	9	5	3	
	c	Import method from different package	1	1	1	
F8	a	Add exception to method	6	4	3	If these are the only changes, affected tests do not need to be re-run if no test dependency uses reflection.
	b	Modify throws clause	1	1	1	
	c	Modify method parameter	2	2	1	
F9	a	Modify class access modifier	3	2	2	If these are the only changes, affected tests do not need to be re-run if no test dependency uses reflection.
	b	Make field final	2	2	1	
	c	Modify field access modifier	2	2	2	
F10	a	Modify a constructor	5	3	2	Affected tests can be re-run based on method-level reasoning.
F11	a	Specialize parameter type	3	2	2	No need to re-run tests if these are the only changes because bytecode of affected (dependent) class has changed.
	b	Add/Change base class to hierarchy	1	1	1	
F12	a	Add runtime annotation	3	3	1	There is no need to re-run tests if there is no reflection (no runtime annotation). Affected tests can be re-run based on method-level reasoning if there is reflection, and annotation is method annotation, parameter annotation, or field annotation.
F13	a	Replace parameter with lambda expression	1	1	1	Affected tests can be re-run based on method-level reasoning.
	b	Compiler modifies bytecode structure	12	6	3	

(3) "Replace parameter with lambda expression" (F13): a lambda expression is passed as a parameter to a changed API.

(4) "Compiler modifies bytecode structure" (F13): compiler optimizations change bytecode structure but not functionality, e.g., constant propagation or synthetic method introduction.

Table 2 is based only on the file-level diffs that we analyze. If a file contains more than one kind of change in one diff, we increment the count of each kind by one. The kinds of changes in Table 2 are only from 95 unique revisions; the remaining (155) revisions merely modify bytecode metadata, or re-format code. Also, the kinds of changes that we identify are from 332 unique files.

Findings that RTS techniques handle. Table 3 shows which findings are supported by recent RTS techniques and the enhanced techniques that we introduce in this paper. There, ✓ or ✗ means that a tool uses or does not use a finding, respectively. Our enhanced techniques are marked with ★. Of 13 findings, 11 are semantics-modifying changes and they are marked with †. Table 3 also shows that we do not yet use some findings from our manual analysis: F5, F12, and F13. Non-trivial compiler support is needed to leverage F5 and F13. F12 has no effect unless reflection is used; we do not implement it since reflection is rare in our evaluated subjects.

Table 3: Findings that RTS techniques support.

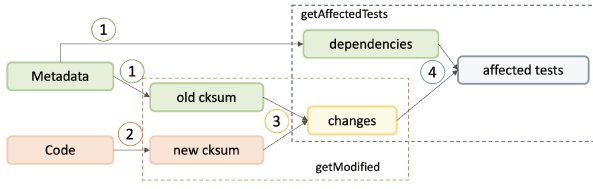| ID	EKSTAZI	STARTS	REKS	★FINEEKSTAZI	★FINESTARTS
F1 †	✓	✓	✓	✓	✓
F2 †	✗	✗	✗	✓	✓
F3	✗	✗	✗	✓	✓
F4 †	✗	✗	✗	✓	✓
F5 †	✗	✗	✗	✗	✗
F6	✗	✗	✓	✓	✓
F7 †	✗	✗	✗	✓	✓
F8 †	✗	✗	✗	✓	✓
F9 †	✗	✗	✗	✓	✓
F10 †	✗	✗	✗	✓	✓
F11 †	✗	✗	✗	✓	✓
F12 †	✗	✗	✗	✗	✗
F13 †	✗	✗	✗	✗	✗

Figure 6: RTS workflow.

4 TECHNIQUE

FineEkstazi and FineSTARTS use the kinds of changes in Section 3 by enhancing Ekstazi and STARTS, respectively, to not select tests that are only affected by such changes. We show how the enhanced techniques differ from Ekstazi and STARTS (Section 4.1), and describe how the enhanced techniques work (Section 4.2).

4.1 Overview of Original vs. Enhanced RTS

Figure 6 shows the steps in Ekstazi and STARTS that we enhance in FineEkstazi and FineSTARTS. The inputs are code (and tests) for the current revision and metadata about test dependencies collected from the previous revision. The outputs are affected *test classes*. Ekstazi and STARTS metadata contains checksums per class and a mapping between tests and classes they transitively depend on.

Ekstazi and STARTS work in four main steps. Step ① loads the old revision's metadata from disk. Step ② computes metadata in the new revision; we will refer to this step as *getNewMetaData*. Step ③ computes changed classes using the old and new metadata; we will refer to this step as *getModified*. Finally, step ④ computes affected test classes as those for which at least one dependency changed; we will refer to this step as *getAffectedTests*.

FineEkstazi and FineSTARTS follow the same main steps as Ekstazi and STARTS, but they differ in two ways:

(1) **Structurally.** Ekstazi and STARTS work only at the class-level but FineEkstazi and FineSTARTS work across class, method, and field levels of program granularity.

(2) **Algorithmically.** FineEkstazi and FineSTARTS have different *getNewMetaData*, *getModified*, and *getAffectedTests* steps than Ekstazi and STARTS, to reason across granularity levels.

Here, we describe the metadata that FineEkstazi and FineSTARTS collect, and how their *getNewMetaData* steps diverge from the ones in Ekstazi and STARTS. We will highlight the other differences between the original and enhanced RTS techniques in Section 4.2. The data structures used by RTS techniques to store metadata is important for their practicality and central to their algorithms.

Ekstazi and STARTS collect metadata for each test class t as $t \rightarrow \{C_1 : \mathsf{chksum}(C_1), C_2 : \mathsf{chksum}(C_2), C_3 : \mathsf{chksum}(C_3)\}$. We do not describe $\mathsf{chksum}(C_i)$ in detail (see [26, 27, 40]): it computes a checksum for each class C_i that t transitively depends on by removing debug information and hashing the remaining bytecode.

FineEkstazi and FineSTARTS use a more complex data structure to collect metadata so that they can capture relationships across three different levels of program granularity. Specifically, they collect metadata for each test class t as $t \rightarrow \{D_1, D_2, D_3, \ldots\}$ where each D_i is a map $C_i \rightarrow (f_i, n_i, m_i)$ for each class C_i that t depends on. Each C_i maps to three sets: (1) $f_i = \{(\mathsf{name}_j, \mathsf{descriptor}_j, \mathsf{value}_j) \mid$

j is a field in $C_i\}$; (2) $n_i = \{k \rightarrow \mathsf{chkSum}(k) \mid k$ is a constructor or static initializer in $C_i\}$; and (3) $m_i = \{(\mathsf{name}_l, \mathsf{descriptor}_l) \rightarrow \mathsf{chkSum}(l) \mid l$ is a method in $C_i\}$.

