Speeding up Mutation Testing via Regression Test Selection: An Extensive Study

Lingchao Chen
Computer Science Department
The University of Texas at Dallas
lxc170330@utdallas.edu

Lingming Zhang
Computer Science Department
The University of Texas at Dallas
lingming.zhang@utdallas.edu

***Abstract*—Mutation testing is one of the most powerful methodologies to evaluate the quality of test suites, and has also been demonstrated to be effective for various other testing and debugging problems, e.g., test generation, fault localization, and program repair. However, despite various mutation testing optimization techniques, mutation testing is still notoriously time-consuming. Regression Testing Selection (RTS) has been widely used to speed up regression testing. Given a new program revision, RTS techniques only select and rerun the tests that may be affected by code changes, since the other tests should have the same results as the prior revision. To date, various practical RTS tools have been developed and used in practice. Intuitively, such RTS tools may be directly used to speed up mutation testing of evolving software systems, since we can simply recollect the mutation testing results of the affected tests while directly obtaining the mutation testing results for the other tests from the prior revision. However, to our knowledge, there is no such study. Therefore, in this paper, we perform the first extensive study (using 1513 revisions of 20 real-world GitHub Java projects, totalling 83.26 Million LoC) on the effectiveness and efficiency of various RTS techniques in speeding up mutation testing. Our study results demonstrate that both file-level static and dynamic RTS can achieve precise and efficient mutation testing, providing practical guidelines for developers.**

I. Introduction

Mutation Testing [1, 2] was originally proposed to evaluate the quality of software test suites. During mutation testing, a number of *mutants*, each of which has small syntactic changes compared to the original program to simulate potential faults, will be generated based on a set of *mutation operators*, e.g., Removing Method Invocations (RMI) [3]. Then, each mutant will be executed against the test suite to check whether the test suite has different outcomes on the mutant and the original program. If the test suite can distinguish the two, the mutant is denoted as *killed*; otherwise, the mutant *survived*. Based on the correlation between mutants and real faults, if a test suite can kill more mutants, it may also detect more real faults.

Mutation testing is often considered as one of the most powerful methodologies in evaluating test suite quality [4–7]. To date, mutation testing has been used for test-suite evaluation in a large number of software testing studies in the literature [8]; mutation testing is also gaining more and more adoptions among practitioners, e.g., mutation testing has been used by developers of The Ladders, Sky, Amazon, State Farm, Norways e-voting system, and the Linux kernel [3, 9–11]. Furthermore, mutation testing has also been successfully applied for various other testing and debugging problems, e.g., real fault simulation for software-testing experimentation [12–15], automated test generation [16–18], fault localization [19–22], and program repair [23–26].

Despite the effectiveness of mutation testing, the application of mutation testing on real-world systems can still be quite challenging. One of the major challenges is that mutation testing can be extremely time consuming due to mutant execution – it requires to execute each mutant against the test suites under analysis. Researchers have proposed various techniques to speed up mutation testing, e.g., *selective mutation testing* that executes a subset of mutants to represent all the mutants [6, 27–30], *weakened mutation testing* that executes each mutant partially [31, 32], and *predictive mutation testing* that predicts mutation testing results based on some easy-to-obtain features (without actual mutant execution) [33]. Although such techniques can speed up mutation testing to some extent, they may provide rather imprecise mutation testing results.

Researchers have also proposed the *regression mutation testing* (ReMT) technique [34] to speed up mutation testing of evolving software systems while providing precise mutation testing results (i.e., producing mutation results close to the traditional un-optimized mutation testing). The basic idea of ReMT is that the high cost of mutation testing can be amortized during software evolution. Based on dynamic coverage collection and static control-flow graph reachability analysis, ReMT incrementally updates the mutation testing results for a new revision based on the mutation testing results of an old revision. ReMT has been shown to be effective in reducing mutation testing cost, and can be combined with other optimization techniques for even faster mutation testing. However, due to the fine-grained analysis and complicated design, there is no practical tool support for ReMT.

Regression Test Selection (RTS) aims to speed up regression testing via only selecting and rerunning the tests that are affected by code changes during software evolution [35–42]. A RTS technique is *safe* if it selects all the tests that may be affected by the code changes. A typical RTS technique computes test dependencies at certain granularity (e.g., method or file level) and then selects all the tests whose dependencies overlap with code changes. Various dynamic and static RTS techniques at different granularities have been proposed: while dynamic RTS techniques [35, 42] trace test dependencies dynamically

via code instrumentation, static RTS techniques [43] use static analysis to over-approximate test dependencies. To date, various mature RTS tools (e.g., Ekstazi [44], STARTS [45], and FaultTracer [46]) have been publicly available, and have been adopted by practitioners (e.g., Ekstazi has been applied to Apache Camel [47], Apache Commons Math [48], and Apache CXF [49]). Our key insight is that we can simply rerun the mutation testing results for the affected tests computed by such mature RTS tools for practical regression mutation testing. The reason is that the unaffected tests do not cover code changes, and thus may have the same mutation testing results as the prior revision. Therefore, in this paper, we perform the first extensive study on speeding up mutation testing via RTS techniques using 1513 revisions of 20 real-world GitHub Java projects, totalling 83.26 Million LoC. Since different RTS techniques may perform differently for mutation testing, we consider state-of-the-art dynamic and static, as well as file-level and method-level RTS techniques.

This paper makes the following contributions:

- **Insight** We propose the first attempt to directly apply RTS techniques for practical regression mutation testing.
- **Study**: We perform an extensive study of various practical RTS techniques (e.g., Ekstazi, STARTS, and FaultTracer) on speeding up mutation testing (using the PIT tool) of 20 open-source GitHub Java programs with 1513 revisions, totalling 83.26 Million LoC. To our knowledge, this is the largest scale study on mutation testing.
- **Findings**: We find that surprisingly both file-level static and dynamic RTS techniques can be used for precise and efficient regression mutation testing, while the dynamic method-level RTS tends to be less precise, providing important guidelines for practical mutation testing.

II. Preliminaries

This section introduces the preliminaries for mutation testing (Section II-A) and regression test selection (Section II-B).

A. Mutation Testing

Mutation testing is a fault-based testing methodology for evaluating the quality of test suites, which was firstly proposed by DeMillo et al. [1] and Hamlet [2]. Given a program $\mathcal{P}$ under analysis, mutation testing applies a set of *mutation operators* to generate a set of *mutants* $\mathcal{M}$ for $\mathcal{P}$. Each mutation operator applies a transformation rule (e.g., negating a conditional statement from `if (x>0)` to `if (x<=0)`) to generate mutants; each mutant $m \in \mathcal{M}$ is the same as the original $\mathcal{P}$ except the mutated program statement. Then, all the mutants in $\mathcal{M}$ are executed against the test suite $\mathcal{T}$ of $\mathcal{P}$ to evaluate its effectiveness – for each mutant m, when the execution of $t \in \mathcal{T}$ on m has different result from the execution of t on $\mathcal{P}$, m is killed by t; otherwise, m survives. In this way, mutation testing results can be represented as a *mutation matrix* [34]:

Definition 2.1 (Mutation Matrix): A mutation matrix is a function $\mathcal{MEM} : \mathcal{M} \times \mathcal{T} \rightarrow \{?, ❍, ✓, ✗\}$ that maps a mutant $m \in \mathcal{M}$ and a test $t \in \mathcal{T}$ to: (1) **?** if t has not been executed on m and thus the result is unknown, (2) ❍ if the execution of t cannot execute the mutated statement in m (in this case m cannot be killed by test t and does not even need to be executed against the test), (3) ✗ if t executes the mutated statement but does not kill m, and (4) ✓ if t kills m.

```
1 public class A {
2   public int m1(int a){
3     return a + 1;
4   }
5 }
6 public class B extends A{
7   public int m1(int a){
8     if (a>0){
9         return a + 2;
10    }
11    else{
12        return C.m3(a);
13    }
14  }
15  public int m2(int a){
16    return a − 1;
17  }
18 }
19 public class C{
20   public static int m3(int a){
21 - return a+2;
22 + return a+3;
23   }
24 }
```

```
1 public class T1 {
2   public void test1() {
3     A a = new A();
4     assertEquals(11,a.m1(10));
5   }
6 }
7 public class T2 {
8   public void test2() {
9     A a = new B();
10    assertTrue(a.m1(0)>=2);
11  }
12 }
13 public class T3 {
14  public void test3() {
15    A a = new B();
16    assertEquals(9,a.m2(10));
17  }
18 }
19 public class T4 {
20  public void test4() {
21    A a = new B();
22    assertEquals(12,a.m1(10));
23  }
24 }
```

