Operator-Based and Random Mutant Selection: Better Together

Lingming Zhang
University of Texas
Austin, TX 78712
zhanglm@utexas.edu

Milos Gligoric
University of Illinois
Urbana, IL 61801
gliga@illinois.edu

Darko Marinov
University of Illinois
Urbana, IL 61801
marinov@illinois.edu

Sarfraz Khurshid
University of Texas
Austin, TX 78712
khurshid@utexas.edu

***Abstract*—Mutation testing is a powerful methodology for evaluating the quality of a test suite. However, the methodology is also very costly, as the test suite may have to be executed for each mutant. *Selective mutation testing* is a well-studied technique to reduce this cost by selecting a subset of all mutants, which would otherwise have to be considered in their entirety. Two common approaches are *operator-based mutant selection*, which only generates mutants using a subset of mutation operators, and *random mutant selection*, which selects a subset of mutants generated using all mutation operators. While each of the two approaches provides some reduction in the number of mutants to execute, applying either of the two to medium-sized, real-world programs can still generate a huge number of mutants, which makes their execution too expensive. This paper presents eight random sampling strategies defined on top of operator-based mutant selection, and empirically validates that operator-based selection and random selection can be applied in tandem to further reduce the cost of mutation testing. The experimental results show that even sampling only 5% of mutants generated by operator-based selection can still provide precise mutation testing results, while reducing the average mutation testing time to 6.54% (i.e., on average less than 5 minutes for this study).**

I. Introduction

Mutation testing [1]–[10] has been shown to be a powerful methodology for evaluating the quality of test suites. In (first-order) mutation testing, many variants of a program (known as *mutants*) are generated through a set of rules (known as *mutation operators*) that seed one syntactic change at a time in order to generate one mutant. Then, all the generated mutants are executed against the test suite under evaluation. A test suite is said to *kill* a mutant if at least one test from the suite has a different result for the mutant and original program. If a mutant is semantically *equivalent* to the original program, then it cannot be killed by any test. The more non-equivalent mutants a test suite can kill, the higher quality the test suite is predicted to be. Indeed, several studies [11], [12] have found mutants to be suitable alternatives for real faults which can be used for comparing testing techniques and test suites. The predicted quality of test suites also depends on the quality of mutants, consequently higher-order mutation testing has been proposed to generate a small number of mutants that each have multiple syntactic changes [13], [14]. Mutation testing has also been used to guide the test generation to kill mutants automatically [14]–[18].

While mutation testing could be useful in many domains, it is extremely expensive. For example, a mutation testing tool for C, Proteum [19], implements 108 mutation operators that generate 4,937 mutants for a small C program with only 137 non-blank, non-comment lines of code [7]. Therefore, generating and (especially) executing the large number of mutants against the test suite under evaluation is costly. Various methodologies for reducing the cost of mutation testing have been proposed. One widely used methodology is *selective mutation testing* [3]–[7], [9], [20], [21], which only generates and executes a subset of mutants for mutation testing. Ideally, the selected subset of mutants should be representative of the entire set of mutants.

The most widely studied approach for selective mutation testing is *operator-based mutant selection* [3]–[5], [7], [9], [20], which only generates mutants using a subset of mutation operators; the selected subset of mutation operators is required to be *effective*, i.e., if a test suite kills all the non-equivalent mutants generated by the selected set of operators (i.e., the test suite is *adequate* for selected mutants), then the test suite should kill (almost) all the non-equivalent mutants generated by all mutation operators. Further, selected operators should lead to high savings; the *savings* is usually calculated as the ratio of non-selected mutants over all the mutants. Researchers also evaluated a simple approach of *random mutant selection* [7], [9], [22], and a recent study [7] reported that random selection is as effective as operator-based mutant selection when random selection selects the same number of mutants from *all mutants* as the operator-based selection selects.

Although the existing approaches are effective, mutation testing remains one of the most expensive methodologies in software testing. No previous study has explored how to further reduce the number of mutants generated by operator-based mutant selection, and whether operator-based selection and random selection can be combined. Also, previous work has not explored how random mutant selection and operator-based selection relate for test suites that do not kill all non-equivalent mutants (i.e., *non-adequate* test suites). In addition, all the studies for sequential mutants [3]–[5], [7], [20], [22] evaluated mutant selection on small C and Fortran programs—the largest program used for selective mutation testing was only 513 lines of code. Empirical studies on larger, real-world programs are lacking.