The chkSum that FineEkstazi and FineSTARTS use is the same as in Ekstazi and STARTS, except that we apply it to parts of a class instead of the whole class. Note that the way that the original and enhanced algorithms use the metadata is also different. Ekstazi and STARTS compute affected test classes using the set of test classes in the new revision and the metadata from the old revision. The *getModified* step in Ekstazi and STARTS considers a class as changed if its checksum in the current and old revisions differ. The *getModified* step in FineEkstazi and FineSTARTS is more complicated because of the need to reason across granularity levels. We describe *getModified* as part of the algorithms in Section 4.2.

4.2 How FineEkstazi and FineSTARTS work

Algorithm 1 shows the *getAffectedTests* procedure that FineEkstazi and FineSTARTS use. Instead of just checking if a class changes (like Ekstazi and STARTS do), Algorithm 1 additionally checks if and how the fields, constructors (including static initializers), and methods are modified. Reasoning across multiple granularity levels is needed to benefit from the kinds of changes that we found in Section 3 for safely speeding up RTS. HyRTS [73] also reasons across class and method granularity, but our enhanced techniques also reason about fields. We experimentally compare FineEkstazi and FineSTARTS with HyRTS in Section 5.

In Algorithm 1, *getAffectedTests* iterates over each test class t in the new revision and calls the *getModified* on each class C that t depends on. If *test* is a newly added test class, on line 4, it is added to the set of affected tests. If any t that depends on C is only affected by changes that match our findings, *getAffectedTests* will not return t as an affected test class.

On line 13, if C's metadata is unchanged, *getModified* returns `false`: no test that depends on C should be selected. If line 14 is reached, then C changed. All that remains is to check whether the change warrants re-running tests that depend on C. To do so, *getModified* calls the *fldChanged*, *conChanged*, and *mtdChanged* procedures to check if the change matches our findings at the field, constructor, and method levels respectively. If no change is detected, line 21 returns `false`.

Procedure *fldChanged* checks if the set of fields changed. If the change is not just adding new fields or deleting old fields, then line 24 will return true. In that case, all tests that depend on C will be selected. This is an imprecise check of change to fields because we cannot distinguish between "renaming a field" and "deleting a field with old name and adding a field with new name". Renaming fields, which is a refactoring, is not ignored by our algorithm and all tests that depend on C are selected. Access modifiers are not stored in f, so, if field access modifiers are the only changed part of C, `fldChanged` will return `false`.

If fields in C did not change, *conChanged* returns `true` if a constructor or static initializer changed (line 27) so that all tests that depend on C are selected. Constructors and static initializers are typically much fewer than other kinds of methods. So checking them first in *conChanged* makes *getAffectedTests* faster.

Algorithm 1 *getAffectedTests* for FineEkstazi and FineSTARTS

Inputs: T: the set of test classes in the new revision, $M: t \rightarrow D$
▷ Section 4.1 describes D

Outputs: $T^a \subseteq T$: affected test classes
1: **procedure** getAffectedTests(T, M)
2:   $T^a \leftarrow \{\}$
3:   **for all** test **in** T **do**
4:     **if** test $\notin M$ **then**   ▷ *test* is a newly added test class
5:       $T^a \leftarrow T^a \cup \{\text{test}\};$ **continue**
6:     **for all** C **in** $M[\text{test}].\text{keys}()$ **do**
7:       **if** getModified(test, C, M) **then**   ▷ *test* should be re-run
8:         $T^a \leftarrow T^a \cup \{\text{test}\};$ **break**
9:   **return** T^a
10:
11: **procedure** getModified(test, C, M)
12:   $I^{new} \leftarrow$ getNewMetaData(C)
13:   **if** $M[\text{test}][C] = I^{new}$ **then return** false   ▷ C did not change
14:   **else** ▷ did fields, constructors, initializers, or methods in C change?
15:     **for all** f **in** getFieldData($M[\text{test}][C]$) **do**
16:       **if** fldChanged($M[\text{test}][C][f], I^{new}[f]$) **then return** true
17:     **for all** n **in** getConstructorAndInitData($M[\text{test}][C]$) **do**
18:       **if** conChanged($M[\text{test}][C][n], I^{new}[n]$) **then return** true
19:     **for all** m **in** getMethodData($M[\text{test}][C]$) **do**
20:       **if** mtdChanged($M[\text{test}][C][m], I^{new}[m], C$) **then return** true
21:     **return** false
22:
23: **procedure** fldChanged(f, f^{new})
24:   **return** $\neg(f \setminus f^{new} = \emptyset \vee f^{new} \setminus f = \emptyset)$ ▷ true if field info changed
25:
26: **procedure** conChanged(n, n^{new})
27:   **return** $n \neq n^{new}$ ▷ true if constructor or static initializer changed
28:
29: **procedure** mtdChanged(m, m^{new}, C)
30:   **for all** sig **in** (m.keys() $\cup$ m^{new}.keys()).copy() **do**
                   ▷ old and new signatures
31:     **if** sig $\in$ m **and** sig $\in$ m^{new} **then**   ▷ unchanged signatures
32:       **if** m[sig] = m^{new}[sig] **then**
                  ▷ same bytecode; ignore the change
33:         m$^{new} \leftarrow$ m$^{new} \setminus \{(\text{sig}, m^{new}[\text{sig}])\};$ m $\leftarrow$ m $\setminus \{(\text{sig}, m[\text{sig}])\}$
34:       **else return** true   ▷ change: same signature, different bytecode
35:     **else if** m^{new}[sig] $\in$ m.values() **or** m[sig] $\in$ m^{new}.values() **then**
        ▷ found old bytecode with new signature; ignore the change
36:       m $\leftarrow$ m $\setminus \{(\text{sig}, m^{new}[\text{sig}])\};$ m$^{new} \leftarrow$ m$^{new} \setminus \{(\text{sig}, m[\text{sig}])\}$
37:   cHasHrchy $\leftarrow$ hasHrchy(C) **or** hadHrchy(C)
38:   **if** !cHasHrchy **and** (m = $\emptyset$ **or** m^{new} = $\emptyset$) **then**
    ▷ one empty map: method added or deleted without affecting hierarchy
39:     **return** false
40:   **else if** $C \in T$ **and** m^{new} = $\emptyset$ **then**
                ▷ deleted a method from a test class
41:     **return** false
42:   **return** true

Finally, *mtdChanged* performs method-level reasoning. The union of (signature, bytecode) pairs for all methods in the old and new revisions is iterated over to check for changes. If the pair for method m_i is the same in the old and new revisions, m_i did not change and *mtdChanged* proceeds with the next method, m_{i+1} (line 32). If the signatures are the same but the bytecode differ, then the method changed and line 34 returns true. On line 35, if the signatures differ

---

Algorithm 2 Embedding mRTS in FineEkstazi

Inputs: Test class t, Ekstazi metadata *.ekstazi*, mRTS metadata *.mrts*
Outputs: true if the test should run; false otherwise
1: **procedure** Affected(t, .ekstazi, .mrts)
2:   $cg \leftarrow$ FineEkstazi.getModifiedClasses(t, .ekstazi)
3:   **if** $cg = \emptyset$ **then**   ▷ Nothing is modified
4:     **return** $false$
5:   $mg \leftarrow$ mRTS.getModifiedClasses(t, .mrts)
6:   **if** $cg \subsetneq mg$ **then**   ▷ Reflection or third-party class
7:     **return** $true$
8:   **for** clz : cg **do**
9:     **if** mRTS.isModified(t, clz, .mrts) **then**
10:       **return** $true$
11:   **return** $false$

---

but the bytecode is the same, m_i was refactored—no test should be selected—and *mtdChanged* proceeds with m_{i+1}.

