ABSTRACT
Regression testing is a widely-used approach to check that changes made to software do not break existing functionality. Unfortunately, regression test suites often grow at a rate that makes running them fully rather expensive. Researchers have proposed test-suite reduction and regression test selection as approaches to mitigate the growth of the test suite size by not running some of the tests in the test suite. However, previous research has not empirically evaluated how the two approaches compare to each other, and how well a combination of these approaches could perform.

This paper presents the first study that comprehensively compares test-suite reduction and regression test selection approaches individually. This paper is also the first to ask the question of whether the combination of the two approaches results in significant advantages. We propose a new metric to compare the quality of tests with respect to software changes. Our large-scale experiments on 15 open-source projects show that regression test selection on average runs fewer tests by 37.46pp than using test-suite reduction would. We also find that a reduced test suite can result in a high loss in fault-detection capability if considering the changes made to software. We find that a combination of the two approaches leads to greater savings in tests to run, running on average 5.77pp fewer tests than using regression test selection would. However, these tests still suffer a loss in fault-detection capability with respect to changes.

1. INTRODUCTION
Regression testing is a widely-used technique to check that changes to a software system do not break existing functionality [17, 27]. Modern software evolves fast, with changes being pushed to a repository even several times per minute [6–8]. Consequently, regression test suites grow to the point that they are too slow to run, with previous research reporting test suites taking weeks to finish [20]. A recent study from Google has shown that even for a very resourceful corporation, running all tests is prohibitively expensive, manifesting in a quadratic growth in the execution time for tests across time [23].

Previous research has proposed test-suite reduction and regression test selection as approaches to mitigate the problems associated with the blow-up of regression test suites. Test-suite reduction [27] is an approach that aims to speed up regression testing by removing tests to run. The main goal is to identify and eliminate redundant tests from a test suite. Redundancy is commonly defined with respect to some testing requirement such as statement coverage or branch coverage. Regression test selection is an approach that aims to select a subset of the original test suite that is relevant to the changes made in the software [27]. It differs from test-suite reduction as the tests selected to run are based on the changes made. Safe regression test selection aims to select all tests that may be impacted by a software change, guaranteeing that the outcome of the tests that are not selected will not be affected by the change.

When test engineers wish to speed up their test suites, it is unclear whether regression test selection or test-suite reduction is the better approach to deploy in their testing process. To the best of our knowledge, there is no prior empirical comparison of the benefits of regression test selection to test-suite reduction. Each of the two approaches comes with its own advantages. Regression test selection is a safe approach that does not fail to run tests that could detect faults related to the changes (but could fail to run tests that detect faults still in unchanged code), while test-suite reduction in general can eliminate tests that could detect faults, even though the tests are redundant with respect to the set requirements. Test-suite reduction is also an approach that is not evolution-aware, i.e., irrespective of changes removes tests that are deemed redundant. On the other hand, regression test selection needs to be rerun at every change, which incurs an additional cost every time tests are run, while test-suite reduction is usually performed less frequently and hence has a lower analysis cost over time. Practice shows, though, that the testing time in big systems is dominated by actually executing the tests, rather than running the analysis [16]. Since we aim to minimize the time tests take to run, it is unclear whether test-suite reduction or regression test selection would yield a smaller set of tests to run over time.

To find how regression test selection and test-suite reduction compare, we need to establish which approach yields smaller test suites that are also effective in the context of software evolution. Regression test selection would perform better with respect to fault-detection capability due to
changes, since regression test selection will not miss faults due to changes that test-suite reduction could miss. However, it is unclear whether, with respect to size, test-suite reduction would yield smaller test suites to run than regression test selection performed on the full test suite. In some contexts, it may be tolerable to lose some fault-detection capability in order to get faster test results to allow for a more agile development process; the full test suite could be run more rarely, when spare resources are available.

It is also unclear how to evaluate the fault-detection capability of test suites with respect to some software changes. Just measuring the fault-detection capability of the entire test suite on randomly-seeded faults could end up with too many faults unrelated to the changes and hence dilute the proportions. Also, just seeding faults in the changed code will not simulate latent faults that could be exposed due to changes. We propose a new metric that captures the quality of a test suite with respect to both latent faults and newly-introduced faults in the changed code. We use this metric to compare regression test selection and test-suite reduction and to quantify the losses in fault-detection capability that each of these approaches yields when considering software evolution.

To further speed up testing, one could even combine the two approaches; namely, one could perform test-suite reduction at some point, to create a smaller, reduced test suite, and then perform regression test selection on this reduced test suite, yielding a smaller number of tests to run than selecting from the original would. Combining the two approaches would yield the greatest speed-up in terms of the running time of the tests. We call this combination of the two approaches selection of reduction. However, it is unclear how the quality of these selected tests would be impacted by performing regression test selection after test-suite reduction. The fault-detection capability of the tests selected from the reduced test suite must be as good as the reduced test suite’s for the changed code because regression test selection must be safe with respect to the test suite it selects from. It is, however, an open question how the fault-detection capability of the tests selected by selection of reduction would compare to the tests selected by regression test selection on the original.

In this paper, we make the following contributions:

Size Comparison We evaluate the difference in the number of tests to run indicated by regression test selection versus test-suite reduction across software history.

Quality Metric with Respect to Changes We propose a metric for evaluating the quality of running tests with respect to the changes.