Fig. 1: Example code

TABLE I: Old version mutation test result

Mutant	Statement	Mutated Statement	T1	T2	T3	T4
m1	n3	**return** a−1;	✓	❍	❍	❍
m2	n3	**return** 11;	✗	❍	❍	❍
m3	n8	**if**(a<0)	❍	✗	❍	✗
m4	n16	**return** a−2;	❍	❍	✓	❍
m5	n21	**return** (−−a)+2;	❍	✓	❍	❍

To illustrate, Figure 1 shows an example program and its corresponding test suite. The program is changed from an old revision into a new revision by modifying Class `C`. Following Definition 2.1, the mutation matrix for the old revision of the program is presented in Table I. Given such mutation testing results, developers can get feedbacks about the limitations of the existing test suites from the surviving mutants. Based on the mutation matrix, the ratio of killed mutants (i.e., *mutation score* [50]) can be easily computed and has been widely recognized as one of the most powerful methodologies for test suite evaluation. Note that in practice, *partial* mutation matrices [34], which aborts executing remaining tests against a mutant once the mutant is killed, have been widely used for computing mutation scores for sake of efficiency, since they return the same mutation scores as the original *full* mutation matrices. Besides its original application for evaluating test suite effectiveness, now it has also been successfully applied in various other testing and debugging problems, e.g., it has been applied for simulating real faults for testing experiments [12, 13, 15], guiding automated test generation [16–18], boosting fault localization [19–22], and transforming code for automated program repair [23–26].

B. Regression Test Selection

Regression test selection (RTS) [35–42] is one of the most widely used approaches to speeding up regression testing. The

main purpose of RTS is to reduce regression testing efforts by just re-running the tests affected by code changes, since the tests not affected by code changes should not change their outcomes in the new revision. In the literature, various dynamic and static RTS techniques have been proposed. Dynamic RTS techniques [35, 36, 38–40, 42] collect the test dependency information dynamically when executing tests on the old revision; then any tests whose dependencies overlap with code changes get selected. Although widely studied and used, dynamic RTS may not be suitable when dynamic test dependencies are not available, challenging to collect (e.g., code instrumentation for collecting dynamic test dependencies may cause timeouts or interrupt normal test run for real-time systems), or even unsafe (e.g., dynamic test dependencies for code with non-determinism may not cover all the possible traces, thus making RTS unsafe). Therefore, researchers have also proposed static RTS techniques [37, 41, 43, 51] to use static analysis to over-approximate the test dependencies. For both static and dynamic RTS, test dependencies and code changes can be computed at different granularities (e.g., file and method levels). Since prior work has demonstrated that file-level RTS performs the best for both static and dynamic RTS [35, 43], we introduce the basics of state-of-the-art file-level RTS techniques in the rest of this section.

1) Static RTS: STARTS [43] is state-of-the-art static file-level RTS technique based on *class firewall* analysis. STARTS computes the set of classes that might be affected by the changes and builds a "firewall" around those classes; then, any test classes within the class firewall can potentially be affected by code changes and are selected as affected tests. The notion of *firewall* analysis was first proposed by Leung et al. [52] to compute the code modules that might be affected by code changes. Then, Kung et al. [37] further proposed the notion of *class firewall* analysis by considering the characteristics of object-oriented languages, such as inheritance. STARTS performs class firewall analysis based on the Intertype Relation Graph (IRG) proposed by Orso et al. [38] to consider the specific features of the Java programming language. Formally, IRG can be defined as follows:

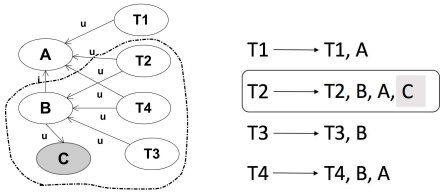
Definition 2.2 (Intertype Relation Graph): The intertype relation graph of a given program is a triple $\langle \mathcal{N}, \mathcal{E}_i, \mathcal{E}_u \rangle$ where:

- $\mathcal{N}$ is the set of nodes representing all classes or interfaces;
- $\mathcal{E}_i \subseteq \mathcal{N} \times \mathcal{N}$ is the set of inheritance edges; there exists an edge $\langle n_1, n_2 \rangle \in \mathcal{E}_i$ if type n_1 inherits from n_2, or implements the n_2 interface;
- $\mathcal{E}_u \subseteq \mathcal{N} \times \mathcal{N}$ is the set of use edges; there exists an edge $\langle n_1, n_2 \rangle \in \mathcal{E}_u$ if type n_1 uses any element of n_2 (e.g., via field accesses and method invocations).

Based on IRG, the class firewall can be computed as:

Definition 2.3 (Class Firewall): The class firewall for a set of changed types $\tau \subseteq \mathcal{N}$ is computed over the IRG $\langle \mathcal{N}, \mathcal{E}_i, \mathcal{E}_u \rangle$ using as the transitive closure: $firewall(\tau) = \tau \circ \overline{\mathcal{E}}^*$, where $\circ$ is the relational product, $\overline{\mathcal{E}}$ denotes the inverse of all use and inheritance edges, i.e., $(\mathcal{E}_i \cup \mathcal{E}_u)^{-1}$, and * denotes the reflexive and transitive closure.

To illustrate, class `C` is changed during software evolution for the example program in Figure 1. Here, we apply STARTS to select the affected tests to test the revision. Figure 2(a) shows the class firewall analysis results using STARTS. In the IRG, the use and inheritance edges are marked with labels "u" and "i", respectively. The dashed area is the class firewall analysis results, i.e., all the classes within the dashed area can potentially reach the changed class (marked in gray), and thus be affected by the changed class. In this way, Test `T2`, `T3` and `T4` are all within the class firewall, and are selected.



(a) STARTS (b) Ekstazi

Fig. 2: Static and dynamic RTS example

2) Dynamic RTS: Ekstazi [35] is state-of-the-art dynamic RTS technique based on file-level test dependencies. When executing the tests on the old program revision, Ekstazi performs on-the-fly bytecode instrumentation to record the class files used by each test. Then, given the new program revision, Ekstazi computes the changed class files via Checksum differencing, and selects any tests whose file-level dependencies involve the changed classes. Although a number of dynamic RTS techniques based on finer-granularity analysis have been proposed, they may incur large overhead due to the finer-grained analysis. The Ekstazi work [44] shows that compared with the method-level dynamic FaultTracer technique [42], Ekstazi can greatly save the end-to-end testing time, i.e., including both test execution time and RTS overhead. The Ekstazi tool now has been integrated with various build systems (including Ant, Maven, and Gradle), and has been adopted by practitioners from the Apache Software Foundation.

To illustrate, Figure 2(b) presents the RTS process using Ekstazi. Shown in the figure, for each test, Ekstazi dynamically traces the set of class files used by it in the old revision. Then, Ekstazi simply selects the tests that execute changed classes as the affected tests. In this example, only `T2` executes the changed class (marked in gray) in the old revision, and thus is selected for rerun. Note that Ekstazi may select less tests than STARTS since STARTS use static dependency information to over-approximate the potential test dependencies.

III. Studied Approach

In this section, we first present the overview of the studied approach, i.e., speeding up mutation testing via RTS (Section III-A). Then, we use examples to illustrate the studied approach (Section III-B). Finally, we present the measurements that are suitable to measure the effectiveness and efficiency of the studied approach (Section III-C).

A. Overview

Algorithm 1 presents the overview of RTS-based regression mutation testing. The inputs include two program revisions

during software evolution (i.e., $\mathcal{P}_1$ and $\mathcal{P}_2$), the regression test suite (i.e., $\mathcal{T}$), the mutation matrix collected on the old revision (denoted as $\mathcal{MEM}_1$), and the configuration argument indicating whether to collect partial mutation matrices (denoted as $Partial$). The output is the mutation matrix for the new program revision. Shown in the algorithm, Line 2 first initializes $\mathcal{MEM}_2$ as empty. Line 3 then performs RTS to select the affected tests. Note that the algorithm is general for Ekstazi, STARTS, or any other RTS techniques. Line 4 generates all the mutants for $\mathcal{P}_2$. Then, for each mutant of $\mathcal{P}_2$, the algorithm tries to collect its mutation testing results. For each test, the algorithm checks whether it is selected as affected tests. For each selected affected test, the algorithm runs the test on the mutant and store the mutant execution results in $\mathcal{MEM}_2$ (Lines 7-8). Note that most modern mutation testing tools (e.g., PIT [3], Major [53], and Javalanche [54]) will be able to skip executing a test on a mutant when the test does not cover the mutated statement of the mutant, and thus here we apply the same optimization. For each unselected test, the algorithm simply copies its mutant execution results from prior revision into $\mathcal{MEM}_2$ (Line 10). Note that Lines 11-13 are only enabled when collecting *partial* regression mutation matrix, i.e., aborting test execution for each mutant as soon as the mutant is killed. Finally, the algorithm returns the collected $\mathcal{MEM}_2$ as the resulting mutation matrix.