On line 38, if exactly one revision's map is empty then a method must have been added to or deleted from a class C. If C is not part of a class hierarchy, and assuming the code compiles, then no test that only depends on C should be selected, so *mtdChanged* returns false. Line 40 deals with a special case: if a test method is added/revised, it is considered as changed. Speedups result when the only change is to delete a method in a test class—the remaining tests cannot be affected by the deleted test if the code still compiles.

4.3 Embedding Method-Level Reasoning

Using several findings in Section 3, e.g., F10, to improve RTS precision requires method-level reasoning. So, we develop a static method-level analysis (mRTS) that can be *combined* with class-level analyses to improve their precision without making them less safe. Algorithm 2 shows how we integrate mRTS analysis into FineEkstazi; its inputs are a test class, Ekstazi metadata (.ekstazi), and mRTS metadata (.mrts). It outputs true if the test should be selected and false otherwise. The analogous algorithm for embedding mRTS into FineSTARTS works in a similar way.

Line 2 obtains a set of modified classes (cg) for test t (as identified by FineEkstazi). If the set is empty (line 3), false is returned—t is not affected. If the set is not empty, mRTS is called to obtain the set of modified classes mg. If $cg \not\subset mg$, then t is affected for one of two reasons (1) a third-party class is modified (which is not tracked by mRTS), or (2) reflection is used to access some classes (which are not captured by mRTS). If $cg \subset mg$, lines 8-10 iterate over cg. For each $C \in cg$ line 9 invokes mRTS and returns true if the mRTS finds that t is affected. If mRTS analysis returns false for all changed classes, then t is not affected (line 11).

5 EVALUATION

We evaluate the effectiveness of using our manual analysis findings for improving RTS precision. We address these research questions:

- **RQ1:** How much does using the kinds of changes in our manual analysis reduce the *tests selected* by Ekstazi and STARTS?

- **RQ2:** How much does using the kinds of changes in our manual analysis reduce the *end-to-end time* of Ekstazi and STARTS?

- **RQ3:** What is the impact of using the kinds of changes in our manual analysis on the *safety* of Ekstazi and STARTS?

- **RQ4:** To what extent do the findings from our manual analysis re-occur in other projects and revisions that we did not analyze?

5.1 Experimental Setup

Evaluation subjects and revisions: Table 4 shows the 23 projects that we evaluate (sorted by average test time), plus some characteristics. Projects in our manual analysis are highlighted in gray. We choose 18 of the projects in Table 4 because they are widely used in RTS research, and prior work [39, 40, 60] showed that Ekstazi and STARTS work well on many of their revisions. We omit four projects from prior RTS research where average test time is less than 10 seconds. We added another five projects that we are familiar with and whose tests run longer than 10 seconds.

To find revisions, for each project, we run `git diff` on its revision history until we found 50 revisions where at least one Java file was modified, and the project compiles. For projects in our manual analysis, we used different sets of 50 revisions in our evaluation. Doing so reduces the chance that FineEkstazi and FineSTARTS are tuned to the projects and versions from which we obtained our findings. In a sense, we aim to not "overfit" the data.

Running experiments: We run RetestAll, Ekstazi, FineEkstazi^F, FineEkstazi, STARTS, FineSTARTS^F, FineSTARTS, and HyRTS on each of the 50 revisions in the 23 projects. FineEkstazi^F and FineSTARTS^F do not use mRTS.

We record the number of test classes selected and the end-to-end RTS time. We run separate experiments to collect the analysis (A), test execution (E), and collection (C) times. Measuring A, E, and C times enables us to analyze the time costs of reasoning about these kinds of changes during RTS. We run all experiments on a 3.20 GHz Intel® Core™ machine with 64GB of RAM, running Ubuntu Linux 18.04 and Oracle Java 64-Bit Server version 1.8.0_241.

5.2 RQ1: Impact on RTS Selection Rates

We evaluate whether and by how much FineEkstazi^F, FineEkstazi, FineSTARTS^F, and FineSTARTS select fewer tests than Ekstazi and STARTS. We also compare with HyRTS. Figure 7 shows the percentage of all tests selected by these techniques per project. Note that HyRTS crashed on P13, P17, and P18; we mark them as N/A and exclude them when comparing with other tools. Exact numbers of tests are shown in the appendix in our data package [4].

Reasoning about findings from our manual analysis reduces selection rates, compared with Ekstazi and STARTS, in every evaluation subject. Also, combining with method-level analysis (mRTS) further reduced selection rates. Reasoning about those findings without mRTS yields up to 46.5% (average: 17.8%) reduction in tests selected by Ekstazi, and up to 50.4% (average: 16.0%) reduction for STARTS. If the code-change information and mRTS are used together, these reductions grow to 80.8% (average: 41.7%) and 71.2% (average: 31.8%), respectively. The selection rates of FineEkstazi are on par with those of HyRTS: FineEkstazi selects fewer tests in 7 of 20 projects (HyRTS fails on 3 projects). Both use dynamic cross-granularity analysis; FineEkstazi uses field-, method-, and

Table 4: Projects in our study.

PID	Name	SHA	KLOC	#Test	Test Time (s)
P1	imaging	70dd698	39.3	115.2	15.6
P2	lang	bff7521	78.1	148.8	16.6
P3	collections	954c29f	63.7	170.4	17.6
P4	asterisk-Java	aca95a7	60.1	46.0	20.1
P5	codec	35e9cf2	23.9	60.7	22.4
P6	configuration	7e4b3fa	51.6	169.0	25.2
P7	compress	8a65cc9	50.2	142.6	28.9
P8	gerrit-events	6585777	8.1	24.0	30.1
P9	tabula-java	5f43a93	6.8	15.3	46.2
P10	fastjson	3ea25de	178.2	2297.0	47.8
P11	math	dff1a09	149.7	467.6	50.3
P12	net	48e0662	28.3	44.0	62.3
P13	beanutils	85b8cc9	33.5	102.6	74.6
P14	rxjava-extras	62fb6a3	13.9	48.3	89.4
P15	dbcp	64a3b97	31.4	42.6	92.9
P16	io	fc418a7	34.4	112.9	113.1
P17	b.HikariCP	acc9ac7	11.9	39.3	157.2
P18	sdk-rest	1617bb1	65.2	24.0	169.8
P19	email-ext-plugin	8761c27	12.9	38.3	231.7
P20	pool	be87cfc	14.6	22.0	333.8
P21	LogicNG	1bcead7	49.6	120.0	336.9
P22	finmath-lib	03befd8	76.6	100.6	1185.9
P23	lmdbjava	680e0a8	5.6	14.3	1308.6

class-level analyses, while HyRTS uses only method- and class-level analyses. Future work could enhance HyRTS precision by enhancing it to reason about our findings.