Comparing Quality Between Approaches We evaluate the difference in quality of tests selected by regression test selection versus test-suite reduction with respect to software changes.

Evaluating Selection of Reduction We evaluate the combination of the two approaches in terms of number of tests to run and quality of these tests.

2. METHODOLOGY

In this section, we describe our methodology for comparing the effectiveness of test-suite reduction and regression test selection. We start with some background by first defining what test-suite reduction does and introduce a precise notation for describing the reduced test suite and its quality.

We then describe how we simulate the evolution of a reduced test suite across multiple software versions to account for software changes. We describe what regression test selection does and introduce some notation for it as well. Based on this, we next describe our contributions. We define our novel metric that compares the loss in quality of the reduced test suite with change-related focus at a version to properly compare test-suite reduction with regression test selection. We finally describe our proposed combination of the two approaches where regression test selection is performed on top of test-suite reduction.

2.1 Test-Suite Reduction

Test-suite reduction seeks to remove redundant tests from a test suite with respect to some testing requirements. Requirements are commonly defined by coverage criteria, such as statement coverage or branch coverage, where the coverage criteria is a proxy for the ability to detect faults. If a test satisfies only the requirements already satisfied by some other tests in the test suite, then that test is considered redundant with respect to that test suite. For a set of tests \( T \), we write \( \text{req}(T) \) for the set of requirements satisfied by \( T \); note that \( \text{req}(T) \) gives a set that is generic in the requirements it could contain. Then, for the original, full test suite \( O \), test-suite reduction constructs a reduced test suite \( R \) such that \( R \subseteq O \) and \( \text{req}(O) = \text{req}(R) \).

Previous research has focused on developing test-suite reduction techniques that construct a reduced test suite from the full test suite, such that removing any one test would cause the reduced test suite to satisfy fewer requirements than the full test suite, but such a reduced test suite would not necessarily be the minimum-sized reduced test suite. The effectiveness of a reduced test suite is measured by the reduction of size and the loss in fault-detection capability in comparison to the full test-suite size and capability. For a reduced test suite \( R \) constructed from the full test suite \( O \), the size benefit of reduction is measured by \( 100 \times \frac{|R|}{|O|} \); a smaller value is better as it means less tests need to be run. Loss in fault-detection capability is measured with respect to some metric that is a proxy for the number of faults detected. We can again use \( \text{req}(T) \) to denote the set of faults detected by the set of tests \( T \) (as faults can be seen as requirements in a broad sense). The loss in fault-detection capability is then measured by \( 100 \times \frac{|\text{req}(O) \setminus \text{req}(R)|}{|\text{req}(O)|} \), where a smaller value is better, denoting that the number of faults detected by the full test suite and not detected by the reduced test suite is small relative to the set of all detected faults. Clearly, if the requirements used here for evaluation are the same as the ones used to perform reduction in the first place, there would be no loss in fault-detection capability by construction. Therefore, it is necessary to use a different set of requirements in the evaluation.

2.2 Evolving Reduced Test Suite

Test-suite reduction constructs a reduced test suite on a single version, so to evaluate how effective a reduced test suite is as software evolves, it is necessary to evolve the reduced test suite into each subsequent version. In general, one can imagine a procedure that takes a test suite from one version of software and outputs the “evolved” test suite at a later version. In our previous work [21], we considered the evolution of the reduced test suite into a future version by tracking tests based on name and only considering the tests
that existed across all versions while ignoring the tests not present in the versions subsequent to the reduction point. The advantage of that approach is that it focuses on how the existing tests from the reduced test suite behave in the future, but the disadvantage is that it ignores new tests.

We modify our previous approach to evolving the reduced test suite by also considering the tests that are added to the test suite. We define $R_i$ to be the reduced test suite constructed at version $i$ from the full test suite $O_i$. When $R_i$ evolves to a later version $i'$, we consider any new and removed tests between $O_i$ and $O_{i'}$. The new tests are the ones that are in the subsequent version $O_{i'}$ while not in the previous version. Further, we define the removed tests to be those in $O_i$, but not in $O_{i'}$. (A more precise approach could track tests not only by name but by the semantics, inferring likely renames [11], but test renames are infrequent.)

**Definition 1.** The reduced test suite computed at version $i$ and evolved to subsequent version $i'$ is the evolved reduced test suite:

$$E_{i,i'} = (E_{i,i} \cup (O_{i'} \setminus O_i)) \cap O_{i'}$$

Note that $E_{i,i} = R_i$. An $E_{i,i'}$ can be constructed iteratively by computing $E_{i,i+1} = E_{i,i} \cup (E_{i+1,i} \cap O_{i+1})$ and so on until $E_{i,i'}$.

### 2.3 Regression Test Selection

Unlike test-suite reduction techniques that consider only one software version, regression test selection techniques consider changes between two software versions. The aim of regression test selection is to only run the tests that may be affected by the changes made since the last test run. Regression test selection techniques aim to be safe with respect to the changes, i.e., they will not select a test to be run only if the software changes cannot affect the outcome of the test [17].