Note that although the algorithm is surprisingly simple, it actually supports both full and partial mutation testing. Furthermore, it handles various corner cases in practice: (1) when a test is newly added, a practical RTS tool (such as STARTS and Ekstazi) will always select it, thus its mutation testing results can be collected by the algorithm; (2) when a test is deleted during software evolution, it will not be within the current test suite $\mathcal{T}$, thus its mutation testing results will not be collected; (3) when a mutant is newly added due to code modifications, tests covering the code modifications will be selected, and thus its mutation testing results will be collected; (4) when a mutant is deleted due to code modifications, all the tests covering it will be selected, making it no longer available in the resulting mutation matrix. However, it does not mean that the algorithm always provides the same results as simply rerunning all tests on all mutants. We will provide detailed analysis in our next subsection.

Algorithm 1: RTS-based mutation testing

```
Input: Old program P1, new program P2, test suite T, old mutation matrix
       MEM1, partial matrix tag Partial
Output: New mutation matrix MEM2
1  begin
       // Initialize the new mutant execution matrix
2      MEM2 ← ∅
       // Perform RTS to selected affected tests
3      Ts ← RTS(P1, P2, T)
       // Generate mutants for P2
4      M2 ← MutGen(P2)
5      for m ∈ M2 do
6          for t ∈ T do
7              if t ∈ Ts then
                   /* Execute the test against the mutant,
                      and record results                    */
8                  MEM2 ← Execute(t, m)
9              else
                   /* Copy mutant execution results from the
                      old version                           */
10                 MEM2 ← MEM1(t, m)
11             /* Only enabled for collecting partial
                  mutant execution matrix: abort test
                  execution against a mutant once the mutant
                  is killed                                 */
12             if Partial ∧ M × T = ✓ then
13                 Break;
14     return MEM2 // Return the final mutation matrix
```

B. Analysis and Illustration

We now illustrate the RTS-based regression mutation testing using the example program in Figure 1. Note that we only illustrate the full mutation matrix collection, since the partial matrices can be collect in similar ways. The two program revisions and the test suite are already shown in Figure 1, the full mutation matrix for the old version has already been shown in Table I. The expected full mutation matrix for the new revision (via running all mutants against all tests for the new revision) is shown in Table II. Note that due to software revision, mutation testing results for mutant $m3$ have changed, while both mutant $m5$ itself and its mutation results have changed. We now show how STARTS and Ekstazi can be applied for regression mutation testing.

TABLE II: New version mutation test result

Mutant	Statement	Mutant Statement	T1	T2	T3	T4
m1	n3	**return** a−1;	✓	❍	❍	❍
m2	n3	**return** 11;	✗	❍	❍	❍
m3	n8	**if**(a<0)	❍	✗	❍	✓
m4	n16	**return** a−2;	❍	❍	✓	❍
m5	n22	**return** (−−a)+3;	❍	✗	❍	❍

Shown in Section II-B1, the STARTS technique selects tests `T2`, `T3`, and `T4`. Therefore, for each mutant of the new revision, its execution results on `T1` can be directly copied from the old revision, while its execution results on all the other tests have to be recollected. In this way, we have the resulting mutation matrix shown in Table III. Shown in the table, with STARTS, only 4 out of the 6 mutant-test cells need to be collected via mutation testing, and the mutation results for both mutants $m2$ and $m5$ (which are the only two mutants with different results in the two revisions) are correctly updated. On the contrary, shown in Section II-B2, the Ekstazi technique only selects test `T2` for the new revision. Therefore, for each mutant of the new revision, only `T2` needs

TABLE III: Regression mutation testing via STARTS

Mutant	Statement	Mutant Statement	T1	T2	T3	T4
m1	n3	**return** a−1;	✓	❍	❍	❍
m2	n3	**return** 11;	✗	❍	❍	❍
m3	n8	**if**(a<0)	❍	✗	❍	✓
m4	n16	**return** a−2;	❍	❍	✓	❍
m5	n22	**return** (−−a)+3;	❍	✗	❍	❍

TABLE IV: Regression mutation testing via Ekstazi

Mutant	Statement	Mutant Statement	T1	T2	T3	T4
m1	n3	**return** a−1;	✓	❍	❍	❍
m2	n3	**return** 11;	✗	❍	❍	❍
m3	n8	**if**(a<0)	❍	✗	❍	✗
m4	n16	**return** a−2;	❍	❍	✓	❍
m5	n22	**return** (−−a)+3;	❍	✗	❍	❍

to be executed for mutation testing. The resulting matrix is shown in Table IV. Shown in the table, with Ekstazi, only 2 out of the 6 mutant-test cells need to be recollected, even faster than regression mutation testing via STARTS. However, although Ekstazi is able to correctly update the mutation results for mutant $m5$, it fails to correctly update the mutation results for mutant $m3$, which should be killed by `T4`.

We further analyze the different performance by Ekstazi and STARTS in terms of both time savings and mutation testing precision. Shown in prior work [43], STARTS uses static analysis to over-approximate test dependencies, and selects any test that may potentially reach code changes. In terms of time savings, STARTS may perform worse than Ekstazi due to the more conservative selection (for both regression testing and mutation testing). This is also confirmed in our example, on which STARTS runs 4 mutant-test cells, while Ekstazi runs only 2. However, in terms of mutation testing precision, the conservative selection by STARTS may actually be beneficial. For two program revisions $\mathcal{P}_1$ and $\mathcal{P}_2$, if test t's dynamic dependencies (e.g., computed by Ekstazi) on $\mathcal{P}_1$ do not touch code changes between $\mathcal{P}_1$ and $\mathcal{P}_2$, t's execution on $\mathcal{P}_2$ should be the same as $\mathcal{P}_1$ (unless the code is non-deterministic), thus Ekstazi is safe for regression testing. However, t's execution on a mutant of $\mathcal{P}_1$ may be diverged (e.g., the mutant may negate a branch statement) and execute code changes. Therefore, the mutation testing results of t may change for $\mathcal{P}_2$, making Ekstazi unsafe for mutation testing. On the contrary, when test t's static dependencies (e.g., computed by STARTS) do not touch code changes, the execution of t on a mutant also may not touch code changes. The reason is that traditional mutation operators (e.g., all the mutation operators used by modern mutation tools such as PIT [3], Major [53], and Javalanche [54]) change program statements within method bodies, and usually cannot diverge program execution to the statically unreachable code. In our study, we further investigate the mutation testing precision issues in real-world systems.

C. Measurements

We now talk about the metrics to evaluate the effectiveness and efficiency of the RTS-based regression mutation testing.

1) Effectiveness: **Error Cells** In the mutation matrix, each test has an execution result on each mutant. Thus, a test and a mutant can compose a mutant-test cell. The error cell metric measures the number of cells with different execution results between the actual and the RTS-based mutation matrices. To illustrate, there is one error cell ($m3$-`T4`) between Ekstazi mutation matrix (Table IV) and the actual new mutation matrix (Table II). On the contrary, the regression mutation matrix collected by STARTS (Table III) is exactly the same as the actual mutation matrix, and does not have any error cell.

Error Mutation Score The users often use the mutation score information (e.g., the ratio of mutants killed by tests) to evaluate test effectiveness. Therefore, we also compute error mutation score as the difference in mutation scores of the actual mutation matrix and the mutation matrix collected via RTS, i.e., $|\mathcal{MS}_{actual} - \mathcal{MS}_{RTS}|$. For our example, 3 out of the 5 mutants are killed in the actual mutation matrix of the new revision, i.e., mutation score is 60%. When using both Ekstazi and STARTS, the regression mutation score is also 60%, indicating 0% error mutation score.

2) Efficiency: **Test-level Reduction** shows the ratio of tests reduced by RTS. For example, Ekstazi and STARTS select 1 and 3 out of the 4 tests, respectively; thus, the test-level reduction for Ekstazi/STARTS is 75%/25%.