5.3 RQ2: Impact on End-to-End Testing Times

We measure how reduced selection rates (Section 5.2) translate to reductions in end-to-end regression testing times. Recall that end-to-end time with RTS includes *analysis time* (to reason about changes and find affected tests), *execution time* (to run selected tests), and *collection time* (to create metadata for performing RTS on the next revision). Figure 8 shows the percentage of end-to-end time of the RTS tools compared to RetestAll time. This percentage can be greater than 100% if RTS incurs significant overhead (analysis and collection times). Exact end-to-end times, and breakdown of A, E, and C times are in the appendix in our data package [4].

Compared to Ekstazi and STARTS, FineEkstazi and FineSTARTS reduce end-to-end times by up to 44.6% (average: 13.7%) and 42.9% (average: 12.5%), respectively, without mRTS. These reductions are up to 74.0% (average: 33.7%) and 68.0% (average: 28.7%), respectively, when also using mRTS. But, reasoning about our findings increases RTS analysis time, leading to an increase (rather than a decrease) in the end-to-end times with FineEkstazi and FineSTARTS in some cases. FineEkstazi is faster than HyRTS in 15 out of 20 projects. In a few cases, HyRTS takes longer time than RetestAll (up to 199.4% of RetestAll time), showing that it can incur high analysis and collection times to obtain its high precision.

Concerning the analysis, execution, and collection parts of end-to-end time, we find that FineEkstazi and FineSTARTS trade a
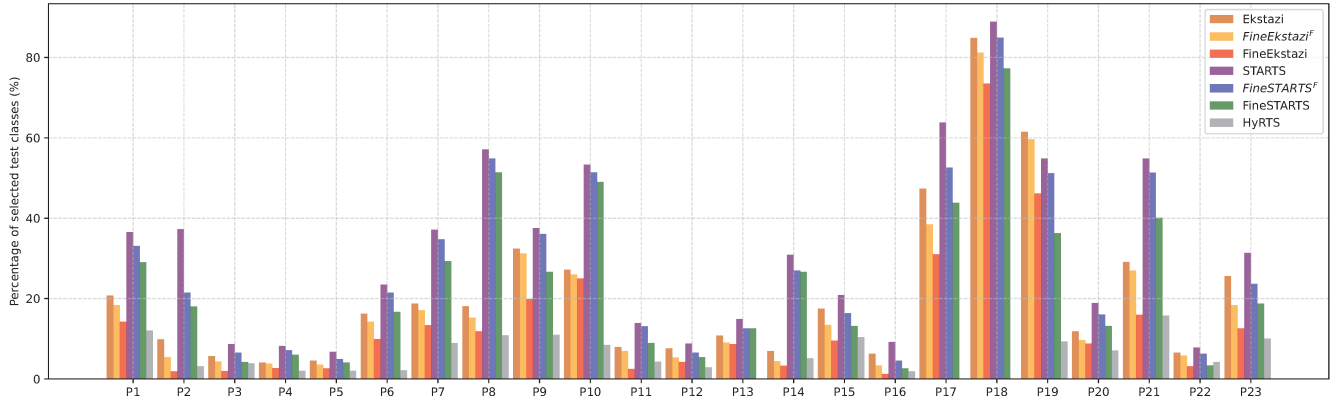
Figure 7: Percentage of number of selected test classes of RTS tools over RetestAll.
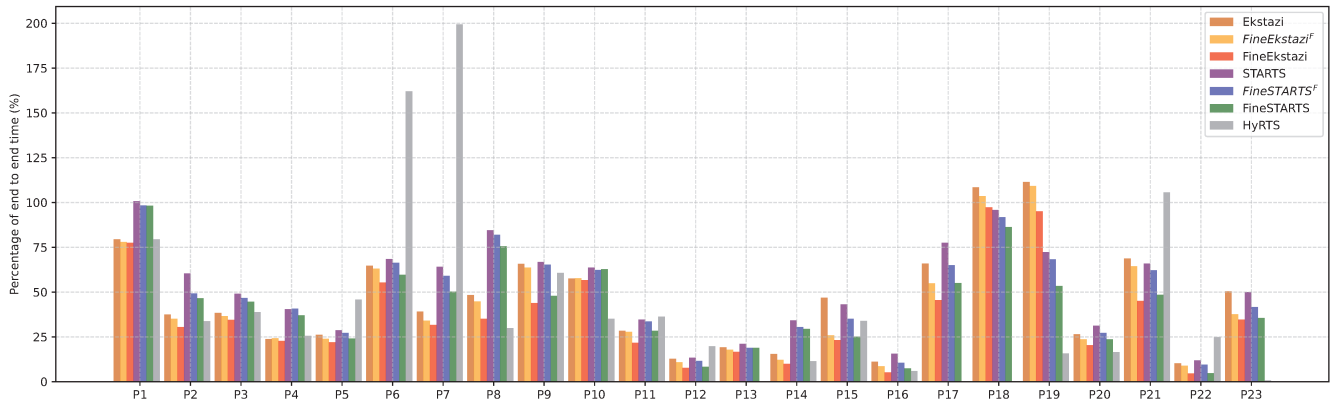

Figure 8: Percentage of end-to-end time of RTS tools over RetestAll.

higher analysis time for improved precision and reduced end-to-end times. Without mRTS, the analyses times of FineEkstazi and FineSTARTS are up to 28.5% (average: 14.1%) and 155.5% (average: 65.6%) higher than those of Ekstazi and STARTS, respectively. The analyses times are higher when mRTS is also used: 87.8% (average: 25.9%) and 398.0% (average: 182.4%) for FineEkstazi and FineS-TARTS, respectively. FineEkstazi and FineSTARTS are still able to reduce end-to-end times because (1) the analyses times of Ekstazi and STARTS are very small both in absolute numbers and as percentages of end-to-end times (on average, 0.9% for Ekstazi and 0.1% for STARTS); and (2) despite the increase, the analyses costs of FineEkstazi and FineSTARTS are still small portions of end-to-end time (1.2% and 0.3%, respectively, on average).

5.4 RQ3: Impact on Safety

We use RTSCheck [75], the state-of-the-art technique for testing RTS tools, to check FineEkstazi and FineSTARTS correctness and efficiency. Inputs to RTSCheck are RTS tools, and the outputs are each tool's number of safety, precision, and generality violations (a generality violation shows problems with integrating an RTS tool). The AutoEP, DefectsEP and EvoEP components of RTSCheck respectively use thousands of automatically generated evolving

Table 5: Violations that RTSCheck finds in RTS tools.