Given a full test suite at version $i$, $O_i$, regression test selection has to select a subset of these tests $S_{i,\delta}$ to run. We use $\delta$ to indicate all the other inputs that regression test selection takes, including most importantly the changes between version $i$ and the previous version (typically version $i - 1$ in the case of a non-distributed version control system [13]) but also the coverage matrix from the previous version. The effectiveness of regression test selection is measured by the ratio of number of tests selected to run to the total number of tests, $100 \times \frac{|S_{i,\delta}|}{|T_i|}$, where the smaller number of tests to run the better.

### 2.4 Comparing Quality of Tests Run

![Figure 1: Matrix for requirements($R$) satisfied by a test ($T$). Requirements in red are affected by changes. Tests in red are selected by regression test selection. Tests in brackets are in the evolved reduced test suite.](image)

Our goal is to compare the subsets of tests run by regression test selection and test-suite reduction, which requires an appropriate metric. As a simple example, consider a relationship between tests and what requirements they satisfy in Figure 1. In this example, only $R3$ is affected by the change from a previous version, and $S_{i,\delta}$ contains the tests $T1$, $T2$, and $T5$. We see that $E_{i,i'}$, which consists of tests $T4$ and $T5$, satisfy requirements $R1$, $R2$, $R3$, and $R5$. In the context of test-suite reduction, the loss in quality of a reduced test suite is typically measured by $100 \times \frac{|req(T)\setminus req(T_i)|}{|req(T)|}$, where we use $req(T)$ in the broadest sense to denote any requirements, e.g., a set of statements covered or a set of (seeded or real) faults detected. This metric includes requirements related to the code in the entire software, even in the parts unchanged from the last version. Because regression test selection aims to select only tests affected by the changed code, the selected tests need not satisfy requirements not related to the changes. In this example, if we use this traditional metric, we see that both $S_{i,\delta}$ and $E_{i,i'}$ fail to satisfy two of these requirements. However, as $R3$ is the only requirement affected by the change, it is only necessary for tests to satisfy this requirement if we are concerned with only the changes. Therefore, we cannot directly compare the quality of a reduced test suite to the tests selected by regression test selection using this traditional metric. Instead of using the requirements satisfied by the full test suite as the baseline for comparison, we need a new baseline that encompasses just the changes between two versions. We call such requirements as being with change-related focus.

If regression test selection is safe, then the requirements satisfied by the tests selected by regression test selection to run, $S_{i,\delta}$, will encompass all of the requirements with change-related focus that occur between the two versions. Therefore, we would want to see how a reduced test suite is able to satisfy the same requirements that $S_{i,\delta}$ satisfies, namely $req(S_{i,\delta})$.

\[\text{ReqChanged}(T, S_{i,\delta}) = \text{req}(S_{i,\delta} \cap T) \setminus \text{req}(T \setminus S_{i,\delta})\]

We use $\text{ReqChanged}(O_i, S_{i,\delta})$ to denote the requirements with change-related focus that $S_{i,\delta}$ satisfies. The tests in $O_i \setminus S_{i,\delta}$ are the tests not selected by regression test selection, so they are determined to not be affected by the changes. Any requirements satisfied by the non-selected tests cannot be with change-related focus. If these requirements were related to the change, then the tests in $O_i \setminus S_{i,\delta}$...
that satisfy these requirements would have been selected and placed in \( S_{\delta} \). By removing these requirements from consideration, we can check if the tests in the evolved reduced test suite satisfy the remaining change-related requirements. We can use \( \text{ReqChanged}(E_{i,i'}, S_{\delta,i}) \) as the set of requirements that an evolved reduced test suite \( E_{i,i'} \) (where the original reduced test suite was constructed at earlier version \( i \)) must satisfy to be at the same quality as \( S_{\delta,i} \) with respect to the change \( \delta \).

**Definition 3.** Given the current full test suite \( O_i \), the tests selected to run \( S_{\delta,i} \) due to change \( \delta \), and the evolved reduced test suite \( E_{i,i'} \), the loss in quality of \( E_{i,i'} \) with respect to \( \delta \) is:

\[
\text{ReqLoss}_{i,i',\delta} = 100 \times \frac{|\text{ReqChanged}(O_i, S_{\delta,i})|}{|\text{ReqChanged}(O_i, S_{\delta,i})| - |\text{ReqChanged}(E_{i,i'}, S_{\delta,i})|}
\]

In this metric, we compute the requirements with change-related focus for \( E_{i,i'} \) using \( S_{\delta,i} \). We then find out all requirements with change-related focus from the full test suite, how many are satisfied by the related test suite. Our baseline for comparison purposes is the set of requirements with change-related focus the full test suite will satisfy, which we compute by Definition 2.

Using \( \text{ReqChanged}(O_i, S_{\delta,i}) \) in our example, we filter out the requirements not with change-related focus by removing the requirements satisfied by the tests not selected, namely T3 and T4. Through this, we see that \( \text{ReqChanged}(O_i, S_{\delta,i}) \) now no longer includes R4 and only contains the affected requirement R3. Then, using \( \text{ReqLoss}_{i,i',\delta} \) to compare the loss in quality between \( S_{\delta,i} \) and \( E_{i,i'} \), we can see that \( E_{i,i'} \) is indeed of the same quality as \( S_{\delta,i} \) with respect to the changes.