Mutant-level Reduction indicates the ratio of mutants requiring at least one test execution. For example, for STARTS, 3 of the 5 mutants require at least one test execution, indicating a mutant-level reduction of 40%; for Ekstazi, 2 of the 5 mutants require at least one test execution, i.e., 60% reduction.

Cell-level Reduction measures the ratio of mutant-test cells requiring execution. Note that we only consider the mutant-test cells where the tests executes the corresponding mutated statements, since the other cells cannot be killed and do not need execution. For example, for STARTS, 4 out of 6 mutant-test cells require execution, indicating a cell-level reduction of 33.3%; for Ekstazi, 2 out of 6 mutant-test cells require execution, indicating a cell-level reduction of 66.7%.

Time Reduction All the above efficiency metrics do not consider the actual mutation testing time savings. Therefore, this metric measures the actual mutation testing time reduction.

IV. EVALUATION

A. Research Questions

We study the following Research Questions (RQs):

- **RQ1:** how do state-of-the-art static and dynamic RTS techniques perform in terms of regression mutation testing effectiveness?
- **RQ2:** how do state-of-the-art static and dynamic RTS techniques perform in terms of regression mutation testing efficiency?
- **RQ3:** how do different test dependency granularities impact the effectiveness and efficiency of RTS-based regression mutation testing?
- **RQ4:** how do state-of-the-art static and dynamic RTS techniques perform in partial regression mutation testing?

Note that we evaluate RTS in collecting full mutation matrices first, and in collecting partial matrices in RQ4.

B. Subject Systems

We used 20 real-world Java projects from GitHub as our subject systems. The selected subject systems have been widely used on regression testing or mutation testing research [10, 33, 55, 56]. Following prior work on regression test selection [43], for each project, we start from the `HEAD` revision, and obtain the 100 most recent revisions. Then, we keep all the revisions on which we can successfully run (1) `mvn test`, (2) the used PIT mutation testing tool, and (3) the studied RTS tools (e.g., Ekstazi, STARTS, and FaultTracer). Table V shows the detailed information about the used projects. In the table, Columns 1 and 2 present the project IDs and names. Column "SHA" presents the SHA for the `HEAD` revision of each project, while Column "REVS" presents the

TABLE V: Projects used in study

ID	PROJECT NAME	SHA	TESTS	REVS	KLOC
p1	HikariCP	980d8d	1922.31	92	14.89
p2	commons-io	fedbfc	99.00	74	55.43
p3	OpenTripPlanner	3cb177	138.55	86	145.81
p4	commons-functor	5d6b05	156.51	40	38.98
p5	commons-lang	013621	142.06	100	136.86
p6	commons-net	2b0f33	42.78	100	58.46
p7	commons-text	aa2a77	42.20	96	26.13
p8	commons-validator	4f60e5	69.04	97	31.12
p9	compile-testing	e4269a	9.57	73	6.07
p10	invokebinder	004d2f	3.07	100	4.31
p11	logstash-logback-encoder	4336fd	40.76	95	15.28
p12	commons-codec	1a4d9c	51.65	79	34.15
p13	commons-dbutils	633749	26.28	66	12.44
p14	commons-scxml	eac3f6	353.22	37	27.47
p15	commons-csv	ed6adc	14.18	89	10.15
p16	commons-jexl	f4babe	43.00	39	36.02
p17	la4j	db2041	39.00	39	59.64
p18	commons-cli	b486fb	24.48	96	11.28
p19	commons-math	79c471	436.48	72	316.22
p20	asterisk-java	684be6	39.00	40	81.98

number of revisions used for each project. Column "TESTS" presents the average number of tests for all the used revisions of each project. Finally, Column "kLOC" presents the average size information for all the revisions of each project. Shown in the table, the sizes of our subject systems range from 4.31K lines of the code (LoC) per revision (`invokebinder`, with 100 revisions) to 316.22K LoC per revision (`commonsmath`, with 72 revisions). In total, all the 1513 studied revisions have in total 83.26 Million LoC, and represent the largest experimental study for mutation testing to our knowledge.

C. Experimental Setup

In this study, we use the PIT [3] mutation testing tool (with all its 16 mutation operators in Version 1.1.5), since it has been demonstrated to be one of the most robust and efficient mutation testing tools for Java, and has been widely used in prior mutation-testing-related studies [10, 33, 55, 56]. The original PIT implementation aborts test execution for a mutant once it is killed and thus only supports partial mutation testing; to also support full mutation testing, following August et al. [55], we force PIT to execute each mutant against the remaining tests even after the mutant is killed. To study the effectiveness and efficiency of static/dynamic RTS tools on regression mutation testing, we apply STARTS and Ekstazi, since they have been shown to represent state-of-the-art RTS tools [35, 43]. Since both STARTS and Ekstazi are file-level RTS techniques, to further explore the impact of RTS with different test dependency granularities on regression mutation testing, we also study state-of-the-art dynamic method-level RTS technique, FaultTracer [42]. Note that we do not study the static method-level RTS since prior work has demonstrated that it is both imprecise and unsafe compared with the file-level RTS STARTS [43].

We now briefly describe our experimental setup. Our goal is to study the effectiveness and efficiency of various state-of-the-art RTS techniques on mutation testing. For each studied RTS technique on each subject system, we first obtain the SHA list of studied revisions of the subject system from GitHub. Then for every revision of the subject, we go through the following steps. (1) We make sure `mvn test` can pass all the tests. (2) We then perform full mutation testing using PIT (e.g., with command `mvn org.pitest-maven:mutationCoverage`) to get the actual mutation testing results for the revision. (3) We run the studied RTS techniques to select the affected tests for the revision (e.g., with command `mvn ekstazi:ekstazi` for Ekstazi, and command `mvn starts:starts` for STARTS). (4) We collect the regression mutation testing matrices via only rerunning the affected tests for each mutant while copying the results for other tests from the prior revision (Algorithm 1). (5) Finally, we compare our RTS-based mutation testing matrix with the actual mutation testing matrix to get effectiveness and efficiency metrics. For time reduction, we record both the RTS-based mutation testing time and the RTS overhead.

D. Result and Analysis

1) RQ1: Effectiveness: Table VI shows the effectiveness of Ekstazi and STARTS in regression mutation testing. In the table, Columns 1 and 2 present the project name and the average mutation score for all the revisions of each project. The mutation scores range from 6.77% to 84.43%, indicating that we cover a variety of test suites with different effectiveness for evaluating RTS-based mutation testing. Columns 3-5 and Columns 6-8 present the error cell and error mutation score metrics when using Ekstazi. Similarly, Columns 9-11 and Columns 12-14 present the error cell and error mutation score metric values when using STARTS. Based on the table, we have the following observations. First, surprisingly, the average error mutation score values across all projects are only 0.0423% and 0.0364% when using Ekstazi and STARTS, respectively. Similarly, the average error cell values are only 381.93 and 356.23 when using Ekstazi and STARTS, respectively. The observation demonstrates the effectiveness of using both Ekstazi and STARTS for regression mutation testing. The reason is that even using dynamic file-level RTS (such as Ekstazi), mutations that diverge test execution may not incur the test to touch a new class/file, making file-level RTS relatively safe for mutation testing. Second, we observe that STARTS has less error mutation score or error cell values than Ekstazi. As also shown in Section III-B, the reason is that STARTS relies on static test dependencies computed via over-approximation which may not be diverged to touch code changes after mutation if the original static dependencies cannot reach code changes. Finally, even STARTS can also incur imprecision in mutation score (although it is negligible), while STARTS may not have any imprecision issue according to our analysis in Section III-B. We look into the code, and find that the imprecision was due to the unsafe STARTS test selection incurred by the use of Java reflections (also observed in prior RTS study [43]). In summary, both dynamic and static file-level RTS can be used for precise regression mutation testing, and STARTS tends to be slightly more precise.