Rule	Ekstazi	FineEkstazi	STARTS	FineSTARTS
R1	2	2	6	6
R2	0	0	0	0
R3	6624	4490	6655	4482
R4	0	0	0	0
R5	0	0	0	0
R6	0	0	1	1
R7	3	3	0	0

programs, the Defects4J benchmarks [36], and GitHub revisions to check RTS tools. RTSCheck uses seven rules: R1 and R2 yield safety violations; R3, R4 and R5 yield precision violations; and R6 and R7 yield generality violations.

Table 5 shows the number of RTSCheck violations found in Ekstazi, FineEkstazi, STARTS, and FineSTARTS. Our enhanced techniques did not introduce new safety violations, despite the reduced selection rates.

We manually checked these violations. R1 violations (selecting fewer failing tests than RetestAll) are caused by (1) all tools not

Table 6: Applicabilty of manual analysis findings to other projects and versions. #F: no. of source files with each kind of code change; #S: no. of revisions with each kind of change.

ID		Kind of Change	#F	#S
F1	a	Add class	4052	190
F2	a	Add instance method	688	285
	b	Remove instance method	322	96
	c	Remove static method	55	32
	d	Add constructor	0	0
	e	Add static method	218	101
F4	a	Add field	273	156
	b	Remove field	74	41
	c	Add static initialized block	0	0
F8		Change signature	125	57
	a	Add exception to method	/	/
	b	Modify throws clause	/	/
	c	Modify method parameter	/	/
F10	a	Modify a constructor	749	233
F11	a	Specialize parameter type	0	0
	b	Add/Modify base class to hierarchy	59	24
		No change	3204	75
F3	a	Sort members	/	/
F6	b	Rename instance method	/	/
	c	Rename static method	/	/
F7	a	Modify field holding version	/	/
	b	Change field initialization	/	/
	c	Modify utilized API interfaces	/	/
F9	a	Modify class access modifier	/	/
	b	Make field final	/	/
	c	Modify field access modifier	/	/
Method			3801	540
Summary			13845	721

detecting changes to non-Java files; (2) STARTS missing static dependencies between Suite (JUnit3 style) and tests. R3 violations (selecting all tests in all versions) are because the programs generated by AutoEP have only a couple of tests, and safe RTS tools may have to select all tests in some programs. Our techniques have less R3 violations than existing RTS tools, which means that our techniques improve the precision. The R6 violation (selecting a different number of tests than RetestAll in the first version) from STARTS is caused by incompatibility with a third-party library version. R7 violations (selecting more failed tests than RetestAll) from EKSTAZI are caused by (1) unexpectedly triggering JUnit4 annotations with JUnit3; (2) improper support for a third-party library's annotations.

5.5 RQ4: Spread of Manual Analysis Findings

Table 6 shows the frequency of the kinds of change in our manual analysis in revisions and projects that we did not analyze. Table 6 omits findings that we do not support (Section 3.2). The "Method" row sums all kinds of changes that occurred at the method-level; the "Summary" row sums all kinds of changes. We automate the categorization of these changes. We count the three kinds of changes in F8 together as "Change signature". Also, F3, F6, F7, F9 are hard to automatically count separately, so we count them together as shown in the "No change" row. They apply to the same branch ("return false" on line 21 in Algorithm 1) after comparing methods, fields, and constructors, and we do not insert more branches to distinguish these four findings to save analysis cost. For example, F3 only changes the constant pool of the class, but does not change the bytecode of methods, fields, and constructors.

Many changes are at the method-level: 3801/13845 of files and 540/721 of revisions. Our results (Section 3.2) show that reasoning about our manual analysis findings complements combining class- and method-level analyses. We show that (1) *by itself*, reasoning about our findings improves RTS precision; and (2) using our findings *together* with method-level reasoning further improves RTS precision. Zhang [73] proposed using method-level analysis to improve RTS precision. But, we are the first to find and reason about semantics-modifying changes, and to use a two-tiered approach based on *both* our findings and mixed-granularity reasoning.

6 DISCUSSION

We discuss the manual effort involved in our approach, experimental comparison with ML-based RTS, and future work.

On manual effort. The manual effort to find the kinds of changes that we use can be non-trivial. But, we do not expect RTS tool users to spend this manual effort. Rather, it is researchers and RTS tool developers who may invest in finding these kinds of changes. Also, note that finding the kinds of changes and enhancing RTS techniques to leverage them is a one-time cost, unlike ML models that may need to be trained per project and across revisions. The manual effort that RTS tool developers invest is expected to result in a pay-offs for the tool users: increased productivity (shorter testing time), reduced impact on climate (less energy expended on testing), and higher-quality code (tests are run more frequently).

Comparison with ML-based RTS. This paper improves analysis-based RTS precision. But readers may wonder if ML-based RTS could perform better. Here, we give some arguments to the contrary, and show some experimental results which support our arguments.

Breaking the "performance wall" that we discussed in Section 1 is driving unsafe ML-based RTS, which is also being adopted, e.g., at Facebook [45] and Gradle Enterprise [7]. These early ML-based RTS adopters have access to a lot of data about code changes and test failures for training models. But, ML is out of the reach for the vast majority of *individual* open-source projects because (1) they have limited histories that contain insufficient usable data for training [72]; (2) their developers may not have ML expertise; (3) subscriptions to ML-based RTS services like Gradle Enterprise may cost more money than developers can afford (pricing for Gradle Enterprise is not public at the time of writing [6]); and (4) there are use cases in which safe RTS is critical, e.g., during debugging [29].

Next, we discuss our preliminary experiments to apply ML-based RTS on open-source projects; the results support our arguments about the lack of fit of ML-based RTS for these kinds of projects.

Models used. Trained models for ML-based RTS [7, 45] are not available publicly. So, we use publicly-available models that we previously trained for using ML-based RTS to improve analysis-based RTS [72]. That training required historical test failures, which are hard to find, so we used mutation testing to simulate failures.