978-1-4799-0215-6/13 © 2013 IEEE

ASE 2013, Palo Alto, USA

This paper makes the following contributions:

- **Sampling mutation:** We investigate a simple idea, which we call *sampling mutation*, to reduce the number of mutants generated by operator-based mutant selection: sampling mutation randomly selects from the set of mutants generated by operator-based mutant selection (rather than from the set of mutants generated by all operators [7], [9], [22]); we call the process of obtaining the mutants *sampling*, the percentage of randomly selected mutants the *sampling ratio*, and the resulting set of mutants a *sample*.
- **Various sampling mutation strategies:** We evaluate 8 sampling strategies; our study is the first to consider random selection based on the program elements (rather than on the mutation operators). Our empirical study shows that although all sampling strategies are effective for mutation testing, sampling based on program elements can provide the most effective results.
- **Extensive study:** We evaluate mutation sampling on 11 real-world Java projects of various sizes (from 2681 to 36910 lines of code) to investigate the effectiveness, predictive power, and savings of sampling mutation. Our study evaluates effectiveness in case of adequate test suites, predictive power in case of non-adequate test suites, and savings in terms of time to generate and execute mutants.
- **Empirical evidence:** The study shows that sampling mutation remains effective and has a high predictive power even while providing high savings. The study shows the cost-effectiveness of applying sampling mutation with various strategies and ratios. Surprisingly, for all our subjects, the experimental results show that sampling only 5% of operator-based selected mutants can still provide a precise mutation score, with almost no loss in precision, while reducing the mutation time to 6.54% on average. Moreover, the study shows that the sampling strategies are more beneficial for larger subjects; as more and more researchers are using mutants to compare testing techniques, our sampling strategies can help researchers to scale mutation to larger programs by choosing a representative subset of mutants for efficient but effective evaluation.

II. Study Approach

A. Problem Definition

Given a program under test, P, and a test suite, T, we denote the set of all *selected mutants* generated by operator-based mutant selection as M, and the set of non-equivalent mutants in M as NEM. Following existing studies [3]–[5], [7], we randomly construct n test suites of various sizes $\{T_1, T_2, ..., T_n\}$; the set of mutants that can be killed by T_i $(1 \leq i \leq n)$ is denoted $K(T_i, M)$. Then the (actual) *selected mutation score* achieved by T_i over the selected mutants M is defined as:

$$MS(T_i, M) = \frac{|K(T_i, M)|}{|NEM|} \quad (1)$$

In this study, we apply a set of sampling strategies on top of the selected mutants. Let S be a sampling strategy; the set of *mutants sampled* by S from M is denoted M_S. We apply each strategy m times (with different random seeds) to generate a set of mutant samples: $\{M_{S_1}, M_{S_2}, ..., M_{S_m}\}$. The set of mutants in M_{S_j} $(1 \leq j \leq m)$ that are killed by test suite T_i $(1 \leq i \leq n)$ is denoted $K(T_i, M_{S_j})$. Then, the *sampling mutation score* achieved by T_i over M_{S_j} can be represented as:

$$MS(T_i, M_{S_j}) = \frac{|K(T_i, M_{S_j})|}{|M_{S_j} \cap NEM|} \quad (2)$$

Intuitively, if $MS(T_i, M_{S_j})$ is close to $MS(T_i, M)$ for all $1 \leq i \leq n$ and $1 \leq j \leq m$, we say that the sampling strategy S applied on top of selected mutants is *effective* at predicting the result that would be obtained on all selected mutants. (Section III precisely defines the predictive power.)