To further investigate the effectiveness of RTS-based mutation testing, we also study the effectiveness of directly copying entire mutation matrices from the previous version. Table VII

TABLE VI: Effectiveness of Ekstazi and STARTS

Project Name	Mutation Score	Ekstazi						STARTS					
		Error Cells			Error Mutation Score			Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg	min	max	avg	min	max	avg
HikariCP	23.24%	0	861	20.3	0.0000%	1.61%	0.0418%	0	197	2.16	0.0000%	0.51%	0.0056%
commons-io	32.03%	0	3479	289.32	0.0000%	0.07%	0.0052%	0	3921	449.67	0.0000%	0.09%	0.0131%
OpenTripPlanner	6.89%	0	107	8.26	0.0000%	0.14%	0.0109%	0	26	3.01	0.0000%	0.05%	0.0037%
commons-functor	6.77%	0	1	0.36	0.0000%	0.01%	0.0024%	0	1	0.36	0.0000%	0.01%	0.0024%
commons-lang	78.75%	0	6036	320.88	0.0000%	0.07%	0.0238%	0	6036	260.51	0.0000%	0.07%	0.0206%
commons-net	23.16%	0	3319	116.74	0.0000%	0.03%	0.0048%	0	3319	116.74	0.0000%	0.03%	0.0048%
commons-text	74.08%	0	322	19.47	0.0000%	0.08%	0.0106%	0	322	19.47	0.0000%	0.08%	0.0106%
commons-validator	67.73%	0	6191	97.35	0.0000%	0.00%	0.0000%	0	6191	97.39	0.0000%	0.00%	0.0000%
compile-testing	56.22%	0	704	19.19	0.0000%	0.14%	0.0038%	0	704	19.19	0.0000%	0.14%	0.0038%
invokebinder	42.65%	0	677	11.12	0.0000%	0.05%	0.0005%	0	677	16.64	0.0000%	0.05%	0.0005%
logstash-logback-encoder	54.24%	0	109	12.55	0.0000%	0.62%	0.1160%	0	192	18.21	0.0000%	0.38%	0.1108%
commons-codec	77.90%	0	262	19.71	0.0000%	0.08%	0.0170%	0	262	19.71	0.0000%	0.08%	0.0170%
commons-dbutils	46.23%	0	3398	418.51	0.0000%	0.50%	0.0276%	0	3398	391.57	0.0000%	0.50%	0.0276%
commons-scxml	44.50%	0	10275	314.22	0.0000%	0.47%	0.0558%	0	10275	301.08	0.0000%	0.47%	0.0325%
commons-csv	71.87%	0	2507	94.41	0.0000%	0.75%	0.1456%	0	2507	94.2	0.0000%	0.75%	0.1405%
commons-jexl	46.10%	0	8364	855.47	0.0000%	0.63%	0.0960%	0	8364	693.63	0.0000%	0.63%	0.0759%
la4j	57.62%	0	367	40.79	0.0000%	0.02%	0.0030%	0	17	1.54	0.0000%	0.01%	0.0007%
commons-cli	84.43%	0	2355	85.42	0.0000%	0.00%	0.0000%	0	2355	83.37	0.0000%	0.00%	0.0000%
commons-math	71.47%	265	60361	4894.31	0.0523%	0.57%	0.2800%	3	60361	4535.89	0.0026%	0.57%	0.2572%
asterisk-java	17.88%	0	2	0.28	0.0000%	0.00%	0.0000%	0	2	0.28	0.0000%	0.00%	0.0000%
Average	49.19%	13	5485	381.93	0.0026%	0.29%	0.0423%	0	5456	356.23	0.0001%	0.22%	0.0364%

TABLE VII: Effectiveness of mutation testing without RTS

Project Name	Mutation Score	Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg
HikariCP	23.24%	5	5794	731.98	0.0000%	11.42%	0.6342%
commons-io	32.03%	9	8102	2897.43	0.0030%	0.74%	0.2899%
OpenTripPlanner	6.89%	0	107	12.61	0.0000%	0.13%	0.0143%
commons-functor	6.77%	0	43	1.46	0.0000%	0.26%	0.0214%
commons-lang	78.75%	14	13160	1137.78	0.0010%	0.41%	0.0256%
commons-net	23.16%	0	6466	248.81	0.0000%	0.40%	0.0132%
commons-text	74.08%	0	2395	265.85	0.0000%	1.58%	0.1226%
commons-validator	67.73%	0	12381	439.29	0.0000%	0.53%	0.0271%
compile-testing	56.22%	0	5971	413.96	0.0000%	16.77%	0.4462%
invokebinder	42.65%	0	1352	126.87	0.0000%	16.77%	0.5993%
logstash-logback-encoder	54.24%	3	743	122.62	0.0000%	2.33%	0.2397%
commons-codec	77.90%	0	1497	107.67	0.0000%	1.07%	0.0640%
commons-dbutils	46.23%	0	5184	890.42	0.0000%	3.06%	0.1805%
commons-scxml	44.50%	0	38543	22354.63	0.0000%	1.79%	0.4094%
commons-csv	71.87%	0	5002	306.98	0.0000%	1.88%	0.1766%
commons-jexl	46.10%	1755	143337	74685.79	0.0209%	1.96%	0.6944%
la4j	57.62%	8	43602	5846.15	0.0000%	2.02%	0.2297%
commons-cli	84.43%	0	4710	365.23	0.0000%	0.52%	0.0192%
commons-math	71.47%	750	102228	7398.85	0.0016%	2.86%	0.1520%
asterisk-java	17.88%	0	170	16.77	0.0000%	0.06%	0.0083%
Average	49.19%	127	20039	5919	0.0013%	3.33%	0.2183%

shows the results for this baseline technique. Columns 1-2 present the project names and average mutation scores of all versions for each subject. Columns 3-5 show error cells of directly copying mutation matrices from the previous version. Here, the average number of error cells is 5919, much larger compared with Ekstazi (381.93) and STARTS (356.23). Columns 6-8 show the error mutation scores, and there are three projects with maximum error mutation scores of even over than 10%. When further investigating the detailed reason, we found that there are massive code changes between such versions. For example, for project *compile-testing*, the maximum error mutation score is 16.77% due to the massive changes between version *5015d6* (with mutation score of 38.15%) and version *bf6de0* (with mutation score of 54.92%). In contrast, when we apply regression mutation testing here, the error mutation score is 0% using both Ekstazi and STARTS, since all tests are affected here.

2) RQ2: Efficiency: Table VIII shows the time savings achieved by Ekstazi and STARTS during regression mutation testing. In the table, Column 1 lists all the studied projects. Columns 2-4 present the average number of tests, mutants, and mutant-test cells executed by the original full mutation testing. Columns 5-7 present the test, mutant, and cell level reduction when using Ekstazi. Similarly, Columns 8-10 present the test, mutant and cell level reduction when using STARTS. In terms of test-level reduction, Ekstazi and STARTS reduce the number of tests by 78.30% and 73.31%, respectively, indicating the effectiveness of both STARTS and Ekstazi in test selection. We also use this metric to cross validate our execution of STARTS and Ekstazi with prior RTS study, and find that our numbers are consistent with prior RTS work [43]. In terms of mutant-level reduction, Ekstazi and STARTS reduce the number of executed mutants by 86.24% and 84.79%, respectively. The difference is smaller than that of the test-level reduction. In terms of the most precise cell-level reduction, Ekstazi and STARTS reduce the number of mutant-test cell executions by 92.07% and 90.80%, respectively. Note that the cell-level reduction values are much higher than the test or mutant level reduction values. The reason is that for any unreduced mutant or test, there can still be mutant-test cells that can be reduced by RTS. In summary, both Ekstazi and STARTS can greatly reduce mutation testing costs.

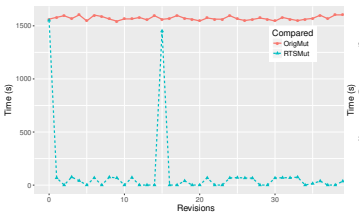


Fig. 3: `asterisk-java`

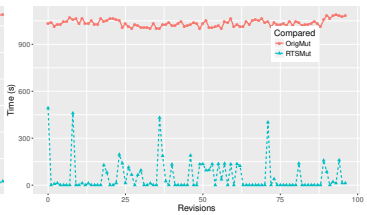


Fig. 4: `commons-validator`

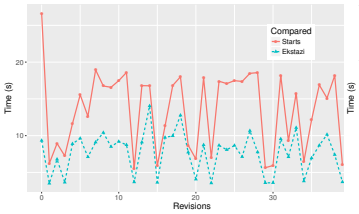


Fig. 5: `asterisk-java`

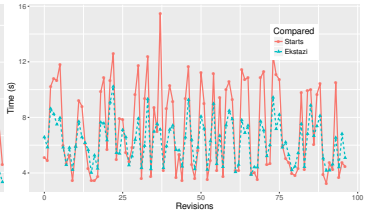


Fig. 6: `commons-validator`

We further investigate the actual time savings achieved by state-of-the-art file-level RTS. Since both Ekstazi and