**2.5 Selection of Reduction**

We compare the effectiveness of test-suite reduction and regression test selection with each other because they both run a subset of tests from the full test suite to save testing time. However, the two approaches are orthogonal: test-suite reduction removes redundant tests from a single version, while regression test selection considers changes between two versions to select tests to run. The two approaches can be combined together. After test-suite reduction constructs \( \mathcal{R}_i \) for version \( i \), \( \mathcal{R}_i \) can be used as a replacement for \( O_i \), and its evolved version \( E_{i,i'} \) can be used for all future versions \( i' \) in lieu of \( O_i \). Regression test selection can then select tests affected by the changes between versions \( i' \) and the version before it from \( E_{i,i'} \) instead of \( O_i \). In fact, we imagine in actual software development that if test-suite reduction is used to construct a reduced test suite, developers would want to still run regression test selection just to see the impact of the changes they make. By combining the two approaches, we can obtain even more savings in what tests need to be run. Fundamentally, to get selection of reduction, a regression test selection technique would need to select tests from a reduced test suite. However, For a version \( i' \), given \( S_{\delta,i} \) and \( E_{i,i'} \), we can simulate the tests selected from the evolved reduced test suite \( \mathcal{S}\mathcal{R}_{i,i'} \) as \( S_{\delta,i} \cap E_{i,i'} \).

**3. Evaluation**

This section describes our evaluation of test-suite reduction, regression test selection, and their combination on 15 open-source projects from GitHub [2]. We first summarize the projects used in our study and then describe how we set up our experiments to answer the following three research questions:

- **RQ1:** Does regression test selection or test-suite reduction yield smaller test suites on average?
- **RQ2:** What is the loss in quality, with change-related focus, of the reduced test suite?
- **RQ3:** How does selection of reduction compare with the other two approaches?

### 3.1 Projects

Figure 2 lists the 15 projects that we use in our evaluation. All these projects were used in our prior study of test-suite reduction [21]. We selected Java projects from GitHub that build using Maven [3] and use JUnit [14] to run tests. In this study, we do not use three of the projects from the prior study: one project does not build any more, and two projects do not work with the Ekstazi regression test selection tool.

For each project, we tabulate the number of commits used in our evaluation; the starting SHA (the commit ID in Git); the minimum, median, and maximum number of lines of code among the commits used in the evaluation (measured using SLOCCount [5]); and the minimum, median, and maximum number of tests in the full test suite among the commits used in the evaluation.

### 3.2 Experimental Setup

For each project, we perform regression test selection across the range of commits specified in Figure 2. We use a state-of-the-art regression test selection tool Ekstazi [1] to find what tests are selected by the changes between each commit and its predecessor commit. Ekstazi selects the tests based on file-level dependencies, i.e., if a file changed between two commits, Ekstazi selects every test that depends on that file. Moreover, Ekstazi selects at the level of test classes, i.e., even any one test from a test class depends on a changed file, Ekstazi the entire test class to run (i.e., all of its tests).

To measure test quality, we use covered statements and killed mutants as the requirements that the tests must satisfy. We use the mutation testing tool PIT [4] to map each test from the full test suite to the statements covered and killed mutants by the test. Due to the high cost of mutation testing, we do not collect these mappings at every commit but only at every 30 commits, starting from the starting commit specified in Figure 2 for each project.

At every point where we record the satisfied requirements, we also perform test-suite reduction to generate a reduced test suite at that point. We generated the reduced test suite by applying the Grecy algorithm [10] on the statements covered. From each reduced test suite, we then create an evolved reduced test suite in each subsequent commit for all of the commits we consider; to perform this evolution, we use the methodology described in Section 2.2.

### 3.3 RQ1: Comparing Number of Tests to Run

To answer RQ1, we measure how many tests would be run if we used test-suite reduction, regression test selection, or their combination across the range of commits for each project. To visualize how the number of tests changes from commit to commit, we plot the number in the form of a line plot. Figure 3 illustrates these plots for three of the projects.
<table>
<thead>
<tr>
<th>Project ID</th>
<th>Project</th>
<th>Revs</th>
<th>Start Sha</th>
<th>LOC</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Min</td>
<td>Median</td>
</tr>
<tr>
<td>P1</td>
<td>Commons-Lang</td>
<td>509</td>
<td>0ce1a4db</td>
<td>6098</td>
<td>63553</td>
</tr>
<tr>
<td>P2</td>
<td>Caelum Stella</td>
<td>309</td>
<td>39e50b7f</td>
<td>18730</td>
<td>26727</td>
</tr>
<tr>
<td>P3</td>
<td>Caelum Vraptor</td>
<td>165</td>
<td>443c0ed</td>
<td>29559</td>
<td>31552</td>
</tr>
<tr>
<td>P4</td>
<td>Cloudfoundry</td>
<td>209</td>
<td>76cbb8c</td>
<td>14587</td>
<td>18930</td>
</tr>
<tr>
<td>P5</td>
<td>Dropwizard</td>
<td>270</td>
<td>6b664144</td>
<td>11074</td>
<td>11759</td>
</tr>
<tr>
<td>P6</td>
<td>Scribe-Java</td>
<td>214</td>
<td>022280fe</td>
<td>2113</td>
<td>3039</td>
</tr>
<tr>
<td>P7</td>
<td>SQL-Parser</td>
<td>251</td>
<td>a1dd59b</td>
<td>13262</td>
<td>16699</td>
</tr>
<tr>
<td>P8</td>
<td>JodaTime</td>
<td>256</td>
<td>39c7fa7f</td>
<td>80660</td>
<td>82371</td>
</tr>
<tr>
<td>P9</td>
<td>AssertJ-Core</td>
<td>348</td>
<td>df1adedd</td>
<td>55073</td>
<td>63693</td>
</tr>
<tr>
<td>P10</td>
<td>MessagePack</td>
<td>340</td>
<td>39c7fa7f</td>
<td>3494</td>
<td>31881</td>
</tr>
<tr>
<td>P11</td>
<td>JOPT-Simple</td>
<td>153</td>
<td>e4c251d6</td>
<td>6264</td>
<td>7380</td>
</tr>
<tr>
<td>P12</td>
<td>SLF4J</td>
<td>335</td>
<td>cfd6dbfa</td>
<td>9986</td>
<td>11152</td>
</tr>
<tr>
<td>P13</td>
<td>Jasmine</td>
<td>180</td>
<td>8d9121ab</td>
<td>1268</td>
<td>3742</td>
</tr>
<tr>
<td>P14</td>
<td>Square Wire</td>
<td>181</td>
<td>3cad6d0c</td>
<td>6040</td>
<td>11625</td>
</tr>
<tr>
<td>P15</td>
<td>LA4J</td>
<td>437</td>
<td>7bd10910</td>
<td>7724</td>
<td>11990</td>
</tr>
</tbody>
</table>