B. Measurement

In the literature, there are two main approaches for evaluating the effectiveness of how a subset of mutants represents a larger set of mutants. (Traditionally, the sets are generated by all operators, and the subsets are selected mutants; in our study, the sets are selected mutants, and the subsets are sampled mutants.) First, researchers [3], [4], [7], [20] construct test suite T_i $(1 \leq i \leq n)$ that *can* kill all non-equivalent mutants from the subset (called *adequate* test suites), and calculate the mutation score of T_i on the original set of mutants. Second, Namin et al. [5] also examined how, for test suites T_i $(1 \leq i \leq n)$ that *cannot* kill all the non-equivalent mutants from the subset (called *non-adequate* test suites), the mutation score of T_i on the subset of mutants compares with the mutation score of T_i on the original set of mutants.

In this study, we use both approaches to evaluate the sampling strategies applied on top of operator-based mutant selection. For the first approach, since our sampling strategy may select different subsets of mutants at different runs, we randomly construct n adequate test suites that can kill all sampled non-equivalent mutants for each of the m sampling runs. We denote the ith $(1 \leq i \leq n)$ test suite that kills all sampled non-equivalent mutants in the jth $(1 \leq j \leq m)$ run of sampling (i.e., the selected mutants are M_{S_j}) as T_{ij}. Following previous work [7], we use the following formula to measure the *effectiveness* of a sampling technique S:

$$EF(S) = \frac{\Sigma_{j=1}^{m} \Sigma_{i=1}^{n} |MS(T_{ij}, M)|}{n * m} \times 100\% \quad (3)$$

The only difference between our formula and the original formula [7] is that we also average over m sampling runs; the previous work did not average over different runs because each run of their operator-based selection gives a fixed subset of mutants. Also note that in the evaluation, we present the standard deviation (SD) values across m sampling runs to show the *stability* of the sampling strategies.

For the second approach, we randomly construct k non-adequate test suites ($\{T_1, T_2, ..., T_k\}$) for each subject and check how m runs of sampling influence the mutation scores

of the constructed test suites. We use the correlation between sampling mutation score of T_i and selected mutation score of T_i to measure the *predictive power* of a random sampling strategy S over the mutants generated by operator-based mutant selection:

$$PP(S) = Corr(\{\langle MS(T_i, M_{S_j}), MS(T_i, M)\rangle \mid 1 \leq i \leq k \wedge 1 \leq j \leq m\}) \quad (4)$$

The correlation analysis is between the mutation scores on the sampled mutant set M_{S_j} and the mutation scores on the selected mutants M for all constructed test suites for all sampling runs ($1 \leq j \leq m$). Section III presents, both visually and statistically, the $Corr$ functions we use. To illustrate, for all the m sampling runs of a strategy S, if we use the mutation scores of the k tests suites on sampled mutants M_{S_j} ($1 \leq j \leq m$), i.e., $MS(T_i, M_{S_j})$ ($1 \leq i \leq k$) as the x-axis values, for each x value we will have a corresponding y value to predict, which is $MS(T_i, M)$ ($1 \leq i \leq k$). For a perfect strategy, the graph will be the straight line function $y = x$, which means all the k test suites have exactly the same mutation score on sampled mutants and all original mutants before sampling.

C. Combining Operator-Based and Random Mutant Selection

Given selected mutants, M, we define eight random sampling strategies that specify which mutants to select from M.

- **Baseline Strategy**, which samples $x\%$ mutants from the selected set of mutants M. Formally, the set of mutants sampled by strategy S_{base} can be defined as:

  $$M_{S_{base}} = Sample(M, x\%)$$

  where $Sample(M, x\%)$ denotes random sampling of $x\%$ mutants from M.[1]
- **MOp-Based Strategy**, which samples $x\%$ mutants from each set of mutants generated by the same mutation operator. Assume the sets of mutants generated by the set of selective mutation operators, say $op_1, op_2, ..., op_k$, are M_{op_1}, M_{op_2}, ..., M_{op_k}, i.e., $M = \cup_{i=1}^{k} M_{op_i}$. Then, the set of mutants sampled by strategy S_{mop} can be formally defined as:

  $$M_{S_{mop}} = \cup_{i=1}^{k} Sample(M_{op_i}, x\%)$$

- **PElem-Based Strategies**, which sample $x\%$ mutants from each set of mutants generated inside the same program element (e.g., class, method, or statement). Assume the sets of mutants generated for the set of elements in the project under test are M_{e_1}, M_{e_2}, ..., M_{e_k}, i.e., $M = \cup_{i=1}^{k} M_{e_i}$. Then, the set of sampled mutants can be defined as:

  $$M_{S_{pelem}} = \cup_{i=1}^{k} Sample(M_{e_i}, x\%)$$

  In this way, S_{class}, S_{meth}, and S_{stmt} can be defined when using the program element granularities of class, method, and statement, respectively.
- **PElem-MOp-Based Strategies**, which sample $x\%$ mutants from each set of mutants generated by the same mutation operator inside the same program element. Assume the sets of mutants generated for the set of program elements in the project under test are M_{e_1}, M_{e_2}, ..., M_{e_k}, then $M = \cup_{i=1}^{k} M_{e_i}$. Also assume the sets of mutants generated by the set of selective mutation operator are M_{op_1}, M_{op_2}, ..., M_{op_h}, then $M = \cup_{j=1}^{h} M_{op_j}$. Finally, the set of sampled mutants can be defined as:

  $$M_{S_{pelem-mop}} = \cup_{i=1}^{k} \cup_{j=1}^{h} Sample(M_{e_i} \cap M_{op_j}, x\%)$$

  In this way, $S_{class-mop}$, $S_{meth-mop}$, and $S_{stmt-mop}$ can be defined when using the program element granularities of class, method, and statement, respectively.

Note that the first two strategies, S_{base} and S_{mop}, have been used by previous studies [7], [22] to evaluate random mutant selection from *all mutants*. We believe that using all mutants as the candidate set may be unnecessary. Therefore, we use these two strategies to evaluate random mutant sampling from *operator-based selected mutants*. In addition, our three S_{pelem} strategies, which aim to sample mutants across all program locations evenly, are *the first* to randomly sample mutants at the program element dimension. Furthermore, our three $S_{pelem-mop}$ strategies are *the first* to sample mutants across two dimensions: mutation operators and program elements.

[1] If the sample size is a float f, we first sample $\lfloor f \rfloor$ mutants at random, and then with probability $f - \lfloor f \rfloor$ pick one more mutant at random.

TABLE I: Subject programs used in the evaluation

Subject	LOC	#Tests	#Mutants All	#Mutants Killed
TimeMoney^{r207} [23]	2681	236	2304	1667
JDepend$^{v2.9}$ [24]	2721	55	1173	798
JTopas$^{v2.0}$ [25]	2901	128	1921	1103
Barbecue^{r87} [26]	5391	154	36418	1002
Mime4J$^{v0.50}$ [27]	6954	120	19111	4414
Jaxen^{r1346} [28]	13946	690	9880	4616
XStream$^{v1.41}$ [29]	18369	1200	18046	10022
XmlSecurity$^{v3.0}$ [30]	19796	83	9693	2560
CommonsLang^{r1040879} [31]	23355	1691	19746	12970
JodaTime^{r1604} [32]	32892	3818	24174	16063
JMeter$^{v1.0}$ [33]	36910	60	21896	2024

III. Empirical Study

We performed an extensive empirical evaluation to demonstrate the effectiveness, predictive power, and savings of the proposed sampling strategies.

A. Subject Programs

The evaluation includes a broad set of Java programs from various sources. We chose programs of different sizes (from 2681 to 36910 LOC) to explore the benefits of our sampling strategies for various cases.

Table I shows 11 subject programs used in the evaluation: TimeMoney, a set of classes for manipulating time and money; JDepend, a tool for measuring the quality of code design; JTopas, a library for parsing arbitrary text data; Barbecue, a library for creating barcodes; Mime4J, a parser for e-mail message streams in MIME format; Jaxen, an implementation of XPath engine; XStream, a library for fast serialization/deserialization to/from XML; XmlSecurity, an Apache project that implements security standards for XML; CommonsLang, an Apache project that extends standard Java library; JodaTime, a replacement for standard Java date and time classes; and JMeter, an Apache project for performance testing. All the 11 subjects have been widely used in software testing research [6], [10], [34]–[37].