TABLE VIII: Efficiency of Ekstazi and STARTS

Project Name	Results			Ekstazi			STARTS		
	Tests	Mutants	Cells	Test Level	Mutant Level	Cell Level	Test Level	Mutant Level	Cell Level
HikariCP	1922.31	5523	672782	80.23%	78.84%	74.06%	75.41%	77.86%	71.93%
commons-io	99.00	10435	791820	39.04%	80.48%	79.24%	38.56%	79.37%	78.02%
OpenTripPlanner	138.55	12679	374177	81.92%	96.93%	88.24%	53.01%	95.14%	81.29%
commons-functor	156.51	7775	105144	93.23%	99.81%	98.76%	90.62%	99.63%	97.67%
commons-lang	142.06	37353	7274165	93.08%	91.27%	97.10%	80.97%	83.79%	93.92%
commons-net	42.78	14826	1017258	95.92%	98.96%	98.65%	95.83%	98.92%	98.56%
commons-text	42.20	7518	1281577	94.94%	92.89%	98.03%	95.12%	93.21%	98.11%
commons-validator	69.04	6680	2858829	96.92%	94.97%	98.79%	96.46%	94.42%	98.79%
compile-testing	9.57	3137	677448	75.60%	80.29%	93.24%	75.42%	80.04%	93.07%
invokebinder	3.07	1546	98889	63.22%	85.04%	91.77%	64.65%	87.22%	92.96%
logstash-logback-encoder	40.76	4068	698856	93.14%	92.76%	97.14%	94.44%	92.37%	97.04%
commons-codec	51.65	10533	2762729	97.46%	98.49%	99.58%	96.46%	97.69%	99.40%
commons-dbutils	26.28	2037	1309638	90.68%	91.36%	99.02%	86.72%	89.41%	98.63%
commons-scxml	353.22	9276	8911911	27.90%	61.70%	81.41%	17.41%	59.72%	79.94%
commons-csv	14.18	1336	352518	88.23%	84.55%	95.38%	85.35%	84.18%	95.01%
commons-jexl	43.00	33077	17435853	21.79%	61.70%	80.29%	17.75%	61.25%	79.73%
la4j	39.00	13561	3703834	49.64%	54.55%	79.95%	37.15%	49.43%	77.01%
commons-cli	24.48	2767	1480348	92.03%	90.36%	97.78%	90.09%	89.03%	97.25%
commons-math	436.48	113838	47057392	94.25%	91.99%	96.83%	85.28%	86.53%	93.80%
asterisk-java	57.00	18774	716054	96.78%	97.91%	96.18%	89.41%	96.54%	93.93%
Average	185.56	15837	4979061	78.30%	86.24%	92.07%	73.31%	84.79%	90.80%

TABLE IX: Effectiveness and Efficiency of Faulttracer

Project Name	Efficiency			Effectiveness					
	Test Level	Cell Level	Mutant Level	Error Cells			Error MS		
				min	max	avg	min	max	avg
HikariCP	80.33%	78.74%	74.09%	0	8	15.13	0.0000%	1.61%	0.0510%
commons-io	42.46%	81.67%	80.55%	0	9	395.08	0.0000%	0.07%	0.0052%
OpenTripPlanner	85.99%	97.39%	90.07%	0	9	8.69	0.0000%	0.14%	0.0115%
commons-functor	93.37%	99.81%	98.76%	0	1	0.36	0.0000%	0.01%	0.0024%
commons-lang	96.99%	95.97%	98.79%	0	87	403.97	0.0000%	0.07%	0.0254%
commons-net	97.01%	99.18%	98.96%	0	99	119.93	0.0000%	0.03%	0.0048%
commons-text	95.45%	92.89%	98.05%	0	97	19.47	0.0000%	0.08%	0.0106%
commons-validator	98.24%	96.35%	99.32%	0	93	187.07	0.0000%	0.00%	0.0000%
compile-testing	91.30%	92.58%	97.60%	0	98	117.22	0.0000%	0.14%	0.0038%
invokebinder	70.29%	87.91%	93.34%	0	677	14.6	0.0000%	0.05%	0.0005%
logstash-logback-encoder	94.87%	93.99%	97.78%	0	9	17.19	0.0000%	0.38%	0.1108%
commons-codec	99.23%	99.42%	99.88%	0	9	28.18	0.0000%	0.08%	0.0172%
commons-dbutils	93.89%	93.97%	99.21%	0	86	399.98	0.0000%	0.50%	0.0276%
commons-scxml	30.24%	62.33%	81.74%	0	9	401.53	0.0000%	0.47%	0.0774%
commons-csv	93.64%	89.84%	97.79%	0	9	124.45	0.0000%	0.75%	0.1550%
commons-jexl	37.03%	63.72%	82.97%	1217	8459	1728.55	0.0000%	0.77%	0.1792%
la4j	74.03%	67.81%	87.03%	1080	914	563.03	0.0000%	0.10%	0.0111%
commons-cli	94.65%	92.62%	98.44%	0	962	97.61	0.0000%	0.00%	0.0000%
commons-math	97.50%	95.80%	98.31%	1084	8740	5123.69	0.0523%	0.89%	0.2944%
asterisk-java	99.41%	99.75%	99.64%	0	9	4.03	0.0000%	0.00%	0.0000%
Average	83.30%	89.09%	93.62%	169	1019	448.49	0.0026%	0.31%	0.0494%

STARTS achieved similar reduction, here we only present the actual time savings for Ekstazi on two example projects, `asterisk-java` and `commons-validator`. The experimental results on the other projects show similar trends. Figures 3 and 4 present the mutation testing time costs before and after using RTS. In each figure, the *x*-axis presents the number of revisions studied for the project, the *y*-axis presents the mutation testing time in seconds, the solid and dashed lines present the actual time when applying original mutation testing and Ekstazi-based mutation testing, respectively. From the figures, we can observe that Ekstazi-based regression mutation testing can significantly speed up mutation testing. For example, the original mutation testing costs 1035.22 seconds on average for `asterisk-java`, while Ekstazi-based mutation testing only costs 54.37 seconds, indicating a reduction of 94.75%. Similarly, the original mutation testing costs 1372.35 seconds on average for `commons-validator`, while Ekstazi-based mutation testing only costs 48.50 seconds, indicating a reduction of 96.47%.

In addition, we also investigate the RTS overhead. Figures 5 and 6 show the Ekstazi and STARTS overhead (i.e., end-to-end time including RTS analysis, selected test execution, and dependency collection) for `asterisk-java` and `commons-validator`. From the figure, we can observe that the cost of both static and dynamic RTS are negligible compared to the mutation testing time, e.g., the average Ekstazi and STARTS overhead for `asterisk-java` is only 7.71 seconds and 13.62 seconds, respectively.

In summary, file-level RTS techniques can significantly speed up mutation testing with negligible overhead.

3) RQ3: Impact of Test Dependency Granularity: To further study the impact of RTS with different granularity, we further apply FaultTracer, state-of-the-art dynamic method-level RTS, for regression mutation testing. We present the effectiveness and efficiency of FaultTracer-based RTS in Table IX. In the table, Column 1 lists all the projects. Columns 2-4 present the test, mutant, and cell level reductions. The remaining columns present the effectiveness metrics including both error cells and error mutation scores. From the table, we can observe