Figure 2: Statistics of projects used in our experiments

![Commons-Lang reduced starting from 0ce1a4db](a)
![JodaTime reduced starting from 07002501](b)
![LA4J reduced starting from 7bd10910](c)
![LA4J reduced starting from c8e61571](d)

Figure 3: Tests to run across multiple versions for select projects

We chose to display the plots for these three projects because they show interesting varying behaviors. For Commons-Lang, regression test selection is relatively small, and for most commits, the number of tests selected by regression test selection from the evolved reduced test suite is smaller than the number of tests in the evolved reduced test suite. However, for JodaTime, we see many

we evaluated on: Commons-Lang, JodaTime, and LA4J. In each of the plots, the lines show the number of tests to be run by the different approaches at each commit. The green line (at the top) represents the total number of tests for each commit, the red line (in the middle) represents the number of tests in the evolved reduced test suite (where test-suite reduction is applied at the starting SHA specified in the caption), the blue line (with bigger “zig-zags”) represents the number of tests selected by regression test selection, and the orange line (with smaller “zig-zags”) represents the number of tests selected by regression test selection from the evolved reduced test suite.
commits throughout the history where regression test selection would have selected many more tests than the evolved reduced test suite. Commons-Lang and JodaTime are both rather mature, and the growth of the test suite seems relatively small and stable, i.e., the number of tests in the full test suite does not change substantially across the commits.

In contrast, for LA4J, the test suite changes greatly as the software evolves. Because of that, we show two line plots for LA4J, which also highlight how we apply test-suite reduction at various points in our evaluation. In Figure 3c, we apply test-suite reduction on the earliest commit and evolve the reduced test suite from there across the entire range of commits used in the evaluation. In Figure 3d, we use the same project but apply test-suite reduction on a later commit and evolve it from there until the end of the range. In both cases we see that regression test selection frequently selects to run more tests than the evolved reduced test suite. We also see that reapplying test-suite reduction on a later commit reduces the number of tests that test-suite reduction has to run. For example, from the starting point shown in Figure 3d, the evolved reduced test suite in Figure 3c had grown to 486 tests, whereas rerunning test-suite reduction at that point drops the reduced test suite to 115 tests.

We do not show detailed line plots for the other projects and starting points, because they are similar in shape to the four line plots shown and would take a lot of space, but we instead summarize their results. For each project and each starting point, we collect for each commit the ratio of the number of tests to be run by each approach over the number of tests in the full test suite in that commit. Figure 4a shows the distribution of these ratios in the form of violin plots. For each project, we show three violin plots, one for each approach: the red • (leftmost) violin plot is for the evolved reduced test suite, the blue • (middle) violin plot is for the tests selected by regression test selection, and the orange • (rightmost) violin plot is for the tests selected by applying regression test selection on the evolved reduced test suite. Each violin plot shows the minimum, median (horizontal line), mean (black dot) and maximum values, and the width of the plot is determined by the number of points for each value.

We find that, across the range of commits we evaluated for each project, using test-suite reduction would have, on average, run more tests than using regression test selection. (As expected, the number of tests selected to run by combining the two approaches is even smaller than the number of tests to run by either of the approaches.) In particular, the median ratio of tests to run for evolved reduced test suite is higher than the median ratio of tests to run for regression test selection for all projects but two (LA4J and SQL-Parser). In other words, the difference in the median ratio of tests to run for evolved reduced test suite and the median ratio of tests to run for regression test selection is positive for all projects but two. The mean of these differences shows that the evolved reduced test suite size is higher than the number of tests selected by regression test selection. However, we note that the number of tests selected to run by regression test selection across the commits has a much wider distribution than test-suite reduction, i.e., one cannot easily predict how many tests are run when using regression test selection.