Table I includes some characteristics of the programs. Column "Subject" shows the name of each subject program, the version/revision number (as applicable) and the reference to the webpage with sources; "LOC" shows the number of non-blank lines of code measured by JavaSourceMetric [38]; "#Tests" shows the number of available tests for the program (it is important to note that we have not created any special test for the purpose of this study: all the tests for 11 subjects come from their code repositories, and to the best of our knowledge, all these tests are manually written); "#MutantsAll" and "#MutantsKilled" show the total number of mutants that Javalanche [6], [34] generated using operator-based selection and the number of killed mutants, respectively. We used Javalanche because it is a state-of-the-art mutation testing tool for Java programs. It generates mutants using the operator-based mutant selection approach proposed by Offutt et al. [3], [20]. Specifically, Javalanche uses the following four mutation operators: Negate Jump Condition, Omit Method Call, Replace Arithmetic Operator, and Replace Numerical Constant. Note that the subjects used in our study are orders of magnitude larger than the subjects used in previous studies on selective mutation testing [3], [5], [7], [20], [22].

B. Experimental Design

We next describe our experimental setup and the data we collected.

1) Independent Variables: We used the following independent variables in the study:

IV1: Different Random Sampling Strategies. We apply each of our eight sampling strategies on top of the mutants generated by operator-based mutant selection, to investigate their effectiveness, predictive power, and savings.

IV2: Different Sampling Ratios. For each sampling strategy S, we use 19 sampling ratios $r \in \{5\%, 10\%, ..., 95\%\}$.

IV3: Different Subject Sizes. For each strategy S with each ratio r, we apply S on all the subjects with various sizes, and investigate the differences.

2) Dependent Variables: We used the following dependent variables to investigate the output of the experiments:

DV1: Effectiveness. For the mutants sampled by each strategy S among all selected mutants, we construct test suites that can kill all sampled non-equivalent mutants, and record the selected mutation score of those test suites. The higher the selected mutation score is, the more effective the selected mutants are for evaluating test suites (Equation 3). (The same experimental procedure was used previously to measure the effectiveness of operator-base selection and random selection [3]–[5], [7], [9], [20].)

DV2: Predictive Power. For each sampling strategy S, we also construct test suites that do not kill all sampled non-equivalent mutants, and use statistical analysis to measure the predictive power of the sampled mutants (equation 4). If the constructed test suites have similar values for sampling mutation score and selected mutation score, then the sampled mutants are a good predictor of the selected mutants. More precisely, we instantiate the $Corr$ function to measure: R^2 coefficient of determination for linear regression, Kendall's τ rank correlation coefficient, and Spearman's ρ rank correlation coefficient.

DV3: Time Savings. For each triple (P, S, r) of subject program P, sampling strategy S, and sampling ratio r, we compare the mutation testing time for the sampled mutants and the mutation testing time for the selected mutants.

3) Experimental Setup: Following previous studies on selective mutation testing [5], [7], we deemed all mutants that cannot be killed by any test from the original test suite as equivalent mutants in our study. We evaluate all the sampling strategies with all sampling ratios on all subjects. Given a subject program and selected mutants for that program, we first run sampling 20 times for each of 8 sampling strategies with each of the 19 sampling ratios. As a result, we get 20*8*19=3,040 samples of mutants for each subject program.

Then, for each sample of mutants, we randomly construct 20 adequate test suites that each kill all the non-equivalent mutants in sampled mutants, i.e., we construct 20*3,040=60,800 test suites for each subject. Next, we measure the selected mutation score for each test suite. Each test suite is randomly constructed by including one test at a time until all sampled non-equivalent mutants are killed. We deviate from the previous work [3], [7], [20] that constructed test suites by including multiple tests at a time (using increment of 50 or 200), as such decision can lead to large test suites and high selected mutation scores that do not correspond to practice. By including one test at a time, we simulate a more realistic use of mutation testing in practice, where a user could include one test at a time until all the mutants are killed.