TABLE X: Effectiveness of Partial Regression Mutation

Project Name	Mutation Score	Ekstazi						STARTS					
		Error Cells			Error Mutation Score			Error Cells			Error Mutation Score		
		min	max	avg	min	max	avg	min	max	avg	min	max	avg
HikariCP	23.24%	0	104	3.56	0.0000%	1.61%	0.0418%	0	248	3.15	0.0000%	0.51%	0.0092%
commons-io	32.03%	0	17	3.26	0.0000%	0.14%	0.0071%	0	17	2.03	0.0000%	0.14%	0.0151%
OpenTripPlanner	6.89%	0	18	1.39	0.0000%	0.14%	0.0109%	0	6	0.47	0.0000%	0.05%	0.0037%
commons-functor	6.77%	0	1	0.18	0.0000%	0.01%	0.0024%	0	1	0.18	0.0000%	0.01%	0.0024%
commons-lang	78.75%	0	138	12.28	0.0000%	0.07%	0.0238%	0	27	9.64	0.0000%	0.07%	0.0206%
commons-net	23.16%	0	24	0.97	0.0000%	0.03%	0.0048%	0	24	0.97	0.0000%	0.03%	0.0048%
commons-text	74.08%	0	110	2.21	0.0000%	1.43%	0.0276%	0	110	2.21	0.0000%	1.43%	0.0276%
commons-validator	67.73%	0	2	0.44	0.0000%	0.00%	0.0000%	0	18	0.84	0.0000%	0.00%	0.0000%
compile-testing	56.22%	0	656	10.88	0.0000%	0.14%	0.0038%	0	656	10.88	0.0000%	0.14%	0.0038%
invokebinder	42.65%	0	1	0.01	0.0000%	0.05%	0.0005%	0	1	0.01	0.0000%	0.05%	0.0005%
logstash-logback-encoder	54.24%	0	100	6.46	0.0000%	0.62%	0.1160%	0	22	5.8	0.0000%	0.38%	0.1108%
commons-codec	77.90%	0	84	3.08	0.0000%	0.08%	0.0170%	0	84	3.08	0.0000%	0.08%	0.0170%
commons-dbutils	46.23%	0	258	12.22	0.0000%	0.50%	0.0276%	0	258	12.22	0.0000%	0.50%	0.0276%
commons-scxml	44.50%	0	349	121.25	0.0000%	0.96%	0.1779%	0	200	41.78	0.0000%	0.96%	0.1546%
commons-csv	71.87%	0	17	2.8	0.0000%	0.75%	0.1456%	0	17	2.66	0.0000%	0.75%	0.1405%
commons-jexl	46.10%	0	303	43.61	0.0000%	0.63%	0.0960%	0	295	32.34	0.0000%	0.63%	0.0759%
la4j	57.62%	0	26	1.1	0.0000%	0.02%	0.0030%	0	2	0.1	0.0000%	0.01%	0.0007%
commons-cli	84.43%	0	188	2.21	0.0000%	0.00%	0.0000%	0	188	2.21	0.0000%	0.00%	0.0000%
commons-math	71.47%	84	1471	422.46	0.0523%	0.57%	0.2820%	3	1471	380.32	0.0026%	0.57%	0.2572%
asterisk-java	17.88%	0	0	0.0	0.0000%	0.00%	0.0000%	0	0	0.0	0.0000%	0.00%	0.0000%
Average	49.19%	4.2	193.35	32.52	0.0026%	0.39%	0.0494%	0.15	182.25	25.54	0.0001%	0.32%	0.0436%

that although FaultTracer has a higher test-level reduction (83.30%) than Ekstazi and STARTS (consistent with prior work [44]), the cell-level reduction is actually quite close to that of Ekstazi and STARTS. This observation demonstrates that finer-grained RTS does not provide clear benefits in efficiency. Furthermore, FaultTracer incurs more severe mutation testing precision issues. For example, on average, FaultTracer incurs 448.49 error cells while Ekstazi/STARTS only incurs 381.93/356.45 error cells. The reason is that a test whose method-level dependencies do not touch code changes may very likely be diverged to cover other changed method-level entities (whereas it is harder to diverge a test execution to execute other changed file-level entities). Therefore, finer-grained RTS brings more severe mutation testing precision issues without clearly improving mutation testing efficiency.

4) RQ4: Partial regression mutation testing: In this RQ, we further investigate the effectiveness of regression mutation testing under the partial mutation testing scenario. In Table X, Columns 1 and 2 show the project names and the average mutation scores. They are the same as Table VI. Columns 3-5 present the error cells and Columns 6-8 present the error mutation scores when using Ekstazi. Similarly, Columns 9-11 and Columns 12-14 present the error cells and error mutation scores under STARTS. Based on the table, we have the following observations. First, the average error cells are 32.52 and 25.55 when using Ekstazi and STARTS. The average error cells are much fewer than those in the full regression mutation testing scenario (381.93 and 356.23 for Ekstazi and STARTS). Note that this is not because it is more accurate, but because the partial mutation testing scenario at most only has one killed test for each mutant, resulting in a much smaller total number of mutation cells than full mutation testing. Second, the average error mutation scores are only slightly higher than those in the full regression mutation testing scenario, e.g., 0.0494% and 0.0436% compared to 0.0423% and 0.0364% when using Ekstazi and STARTS, respectively. The slightly more inaccurate results here is because partial mutation only collects one killing test for each mutant, making imprecise results copied from pior versions have large impact on mutation scores. Note that overall RTS-based regression mutation testing still performs rather precisely in the partial scenario, demonstrating its effectiveness for both scenarios.

E. Threats to Validity

1) Internal: To reduce the threats to internal validity, we use state-of-the-art mature tools/techniques in our study. For example, we choose Ekstazi and STARTS to represent file-level dynamic and static RTS, and FaultTracer to represent dynamic method-level RTS. We also use state-of-the-art PIT mutation testing tool with all its 16 mutation operators. Furthermore, we also cross-validate our results with prior RTS or mutation studies.

2) External: Our experimental results might not generalize, since the projects used in our study are just a subset of all software systems and may not be representative. To reduce the threats, we used 1513 revisions of 20 real-world GitHub Java projects varying in size, application domain, number of tests, and running time. Also, all the used projects are single-module Maven projects for the ease of experimentation, and the results might be different for multi-module Maven projects. However, to our knowledge, this study already represent the largest scale study in the mutation testing literature.

3) Construct: Construct validity is mainly concerned with whether the used measurements are well designed and suitable for our study. To reduce this threat, we apply widely used effectiveness and efficiency measurements for mutation testing. For effectiveness, we use both the detailed error cell metric, and the error mutation score metric, both of which have been used in prior mutation testing work [33, 34]. For efficiency, we study the actual time reduction, as well as the reductions at the test, mutant, and cell levels.

V. RELATED WORK

A. Mutation testing

Research on speeding up mutation testing can be categorized as: selective, weakened, and optimized mutation testing.

Selective mutation testing aims to select a representative subset of mutants that can get the similar results with the original entire set of mutants [57]. Wong and Mathur [29] found that 2 mutation operators have close effectiveness with all the 22 mutation operators in Mothra [58]. To ensure the selected mutants can achieve the same results as the original whole set of mutants, Offutt el al. [28] expanded the two mutation operators into five mutation operators (which are known as the five key mutation operators). Barbaosa et al. [59] further proposed six guidelines about how to determine 10 key mutation operators. Namin et al. [27] then used variable reduction to find 28 key mutation operators. Besides operator-based mutant selection, Wong and Mathur [29] also investigated random mutation selection using Mothra. Although there is much less attention on random mutation selection, Zhang et al. [6] found that actually random selection is not inferior to operator-based mutant selection.

Weakened mutation testing provides another weaker definition of mutant killing. In traditional *strong mutation*, a mutant is determined as killed by a test if and only if the *final* test outcome differs on the mutant and the original program. Howden [31] firstly proposed the idea of *weak mutation testing*, which just checks whether the test has incurred a different *internal* program state when executing the mutant. Woodward and Halewood [32] later on further proposed the idea of *firm mutation testing*, which is a spectrum of techniques between weak and strong mutation testing.

Optimized mutation testing There are also a number of other techniques that aim to provide more efficient ways to generate, compile, and execute mutants. DeMillo et al. [60] proposed the idea of compiler-integrated mutation, which can reduce the cost of generating and compiling a large number of mutants. Later on, Untch et al. [61] proposed the idea of schema-based mutation, which can transform all mutants into one meta-mutant that can be directly compiled by a standard compiler. Researchers have also applied parallel processing techniques, including vector processors [62], SIMD [63], and MIMD [64], to speed up mutant execution. Offutt et al. [65] and Zhang et al. [66] proposed techniques to prioritize and even reduce tests to speed up mutant execution. Zhang et al. also proposed ReMT, a regression mutation testing technique that leverages program difference to reduce the cost of mutation testing. However, due to the complicated design and the costly fine-grained analysis, there is no practical tool supports for ReMT. In contrast, in this work, we firstly demonstrate that the existing off-the-shelf RTS tools can be directly applied for regression mutation testing in practice.

B. Regression Test Selection

Regression test selection (RTS) [35–42, 67] has been extensively studied for decades. Generally, RTS techniques can be divided in to dynamic and static techniques.

Dynamic RTS Rothermel and Harrold [40, 68] proposed to dynamically collect test dependencies via control-flow graph (CFG) analysis for RTS. Harrold et al. [36] then extended it to handle Java features and incomplete programs. Since it can be extremely time consuming to apply CFG analysis to large software systems, Orso et al. [38] investigated a two-phase analysis: the first partitioning phase filters out non-affected classes from an IRG, and the selection phase only performs CFG analysis on the classes kept by the first phase. Ren et al. [39] and Zhang et al. [42, 46] then applied method-level change-impact analysis based on program call graphs for RTS. Gligoric et al. [35] also proposed Ekstazi, which dynamically compute test dependencies and changes at the class/file level. While Ekstazi performs coarser-level analysis and might select more tests, it has been demonstrated to have a significantly lower end-to-end testing time due to the lightweight analysis. Recently, Zhang [67] proposed to combine coarse and fine grained analysis to further advance RTS.