Evaluation procedure. We use the 10 projects and revisions from our prior evaluation. The IDs, names, and numbers of revisions of the projects are in Table 7. We use the best model, FailBasic, the best baseline, BM25, and the selection rates that found the most failing tests. Overall, we use four models: FailBasic^E, FailBasic^S, BM25^E, and BM25^S (E: EKSTAZI, S: STARTS).
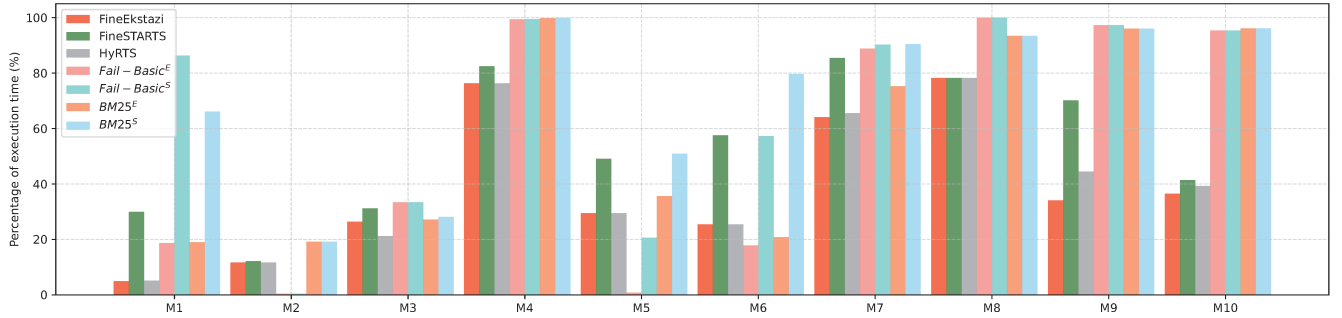
Figure 9: Percentage of execution time of RTS tools and ML models over RetestAll.

Table 7: Percentage of tests missed by ML-based RTS models, compared to intersection of tests selected by three analysis-based RTS tools—Ekstazi, HyRTS, and STARTS—[%].

PID	Name	#sha	FailBasic^E	FailBasic^S	BM25^E	BM25^S
M1	Asterisk	4	73.3	62.3	46.4	40.2
M2	Bukkit	5	56.7	28.9	46.7	8.3
M3	Config	8	21.9	17.6	25.9	21.7
M4	Csv	18	15.8	12.0	8.0	5.6
M5	Lang	36	17.7	1.9	14.4	2.1
M6	Net	17	27.2	27.2	21.2	21.2
M7	Validator	15	24.0	24.0	35.7	35.2
M8	Gedcom4j	21	11.8	11.8	5.7	5.7
M9	Vectorz	20	1.8	1.8	1.9	1.9
M10	Zt-exec	14	15.0	15.0	7.1	7.1
-	Avg	-	26.5	20.3	21.3	14.9

Results. Figure 9 compares FineEkstazi, FineSTARTS, and HyRTS, with the ML-based RTS models in terms of execution time (percentage compared to RetestAll time). We omit training costs from the comparison. Training took 56 minutes per project, on average.

FailBasic^E, the most precise ML-based RTS model, incurs more test-execution time than FineEkstazi and HyRTS in most projects. Like in our prior work. [72], we configure all four ML-based RTS tools to select a fixed fraction of tests to improve their failing test detection rate. More details are in our data package's appendix [4].

To evaluate the safety of ML-based RTS models, we compare their selection rates to the number of tests in the intersection of those that FineEkstazi, FineSTARTS, and HyRTS select. The intersection of sets of selected tests from these tools is an approximation of a "minimal" set of change-traversing tests. Table 7 shows the percentages of change-traversing tests that are not selected by each ML-based RTS model. The most precise ML-based RTS model, FailBasic^E, misses 26.5% of tests, on average. BM25^S misses the fewest tests (21.3% on average), but it is imprecise. In contrast, our enhanced RTS techniques show better precision-safety trade-offs. For example, FineEkstazi is more precise than FailBasic^E with no new safety violations (see RTSCheck evaluation in Section 5.4).

Conclusion. The experiments that we discuss here provide initial evidence that, today, analysis-based RTS performs better than ML-based RTS on open-source projects. Note that many developers may not have the expertise to train their own models, and using

models that are trained on one project may not perform well on different projects. Our re-use of ML models from prior work on RTS performs worse than FineEkstazi and FineSTARTS on projects that those models were trained on. Also, without ML, FineEkstazi and FineSTARTS obtain test reductions comparable to those obtained by using ML to improve the precision of analysis-based RTS [72].

Future work. The use of parallel computing in RTS for analyzing the dependency graph, e.g., TLDR [71], can also speed up RTS and it is orthogonal to our enhancements to RTS by leveraging the kinds of changes. We plan to explore parallelizing the analyses in FineEkstazi and FineSTARTS. Future work could also implement findings that we did not yet implement, and investigate how to use findings from our manual analysis to improve other recent RTS techniques like HyRTS, EALRTS [44], etc.

7 THREATS TO VALIDITY

The findings on which we developed FineEkstazi and FineSTARTS are based on manual inspection of changes in 250 revisions of 5 open-source projects. There are many other projects and kinds of changes. It is important to perform a more extensive study in the future. The results in this paper are limited to the projects and revisions that we evaluate, and may not generalize to other projects. But we show that even when only a few findings apply to a project, the precision and end-to-end time improvements can be beneficial.

Our process of coming up with the findings was manual. For our proposed approach—improving RTS precision by reasoning about manually identified semantics-modifying changes—to be more broadly applicable, we must develop automated techniques for examining the kinds of changes that developers make and for categorizing those into findings. We look forward to addressing these and other limitations in the future.

8 RELATED WORK

Others have used manually identified information to push the limits on other program analysis. For example, Shi et al. [60] manually identify reflection-related methods and use them to make STARTS safer. Also, Livshits et al. [43] allow users to manually provide hints for resolving reflective calls when using their static analysis. In the rest of this section, we discuss other related work on: (1) improving RTS precision, (2) leveraging code changes to improve program analysis, and (3) studies of code changes.

Improving RTS. As RTS grows in maturity and tool adoption, researchers must now start paying more attention to RTS quality improvement. We took a step in this direction with our RTSCheck methodology for testing RTS tools [75], which we now use to test FineEkstazi and FineSTARTS. We developed Reks, which improves RTS precision by not re-running tests affected by only semantics-*preserving* changes, i.e., refactoring [67]. This paper generalizes Reks and leverages semantics-*modifying* code changes that do not require re-running all tests.

Other than Reks, other work on improving RTS precision did not leverage kinds of code changes as we do. Zhang's HyRTS [73] combines method- and class-level analyses to improve RTS precision; we compared FineEkstazi with HyRTS in our evaluation. Palmskog et al. [51] formally conducted a structural hierarchical impact analysis, including coarse-grained and fine-grained components. We adopted a similar three-level hierarchy including both content and structure. Orso et al.'s DejaVOO [50] improves RTS precision with a two-phase approach: computing a class firewall [38] as an upper-bound of the set of classes affected by changed code, then using an edge-level control-flow analysis to improve the precision of the first phase. Our approach exhibits a similar refinement process: we first collect class-level RTS's selected tests, and then refine the results by integrating a method-level and field-level RTS, but we additionally incorporate the knowledge of the nature of changes in our techniques. TLDR [71] is a static method-level RTS tool and saves end-to-end time by analyzing the method-level dependency graph in parallel. TLDR is orthogonal to our work, and both approaches can be combined in the future.