Next, for each subject, we randomly construct 100 (non-adequate) test suites of various sizes that do not necessary kill all the sampled mutants. We randomly construct each test suite by uniformly selecting the size of the test suite to be between 1 and the number of tests available for the subject. Note that our experiments differ in this step from previous work [5], where 100 test suites were generated by taking two test suites for each size between 1 and 50. The reason to deviate from previous work is that our programs greatly differ in size and number of tests, which was not the case in previous studies. For example, taking sizes between 1 and 50 does not seem appropriate for both Barbecue and JodaTime (with 154 and

3818 tests, respectively). Therefore, we uniformly select the sizes of the test suites up to the total number of tests for each subject program. Then we measure the sampling mutation score (i.e., the mutation score on the sampled mutants) and selected mutation score (i.e., the mutation score on the selected mutants) achieved by each of the constructed test suites. We further perform correlation analysis between the sampling mutation score and the selected mutation score for all test suites on each strategy and ratio combination on each subject. (Section III-C2 shows the details.)

Finally, for each sample of mutants, we also trace the time for generating and executing the mutants. Although it is common in the literature to report the savings in terms of the number of mutants not generated, this information is implicitly given in our study through the sampling ratio (e.g., if a sampling ratio is 5%, we have 20x fewer mutants). Therefore, our study also reports the mutation execution time in order to confirm that savings in terms of the number of mutants correspond to the savings in terms of mutation execution time for mutation sampling. We performed all experiments on a Dell desktop with Intel i7 8-Core 2.8GHz processor, 8G RAM, and Windows 7 Enterprise 64-bit version.

C. Results and Analysis

We report the most interesting findings of our study in this section, while some additional results and detailed experimental data are publicly available online [39].

1) Effectiveness for Adequate Test Suites: Table II shows the selected mutation scores achieved by randomly constructed adequate test suites that achieve 100% sampled mutation score, i.e., kill all the sampled non-equivalent mutants. According to our experimental setup, for each triple of subject program, strategy, and sampling ratio, (P, S, r), we obtain 20 samples of mutants and construct 20 adequate test suites for each sample. Thus, for each (P, S, r), we show the average selected mutation score and standard deviation achieved by the 20*20=400 test suites. Specifically, column "Ra." shows sampling ratio, column "Subject" shows the subject name, and columns 3-18 show the average values and standard deviations achieved by 8 sampling strategies. The results for all the 19 sampling ratios can be found on the project webpage [39]. Based on the obtained values, we make several observations as follows.

First, for all subjects and all sampling strategies, one can see that the sampled mutants are extremely effective, i.e., the sampled mutants are representative of the selected mutants. For example, even when sampling 5% of the selected mutants, the test suites that kill all the sampled mutants can kill almost all selected mutants. To illustrate, when sampling 5% of selected mutants, the selected mutation score for S_{base} strategy ranges from 98.23% (on JDepend) to 99.91% (on JodaTime) with the average value of 99.44%. As the sampling ratio increases, all the strategies have higher selected mutation score and lower standard deviation for all subjects. This demonstrate that a user can use the sampling strategies to control the cost-effectiveness of mutation testing: the more mutants sampled, the more precise and stable the results would be.

Second, the studied strategies perform better on larger subjects than on the smaller subjects. For example, when sampling 5% of mutants, S_{meth} achieves the average selected mutation scores ranging from 98.31% to 99.32% for the first four subjects that have fewer than 6000 LOC, while it achieves the average selected mutation scores ranging from 99.69% to 99.92% for all the other seven larger subjects. This demonstrates that using small sampling ratios (e.g., r=5%) of mutants is more beneficial for evaluating test suites for larger subjects. Section III-D further investigates the effectiveness of sampling mutation for ratios even below 5%.

Third, all the strategies perform similarly, but S_{meth} and $S_{meth-mop}$ tend to perform the best of all the strategies for the majority of the subjects. Moreover, the additional use of mutation operator information in $S_{meth-mop}$ does not make it outperform S_{meth}. This demonstrates that sampling mutants across different program elements can be a better choice than sampling mutants globally (S_{base}) or across different mutation operators (S_{mop}). S_{meth} performs better than S_{class} and S_{stmt} likely because sampling at the class level is too coarse (bringing it closer to S_{base}), while sampling at the statement level is too fine making it select no mutant from some statements (because the number of mutants for each statement is relatively small).