We also note that regression test selection relatively often selects to run no test (because no relevant change was made between commits that would affect any of the tests). It could be that the average values for regression test selection are skewed by these zero cases, and in theory, a developer may realize that no relevant change is made and could manually choose to not run any tests even from a reduced test suite. To explore this further, we plot in Figure 4b the number of tests selected to run by regression test selection only for the commits when a non-zero number of tests is selected. We find that there are now four projects (JOPT-Simple and MessagePack in addition to LA4J and SQL-Parser) where regression test selection on average selects to run more tests than the evolved reduced test suite. Also, the median evolved reduced test suite size is now on average only 19.55pp higher (but still higher) than the median number of tests selected to run by regression test selection across all projects.

Finally, performing the Wilcoxon signed-rank test to compare the distributions of the numbers of test selected to run by regression test selection and test-suite reduction yields p-values that are significant with alpha-values of 0.001, indicating that there is a very high probability that the two distributions are different.

RQ1: In sum, for all scenarios we considered, regression test selection selects to run, on average, fewer tests than the evolved reduced test suite.

This is a surprising result, because regression test selection aims to be safe and selects all tests that could be affected by the changes (and potentially selects many more tests given the coarse, imprecise level of classes at which Ekstazi operates), whereas test-suite reduction is unsafe and could miss some test that is affected by the changes.

3.4 RQ2: Comparing Quality of Testing

To evaluate how much test-suite reduction could miss, and to answer RQ2, we measure the quality of the tests to run based on two metrics—statement coverage and killed mutants. We measure this loss in quality at each point where we collected the mapping from the tests to the satisfied requirements, by default every 30 commits. For cases where regression test selection did not select to run any tests (e.g., between commits 29 and 30), we went back one commit at a time (to 28, 27, and so on) and added in any tests selected by regression test selection due to the changes in earlier commits until the denominator in Definition 2 becomes non-zero. This process simulates having had a bigger change happen (i.e., in the Git terminology, as if the developers squashed several commits), because the tests selected by regression test selection reflect the changes between a wider range of commits.

Before we summarize the results for quality loss across all of the projects and various commit points, we show more detailed results for one project. Figure 5 shows, for the LA4J project, how the statement coverage and killed mutants of reduced test suites compare to the corresponding metrics for the test suites selected by regression test selection with change-related focus, as measured by Definition 3. Each column shows, for the specified commit (every 30 commits from the beginning), the percentage of statements cov-
Figure 4: Violin plots showing distribution of sizes for reduction, selection, and selection of reduction

(a) Sizes for all commits (including zeroes for selection)

(b) Sizes for commits where selection was not zero

Figure 5: Loss in quality of evolved reduced test suite evaluated with change-related focus.

For LA4J, distance between row/column is about 30 commits.

Figure 6: Violin plots showing distribution of loss in quality due to reduction for different projects

(a) Statement coverage

(b) Killed mutants

We do not show detailed tables for the other projects and starting points, but we instead summarize their results. Figures 6a and 6b show, for each project, a lavender (leftmost) violin plot representing the distribution of the loss in quality, with change-related focus, of statement coverage and killed mutants, respectively. To provide a way to contrast our proposed metric for loss in quality with change-related focus, we also show for each project a magenta (rightmost) violin plot for the traditional metric of loss for the evolved reduced test suite, where the loss is measured with respect to all the requirements satisfied by the full test...
We see that the loss in statement coverage with change-related focus typically has low absolute values, with the median for a project being at most 2.71% (said project being JOPT-Simple). The values for the traditional metric are also fairly similar, but recall that loss with change-related focus is the metric that matters for the evolving code. In contrast, the loss in killed mutants has higher absolute values across all projects, with the median value of loss being as high as 5.85% for a project (said project again being JOPT-Simple). These higher percentages mean that, after applying test-suite reduction, the resulting evolved reduced test suite at each commit can miss detecting many faults related to the changed code. This is the peril of using test-suite reduction. Compared to the full test suite, test-suite reduction makes the test-suite smaller but can miss some faults that the full test suite can find. This was well understood from previous studies [18, 19, 22, 24, 25, 27, 31], but we are concerned with change-related focus and comparing with regression test selection.

**RQ2:** Compared to regression test selection, test-suite reduction has on average larger test suites and can still miss some faults that the selected tests find with change-related focus. In contrast, (safe) regression test selection finds all the faults that the full test suite can find due to the code changes, i.e., it has zero loss with change-related focus.

### 3.5 RQ3: Evaluating Selection of Reduction

Despite the perils of test-suite reduction, test engineers may still choose to use it for faster regression testing; one can then make regression testing even faster by applying regression test selection on the reduced test suite. To answer RQ3, we evaluate how effective the combined selection of reduction is compared to individual test-suite reduction and regression test selection. As a comparison metric, we use the number of tests selection of reduction selects to run at each commit for a project. Note that we do not need to empirically evaluate the loss in fault-detection capability of selection of reduction: because selection of reduction selects from the reduced test suite those tests that are affected by the changes, assuming a safe regression test selection, the loss of selection of reduction with change-related focus is as good as the reduced test suite.

In Figure 4, we already plotted the violin plots for the number of tests selected by selection of reduction, which makes it easy to compare the numbers for selection of reduction to the numbers for the other two approaches. The distribution across all projects shows that, as expected, selection of reduction always selects fewer tests to run than any of the other two approaches. However, as selection of reduction performs selection on top of the reduced test suite, it is not known how the ratio of tests selected by regression test selection from the full test suite compares to the ratio of tests selected by selection of reduction from the reduced test suite. Figure 7a shows the distributions of these two ratios through violin plots for each project. As before, we also consider the case that regression test selection often selects no tests to run, so we also plot only the non-zero numbers of tests selected by both regression test selection and selection of reduction in Figure 7b.