Recently, companies including Meta [45] and Gradle [7] report using ML-based RTS. Researchers also studied applying ML models for RTS on open-source projects [13, 20, 44, 52, 72]. We discussed in Section 7 that ML-based RTS is not as effective for open-source projects as our enhanced analysis-based RTS.

Leveraging code changes. Bell et al.'s DeFlaker [12] leveraged code change semantics to detect flaky tests (failing tests due to non-determinism) with lower cost. DeFlaker monitors the coverage of the latest code changes, using a hybrid of class-level and statement-level dependency analysis, and marks as flaky newly failing tests that do not execute any of the changes. DeFlaker can potentially detect more flaky tests (in the same number of test runs) by leveraging our findings.

Prior work on defect prediction has studied utilizing various aspects of code change semantics such as code churn [47, 48], complexity of changes [35], and fine-grained code changes [24]. Giger et al.'s work [24] empirically studied the correlation between bugs and code change types and found that leveraging semantics of code changes can improve defect prediction models. Our work is similar, but we focus on arbitrary code changes, and our goal was to improve precision of RTS techniques.

Saha et al. [59] developed REPiR, an information-retrieval-based test-case prioritization technique that leverages code-change information. REPiR uses code changes as queries to search for relevant tests to be prioritized, which can be more computationally efficient and performs better than techniques based on program analysis.

Binkley [15, 16] used slicing to find a reduced program on which selected tests should be run and then selected only tests that exercised some statements in the reduced program, which is different but related to RTS. They focus on using operation semantics to find changed lines, but we focus on the semantics of code changes in terms of transitive dependencies between changes and tests.

Studies of code changes. Prior work has studied alternative ways to identify and categorize code changes in different contexts. Fluri et al. [23] extracted code changes as diffs of abstract syntax tree and identified popular code change types in open source projects. Nguyen et al. [49] studied popular and repetitive code change types on a large corpus of open source projects, in both within-project and cross-project settings. Martinez and Monperrus [46] mined popular code change patterns for program repair. In our work, we focus on identifying and leveraging kinds of code changes that can be used to improve RTS precision.

Ren et al. [54] developed Chianti that uses change impact analysis to determine affected tests whose execution behavior may have been modified by the change. Chianti defined a set of inter-dependent atomic changes responsible for the modified behavior of test; in contrast, we define a set of dependent atomic changes that will not result in the change of test execution behavior. We have applied our technique to improve both a dynamic and a static analysis-based RTS tool.

9 CONCLUSION

We use knowledge distilled from manual inspection of code changes to improve the precision of analysis-based RTS techniques and speed up regression testing, without using ML and without sacrificing safety. We report 13 findings from identifying changes, mostly semantic-modifying ones, in revision histories of open-source projects, and use them to enhance Ekstazi and STARTS. We implement our enhanced RTS techniques, FineEkstazi and FineSTARTS, and find that they are more precise, and that the enhancements generalize to projects that we did not manually analyze. We believe that the work presented in this paper can be a first step in a new line of work that uses semantics-modifying code changes for speeding up RTS. Doing so could help to further increase the adoption of RTS in industry.