Static RTS Kung et al. [37] firstly proposed the ideal of static RTS based on class firewall. Ryder and Tip [41, 51] later on investigated a static change-impact analysis technique based on 0-CFA call graphs, and demonstrated that it can be used for method-level static RTS. Badri et al. [69] further extended the change-impact analysis technique with a CFG analysis to ensure more precise impact analysis. Recently, Legunsen et al. [43] implemented the traditional class firewall analysis as a practical static RTS tool named STARTS, and performed an extensive study of STARTS on a number of real-world GitHub projects. The experimental results demonstrated that STARTS can significantly outperform static method-level RTS, and have close end-to-end testing time compared with Ekstazi, thus representing state-of-the-art static RTS.

VI. Conclusions

In this paper, we propose the idea of directly applying traditional RTS techniques for incrementally collecting mutation testing results for evolving software systems. To evaluate the effectiveness and efficiency of RTS-based mutation testing, we consider both state-of-the-art static and dynamic RTS techniques, i.e., STARTS and Ekstazi. To evaluate the impacts of RTS techniques using different test-dependency granularities, we also study the method-level dynamic RTS technique, FaultTracer. We then apply those RTS techniques to perform regression mutation testing using state-of-the-art mutation testing tool, PIT. 20 real-world GitHub projects (ranging from 4.31 KLoC to 316.22 KLoC) totalling 1513 revisions and 83.26 Million LoC of code have been used for our extensive study. The experimental results show that surprisingly both file-level static and dynamic RTS can provide rather precise and efficient regression mutation testing supports, while RTS based on finer-grained analysis tends to be imprecise. Based on our findings we encourage practitioners to apply file-level RTS (which has practical off-the-shelf tool supports) for practical regression mutation testing of evolving software systems.

VII. Acknowledgments

We thank the ICST reviewers for the valuable comments. This work is supported in part by NSF Grant No. CCF-1566589, UT Dallas start-up fund, Google, Huawei, and Samsung.

References

[1] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[2] R. G. Hamlet, "Testing programs with the aid of a compiler," *TSE*, no. 4, pp. 279–290, 1977.

[3] "PIT mutation testing system," 2017. http://pitest.org/.

[4] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[5] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for java," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 297–298, ACM, 2009.

[6] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?," in *ICSE*, vol. 1, pp. 435–444, IEEE, 2010.

[7] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs mutation testing: an experimental comparison of effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, 1997.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *TSE*, vol. 37, no. 5, pp. 649–678, 2011.

[9] I. Ahmed, C. Jensen, A. Groce, and P. E. McKenney, "Applying mutation analysis on kernel test suites: An experience report," in *ICSTW*, pp. 110–115, IEEE, 2017.

[10] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *ISSTA*, pp. 449–452, ACM, 2016.

[11] A. Groce, I. Ahmed, C. Jensen, and P. E. McKenney, "How verified is my code? falsification-driven verification (t)," in *ASE*, pp. 737–748, IEEE, 2015.

[12] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?," in *ICSE*, pp. 402–411, ACM, 2005.

[13] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *TSE*, vol. 32, no. 9, pp. 733–752, 2006.

[14] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?," in *ICSE*, pp. 535–546, 2016.

[15] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *FSE*, pp. 654–665, ACM, 2014.

[16] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *TSE*, vol. 38, no. 2, pp. 278–292, 2012.

[17] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *ISSRE*, pp. 121–130, IEEE, 2010.

[18] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. De Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *ICSME*, pp. 1–10, 2010.

[19] M. Papadakis and Y. Le Traon, "Using mutants to locate" unknown" faults," in *ICST*, pp. 691–700, IEEE, 2012.

[20] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, vol. 48, pp. 765–784, ACM, 2013.

[21] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, pp. 153–162, IEEE, 2014.

[22] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 92, 2017.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *ISSTA*, pp. 24–36, ACM, 2015.

[24] C. S. Timperley, S. Stepney, and C. Le Goues, "An investigation into the use of mutation analysis for automated program repair," in *SSBSE*, pp. 99–114, Springer, 2017.

[25] B.-C. Rothenberg and O. Grumberg, "Sound and complete mutation-based program repair," in *Proc. of FM*, pp. 593–611, Springer, 2016.

[26] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *ICST*, pp. 65–74, IEEE, 2010.

[27] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *ICSE*, pp. 351–360, ACM, 2008.

[28] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *TOSEM*, vol. 5, no. 2, pp. 99–118, 1996.

[29] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, 1995.

[30] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *ASE*, pp. 92–102, 2013.

[31] W. E. Howden, "Weak mutation testing and completeness of test sets," *TSE*, no. 4, pp. 371–379, 1982.

[32] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pp. 152–158, IEEE, 1988.

[33] J. Zhang, Z. Wang, L. Zhang, D. Hao, L. Zang, S. Cheng, and L. Zhang, "Predictive mutation testing," in *Proc. of ISSTA*, pp. 342–353, ACM, 2016.

[34] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *ISSTA*, pp. 331–341, ACM, 2012.

[35] M. Gligoric, L. Eloussi, and D. Marinov, "Practical

regression test selection with dynamic file dependencies," in *ISSTA*, pp. 211–222, ACM, 2015.

[36] 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 *ACM SIGPLAN Notices*, vol. 36, pp. 312–326, ACM, 2001.

[37] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *JOOP*, vol. 8, no. 2, pp. 51–65, 1995.

[38] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 241–251, ACM, 2004.

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

[40] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *TOSEM*, vol. 6, no. 2, pp. 173–210, 1997.

[41] B. G. Ryder and F. Tip, "Change impact analysis for object-oriented programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 46–53, ACM, 2001.

[42] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *ICSM*, pp. 23–32, IEEE, 2011.

[43] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *FSE*, pp. 583–594, ACM, 2016.

[44] "Ekstazi homepage," 2017. http://ekstazi.org/.

[45] "STARTS homepage," 2017. https://github.com/TestingResearchIllinois/starts.

[46] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: a change impact and regression fault analysis tool for evolving java programs," in *FSE*, p. 40, ACM, 2012.

[47] "Apache camel," 2017. http://camel.apache.org/.

[48] "Apache commons math," 2017. http://commons.apache.org/proper/commons-math/.

[49] "Apache cxf," 2017. http://cxf.apache.org/.

[50] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis.," tech. rep., GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUTER SCIENCE, 1979.

[51] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, and J. Dolby, "Chianti: A prototype change impact analysis tool for java," *Rutgers University, Department of Computer Science, Tech. Rep. DCS-TR-533*, 2003.

[52] H. K. Leung and L. White, "A study of integration testing and software regression at the integration level," in *ICSM*, pp. 290–301, IEEE, 1990.

[53] "Major mutation testing system," 2017. http://mutation-testing.org/.

[54] "Javalanche mutation testing system," 2017. https://github.com/david-schuler/javalanche.

[55] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *FSE*, pp. 246–256, ACM, 2014.

[56] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *ICSE*, pp. 72–82, ACM, 2014.

[57] A. P. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *COMPSAC*, pp. 604–605, IEEE, 1991.

[58] R. DeMillo and R. Martin, "The mothra software testing environment users manual," *Software Engineering Research Center, Tech. Rep*, 1987.

[59] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for c," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001.

[60] R. A. DeMillo, E. W. Krauser, and A. P. Mathur, "Compiler-integrated program mutation," in *COMPSAC*, pp. 351–356, IEEE, 1991.

[61] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ACM SIGSOFT Software Engineering Notes*, vol. 18, pp. 139–148, ACM, 1993.

[62] A. P. Mathur and E. W. Krauser, "Mutant unification for improved vectorization," *Purdue University, West Lafayette, Indiana, Technique Report SERC-TR-14-P*, 1988.

[63] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High performance software testing on simd machines," *TSE*, vol. 17, no. 5, pp. 403–423, 1991.

[64] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a mimd computer," in *ICPP*, Citeseer, 1992.

[65] J. Pan and L. T. Center, "Procedures for reducing the size of coverage-based test sets," in *ICST*, 1995.

[66] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. of ISSTA*, pp. 235–245, ACM, 2013.

[67] L. Zhang, "Hybrid regression test selection," in *ICSE*, 2018. to appear.

[68] G. Rothermel and M. J. Harrold, "A safe, efficient algorithm for regression test selection," in *ICSM*, pp. 358–367, IEEE, 1993.

[69] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: a control call graph based technique," in *APSEC*, pp. 9–pp, IEEE, 2005.