Utilizing the Wilcoxon signed rank test to compare the ratio of tests selected from the original test suite to the ratio of tests selected from the reduced test suite yields p-values significant with an alpha-value of 0.001 for all projects except uaa, which would have been significant with a significance level of 0.1. Therefore, there is a very high probability that the distributions of the numbers of tests selected to run by regression test selection from the original test suite and that of tests selected to run by regression test selection from the reduced test suite are different.

Note, however, that the mean of the difference between median ratio of the number of tests selected by regression test selection from the original test suite and the median ratio of the number of tests selected by regression test selection from the reduced test suite for each project is 0.64pp. Therefore, although the Wilcoxon statistical test establishes a high probability that the distributions of these ratios for each project is different, their medians do not differ greatly.

**RQ3:** We find that, in general, the ratio of tests selected from the evolved reduced test suite is about the same as the ratio of tests selected from the full test suite.

### 4. DISCUSSION

**Requirements affected by changes** We define requirements that are affected by changes by contrasting the requirements satisfied by tests selected by regression test selection and the requirements satisfied by tests that are not selected. We could have narrowed down the requirements by looking directly at the changed code between two commits, in our case looking at just the lines changed and considering only those statements and killed mutants on those lines. However, our metric can actually be somewhat stronger than considering just requirements directly on changed lines of code. Consider a mutant that is in some unchanged part of code. If we were to restrict the considered mutants to be only on changed lines of code, this mutant would be ignored. However, it is entirely possible that a change elsewhere in the code could trigger the detection of this mutant in a test. For instance, if the change in the code is to a branch condition and it is through this change that mutant is reached and killed, our metric would classify this mutant as one that is affected by the change. If only mutants on changed lines are classified as those that are affected by change, this mutant would not have been considered, which would be incorrect.

**Why don’t we do reduction of selected?** One could consider, for academic purposes, performing reduction over selected tests, and this could yield different results than selection over the reduced test suite. This process is impractical, since in order to perform reduction, one would need to construct a requirements matrix for the current version in order to eliminate redundant tests. However, such a matrix is usually constructed by running the tests; hence, in order to compute the reduction of the selected tests, one would need first to run the selected tests to compute a requirements matrix, which would defeat the whole purpose of combining the two approaches.

### 5. THREATS TO VALIDITY
External: Our conclusions may not generalize beyond the scope of the projects we evaluated on. To mitigate this issue, we selected actively-developed projects from GitHub, and the projects vary in size, number of tests, size of history, and application. These projects were also used in previous research that studied the effects of software evolution on test-suite reduction [21].

Internal: We automated the process of recording what tests are selected by the regression test selection tool at each commit in each of the projects used in our evaluation. We also implemented the Greedy algorithm to perform test-suite reduction ourselves, as to the best of our knowledge, there is no publicly-available implementation. We confirmed the correctness of our code through many small runs and peer code review.

Construct: We define a new metric to measure loss in quality of a reduced test suite with change-related focus. To determine what are the change-related requirements that need to be satisfied, we effectively assume that regression test selection is not only safe but also relatively precise. We need regression test selection to be safe in order to get all of the requirements affected by change, and the precision is necessary to help filter out the requirements not affected by change but are accidentally satisfied by the tests selected by regression test selection. If regression test selection selects tests that are not affected by the changes (in the extreme selecting all tests), then we cannot filter out the requirements satisfied by these tests. The regression test selection tool we use, Ekstazi, tracks dependencies at the level of files (including classes for code) and is safe (with respect to any code changes). While Ekstazi is not as precise as other tools that may track finer-grained dependencies than files [29], Ekstazi is safer than other tools, and we find safety to be more important to properly determine what requirements are affected by changes. Also, Ekstazi is publicly available [1]. Another source of imprecision for this metric is if tests selected by regression test selection also satisfy a requirement that is not related to the change, yet cannot be satisfied by any test among the non-selected tests. However, in this case, we have some confidence that such a test that uniquely satisfies some requirement would likely be in the reduced test suite we are comparing against (it is a so called “essential test” [9,27]), so it is likely the metric would not show this as a loss in quality when considering the quality of the reduced test suite with respect to the changes.

We used Ekstazi to perform regression test selection across multiple commits in our evaluation. Ekstazi tracks dependencies at the level of test classes, so if a single test (method) is affected by a change, the entire test class is selected to be run. On the other hand, we performed test-suite reduction at the individual test level, which allows removing individual tests based on the requirements each individual test satisfies. As such, there is a discrepancy between the granularity of the selection of tests to run between the two approaches. By using Ekstazi, our results for regression test selection may end up selecting many more tests than actually necessary, so our comparison of test-suite sizes for regression test selection and test-suite reduction overestimates the sizes for regression test selection. Therefore, one of our key findings, that the size of selected test suites is, on average, smaller than the size of reduced test suite, could be even stronger.