ACKNOWLEDGMENTS

REFERENCES

[1] 2023. Apache Commons Codec. https://github.com/apache/commons-codec.
[2] 2023. Apache Commons Email. https://github.com/apache/commons-email.
[3] 2023. Apache Commons Math. https://github.com/apache/commons-math.
[4] 2023. Data package for this paper. https://github.com/EngineeringSoftware/FineRTS.
[5] 2023. Ekstazi. http://ekstazi.org/.
[6] 2023. Gradle Enterprise Pricing. https://gradle.com/pricing/.
[7] 2023. Gradle predictive test selection. https://gradle.com/gradle-enterprise-solutions/predictive-test-selection/.
[8] 2023. HyRTS. http://hyrts.org.
[9] 2023. STARTS—A tool for STAtic Regression Test Selection. https://github.com/TestingResearchIllinois/starts.
[10] Mohammed Nayef Al-Refai. 2019. Towards Model-Based Regression Test Selection. Ph. D. Dissertation. Colorado State University, USA.
[11] Thomas Ball. 1998. On the Limit of Control Flow Analysis for Regression Test Selection. In International Symposium on Software Testing and Analysis. 134–142.
[12] Jon Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically Detecting Flaky Tests. In International Conference on Software Engineering. 433–444.
[13] Antonia Bertolino, Antonio Guerriero, Breno Miranda, Roberto Pietrantuono, and Stefano Russo. 2020. Learning-to-Rank vs Ranking-to-Learn: Strategies for Regression Testing in Continuous Integration. In International Conference on Software Engineering. 1–12.
[14] John Bible, Gregg Rothermel, and David S. Rosenblum. 2001. A Comparative Study of Coarse- and Fine-Grained Safe Regression Test-Selection Techniques. ACM Transactions on Software Engineering Methodology 10, 2 (2001), 149–183.
[15] David Binkley. 1997. Semantics Guided Regression Test Cost Reduction. IEEE Transactions on Software Engineering 23, 8 (1997), 498–516.
[16] David W Binkley. 1992. Using Semantic Differencing to Reduce the Cost of Regression Testing. In International Conference on Software Maintenance. 41–50.
[17] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. 2011. Regression Test Selection Techniques: A Survey. Informatica 35, 3 (2011), 289–321.
[18] Lionel Briand, Yvan Labiche, and Siyuan He. 2009. Automating Regression Test Selection Based on UML Designs. Journal of Information and Software Technology 51, 1 (2009), 16–30.
[19] Ahmet Celik, Young Chul Lee, and Milos Gligoric. 2018. Regression Test Selection for TizenRT. In International Symposium on Foundations of Software Engineering. 845–850.
[20] Daniel Elsner, Florian Hauer, Alexander Pretschner, and Silke Reimer. 2021. Empirically Evaluating Readily Available Information for Regression Test Optimization in Continuous Integration. In International Symposium on Software Testing and Analysis. 491–504.
[21] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A Systematic Review on Regression Test Selection Techniques. Journal of Information and Software Technology 52, 1 (2010), 14–30.
[22] Emelie Engström, Mats Skoglund, and Per Runeson. 2008. Empirical Evaluations of Regression Test Selection Techniques: A Systematic Review. In International Symposium on Empirical Software Engineering and Measurement. 22–31.
[23] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change Distilling: Tree Differencing For Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering 33, 11 (2007), 725–743.
[24] Emanuel Giger, Martin Pinzger, and Harald C. Gall. 2011. Comparing Fine-Grained Source Code Changes and Code Churn for Bug Prediction. In Mining Software Repositories. 83–92.
[25] Milos Gligoric. 2015. Regression Test Selection: Theory and Practice. Ph. D. Dissertation. University of Illinois at Urbana-Champaign, USA.
[26] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Ekstazi: Lightweight Test Selection. In International Conference on Software Engineering (Tool Demonstrations Track). 713–716.
[27] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In International Symposium on Software Testing and Analysis. 211–222.
[28] Milos Gligoric, Rupak Majumdar, Rohan Sharma, Lamyaa Eloussi, and Darko Marinov. 2014. Regression Test Selection for Distributed Software Histories. In International Conference on Computer Aided Verification. 293–309.
[29] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. 2014. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In International Conference on Automated Software Engineering. 361–372.
[30] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 1998. An Empirical Study of Regression Test Selection Techniques. In International Conference on Software Engineering. 188–197.
[31] Pooja Gupta, Mark Ivey, and John Penix. 2011. Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[32] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In International Symposium on Software Reliability Engineering. 112–122.
[33] 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 Conference on Object-Oriented Programming, Systems, Languages, and Applications. 312–326.
[34] Jean Hartmann. 2012. 30 Years of Regression Testing: Past, Present and Future. In Pacific Northwest Software Quality Conference. 119–126.
[35] Ahmed E. Hassan. 2009. Predicting Faults using the Complexity of Code Changes. In International Conference on Software Engineering. 78–88.
[36] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In International Symposium on Software Testing and Analysis. 437–440.
[37] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. 1994. Change Impact Identification in Object Oriented Software Maintenance. In International Conference on Software Maintenance. 202–211.
[38] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. 1995. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. Journal of Object-Oriented Programming 8, 2 (1995), 51–65.
[39] 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 International Symposium on Foundations of Software Engineering. 583–594.
[40] Owolabi Legunsen, August Shi, and Darko Marinov. 2017. STARTS: STAtic Regression Test Selection. In International Conference on Automated Software Engineering. 949–954.
[41] Hareton KN Leung and Lee White. 1989. Insights into Regression Testing. In International Conference on Software Maintenance. 60–69.
[42] Hareton KN Leung and Lee White. 1991. A Cost Model to Compare Regression Test Strategies. In International Conference on Software Maintenance. 201–208.
[43] Benjamin Livshits, John Whaley, and Monica S Lam. 2005. Reflection Analysis for Java. In Asian Symposium on Programming Languages and Systems. 139–160.
[44] Erik Lundsten. 2019. EALRTS: A Predictive Regression Test Selection Tool. Master's thesis. KTH Royal Institute of Technology, Sweden.
[45] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. 2019. Predictive Test Selection. In International Conference on Software Engineering (Software Engineering in Practice). 91–100.
[46] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. Empirical Software Engineering Journal 20, 1 (2015), 176–205.
[47] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In International Conference on Software Engineering. 181–190.
[48] Nachiappan Nagappan and Thomas Ball. 2005. Use of Relative Code Churn Measures to Predict System Defect Density. In International Conference on Software Engineering. 284–292.
[49] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2013. A Study of Repetitiveness of Code Changes in Software Evolution. In International Conference on Automated Software Engineering. 180–190.
[50] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In International Symposium on Foundations of Software Engineering. 241–251.
[51] Karl Palmskog, Ahmet Celik, and Milos Gligoric. 2020. Practical Machine-Checked Formalization of Change-Impact Analysis. In Tools and Algorithms for the Construction and Analysis of Systems. 137–157.
[52] Rongqi Pan, Mojtaba Bagherzadeh, Taher A Ghaleb, and Lionel Briand. 2022. Test Case Selection and Prioritization using Machine Learning: A Systematic Literature Review. Empirical Software Engineering 27, 2 (2022), 1–43.
[53] Marek Parfianowicz. 2017. Open Clover. https://openclover.org.
[54] 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.
[55] Gregg Rothermel and Mary Jean Harrold. 1993. A Safe, Efficient Algorithm for Regression Test Selection. In International Conference on Software Maintenance. 358–367.
[56] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering 22, 8 (1996), 529–551.
[57] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering Methodology 6, 2 (1997), 173–210.
[58] Gregg Rothermel and Mary Jean Harrold. 1998. Empirical Studies of a Safe Regression Test Selection Technique. ACM Transactions on Software Engineering Methodology 24, 6 (1998), 401–419.

[59] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *International Conference on Software Engineering*. 268–279.

[60] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. 2019. Reflection-Aware Static Regression Test Selection. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 187:1–187:29.

[61] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-Suite Reduction and Regression Test Selection. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 237–247.

[62] August Shi, Peiyuan Zhao, and Darko Marinov. 2019. Understanding and Improving Regression Test Selection in Continuous Integration. In *International Symposium on Software Reliability Engineering*. 228–238.

[63] Min Kyung Shin, Sudipto Ghosh, and Leo R Vijayasarathy. 2022. An Empirical Comparison of Four Java-based Regression Test Selection Techniques. *Journal of Systems and Software* 186 (2022), 111174.

[64] Mats Skoglund and Per Runeson. 2005. A Case Study of the Class Firewall Regression Test Selection Technique on a Large Scale Distributed Software System. In *International Symposium on Empirical Software Engineering and Measurement*. 74–83.

[65] Mats Skoglund and Per Runeson. 2007. Improving Class Firewall Regression Test Selection by Removing the Class Firewall. *International Journal on Software Engineering and Knowledge Engineering* 17, 3 (2007), 359–378.

[66] Marko Vasic, Zuhair Parvez, Aleksandar Milicevic, and Milos Gligoric. 2017. File-Level vs. Module-Level Regression Test Selection for .NET. In *International Symposium on Foundations of Software Engineering*. 848–853.

[67] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards Refactoring-Aware Regression Test Selection. In *International Conference on Software Engineering*. 233–244.

[68] David Willmor and Suzanne M. Embury. 2005. A Safe Regression Test Selection Technique for Database Driven Applications. In *International Conference on Software Maintenance*. 421–430.

[69] Guoqing Xu and Atanas Rountev. 2007. Regression Test Selection for AspectJ Software. In *International Conference on Software Engineering*. 65–74.

[70] Nathan York. 2011. Tools for Continuous Integration at Google Scale. https://www.youtube.com/watch?v=b52aXZ2yi08.

[71] Maruf Hasan Zaber. 2021. *Towards Parallelization of Regression Test Selection*. Master's thesis. University of California, Irvine, USA.

[72] Jiyang Zhang, Yu Liu, Milos Gligoric, Owolabi Legunsen, and August Shi. 2022. Comparing and Combining Analysis-based and Learning-based Regression Test Selection. In *ICSE Workshop on Automation of Software Test*.

[73] Lingming Zhang. 2018. Hybrid Regression Test Selection. In *International Conference on Software Engineering*. 199–209.

[74] Jianjun Zhao, Tao Xie, and Nan Li. 2006. Towards Regression Test Selection for AspectJ Programs. In *Workshop on Testing Aspect-Oriented Programs*. 21–26.

[75] Chenguang Zhu, Owolabi Legunsen, August Shi, and Milos Gligoric. 2019. A Framework for Checking Regression Test Selection Tools. In *International Conference on Software Engineering*. 430–441.