We evolved a reduced test suite across many commits in a particular way (Section 2.2) to simulate how a test engineer who used a reduced test suite in lieu of the full test suite in future commits may have evolved the reduced test suite. This simulation allows a comparison to regression test se-
lection, which directly considers code changes. Specifically, our methodology for evolving the reduced test suite considers all changes made to the full test suite and mimics those changes. However, we note that, had the developer actually performed test-suite reduction, the reduced test suite could have evolved differently than the way we simulated it.

6. RELATED WORK

In our previous work, we studied the effect of software evolution on test-suite reduction [21]. We conducted large-scale experimentation on 18 open-source projects from GitHub and measured the quality of the constructed reduced test suite across multiple commits. We computed the difference in quality between points in software history and found that the quality of the reduced test suite does not drop very much even as software evolves. However, these measurements of quality was comparing the reduced test suite to the entire test suite and with respect to the entire software. In this work, we introduce a metric for measuring the quality of the tests run with respect to the changes. We find that when we consider just the changes, at the points where we measured, the reduced test suite does suffer some greater loss in fault-detection capability. We contrast this with tests selected to run by regression test selection, which suffers no loss in fault-detection capability with respect to changes and at the same time generally selects less tests to run that the reduced test suite.

Previous research has proposed multiple regression test selection techniques. Rothermel et al. [17] proposed several slicing techniques which were later surveyed by Yoo and Harman [27]. Rothermel and Harrold [17] implemented interprocedural and intraprocedural test selection tools that would select a ratio of tests to run ranging from 43.4% to 93.6% of the full test suite size. In our study, we used a regression test selection tool that selects tests at the test class level and tracks dependencies based on files. We examine a larger set of projects, consisting of 15 open-source projects, and also compare the number of tests to run for regression test selection with those of test-suite reduction. We find that the ratio of tests selected to run still approximately fell within the range observed by Rothermel and Harrold, and the median number of tests selected for each project are on average much lower than the size of the reduced test suite.

Korel et al. [15] described a test-suite reduction algorithm which utilized Extended Finite State Machine (EFSM) dependence analysis to select regression tests on a system level based on a set of elementary changes (addition and deletion of transitions) between the EFSM models of the original and modified systems. For each of these differences, tests are generated as paths from an initial state to a final state, and these sets of tests are unioned to form a single regression test suite. Test-suite reduction is then performed using this regression test suite and the original and modified EFSM models as inputs to generate a final regression test suite. To achieve this, the test-suite reduction algorithm removes tests equivalent with respect to the elementary modifications, i.e., if tests exhibit the same patterns of interactions with other system elements during traversal of the EFSM model. The study’s preliminary experiments observed significant reductions in the size of the regression test suites, ranging from 83% to 99%. This work is similar to our proposed selection of reduction as there is a composition of test-suite reduction with an approach that takes into account changes between versions of software. However, the dependence analysis described in this work must generate new tests to cover the modifications to the system, while we use regression test selection that selects tests from an already existing test suite, as we rely on code-based test-selection technique as opposed to a model-based one. Our study also measures the relative decreases in size between the test suites generated by regression test selection and by selection of reduction, as well as the loss in test suite quality caused by reduction.

Elbaum et al. [12] studied the effects of software evolution on the coverage of test suites. They found that coverage information changed greatly even for small changes in the program due to evolution; with only 1% of branches in the program affected by evolution, mean coverage of the program statements was reduced by 16%. Our research shares the common theme of evaluating the impact of evolution on test suites. We are more concerned with impact on fault-detection capability of test suites; further our focus is on reduced test suites and selected test-suites and their relation to the original test-suite.

Previous research has also studied alternatives to coverage-based test-suite reduction. Yoo et al. [28] reported on the initial results of a test-suite optimization technique adapted from a previous work [26], which utilized a multi-objective sampling based on coverage, cost, and fault history. Their work found that using multiple objectives reduced the regression testing time by 33% to 82% while retaining fault-detection capability.

Zhang et al. [30] combined test prioritization and reduction for mutation testing and found that prioritization reduced the number of executions by up to 47.52%, while reduction reduced them by around 50% with error rates of just 0.5% for mutation scores.

7. CONCLUSIONS

This paper is the first to empirically compare and propose to combine test-suite reduction (TSR) and regression test selection (RTS), two approaches that can speed up regression testing but were evaluated only separately. We propose a new metric to evaluate the loss in quality of running a set of tests with respect to the changes made in software. We also use the standard metric of test-suite size. Our results on 15 open-source projects show three interesting conclusions. First, RTS on average selects fewer tests than TSR. Second, TSR can lose fault-detection capability with change-related focus (of up to 5.85% for killed mutants), while (safe) RTS has no loss. Third, our proposed selection of reduction approach, which selects tests affected by changes from the reduced test suite, provides the greatest speed-up, though with the same loss in fault-detection capability as the reduced test suite with change-related focus, and has about the same selection ratio from the reduced test suite as RTS has from the full test suite.

In summary, our results show that if a test engineer needs to choose only one technique, either TSR or RTS, to speed up regression testing, the test engineer should choose RTS, as it selects fewer tests and preserves fault-detection capability with change-related focus, where the faults are most likely to appear. If there is a need to speed up testing even further, then combining both TSR and RTS is a worthwhile approach that provides even greater savings in testing time as long as one is willing to tolerate the possible loss in fault-detection capability for some changes in software.
8. REFERENCES