2) Predictive Power for Non-Adequate Test Suites: While the above results showed that *adequate* sampling mutation score implies high selected mutation score, it is uncommon in practice to have adequate test suites. Thus, we further investigate the predictive power of the sampling strategies for *non-adequate* test suites that do not kill all sampled non-equivalent mutants. More precisely, we analyze whether the sampling mutation score is a good predictor of the selected mutation score across a range of test suites, which are almost all non-adequate. Ideally, for all (non-adequate) test suites sampling and selected mutation score would have the same value. In practice, if a test suite achieves selected mutation score MS, the same test suite may achieve sampling mutation score MS' such that $MS < MS'$, $MS = MS'$, or $MS > MS'$. We use three statistical measures to evaluate the predictive power of sampling mutation score for all strategies.

Evaluating Single Test Suite. Originally, mutation testing was proposed as a method for evaluating the quality of test suites by measuring mutation score; the higher mutation score means higher quality. To evaluate a test suite using one of the sampling strategies, we have to ensure that the result obtained on the sampled mutants predicts the result that would be obtained on all the selected mutants. Following previous work [5], we determine how well the independent variable (sampling mutation score) predicts the dependent variable (selected mutation score) using a linear regression model. We measure the quality of fit of a model by calculating the adjusted coefficient of determination R^2, which is a statistical measure of how well the regression line approximates the real

TABLE II: Selected mutation scores (%) achieved by the test suites that achieve 100% sampled mutation scores

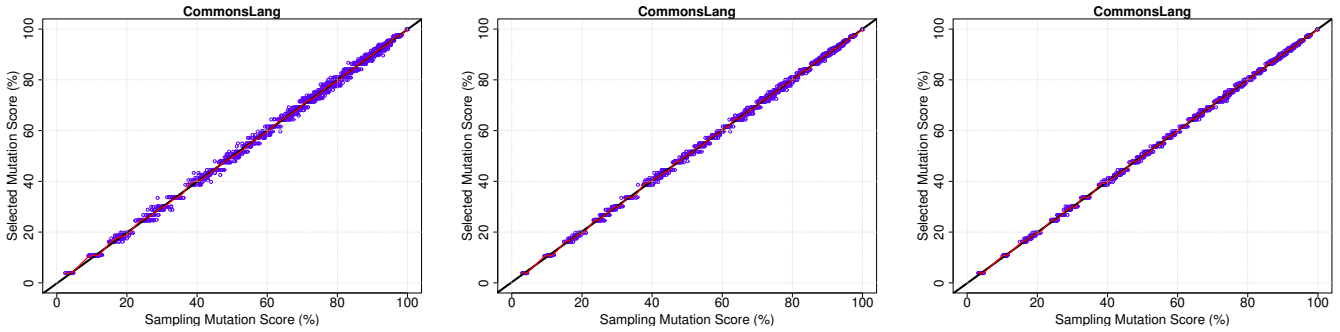
Ra.	Subjects	Base		MOp		Class		Meth		Stmt		Class-MOp		Meth-MOp		Stmt-MOp	
		MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.	MS.	Dev.
5%	TimeMoney	99.14	1.12	99.30	0.98	99.15	1.20	99.32	0.90	99.35	0.94	99.10	1.23	99.26	1.00	99.09	1.17
	JDepend	98.23	1.85	98.10	2.09	97.81	2.34	98.31	1.91	98.31	1.71	97.99	2.31	98.54	1.84	98.67	1.54
	JTopas	99.37	1.09	99.24	1.23	99.32	1.18	99.25	1.18	99.30	1.22	99.19	1.35	99.25	1.31	99.16	1.30
	Barbecue	98.63	1.66	98.64	1.82	98.92	1.56	98.99	1.26	99.03	1.37	98.64	1.76	99.04	1.41	98.69	1.73
	Mime4J	99.77	0.34	99.78	0.34	99.77	0.32	99.81	0.29	99.76	0.32	99.82	0.26	99.80	0.28	99.78	0.39
	Jaxen	99.72	0.33	99.69	0.36	99.67	0.37	99.69	0.33	99.67	0.42	99.70	0.36	99.64	0.39	99.70	0.36
	XStream	99.85	0.21	99.86	0.17	99.87	0.18	99.90	0.14	99.88	0.15	99.87	0.18	99.88	0.15	99.85	0.19
	XmlSecurity	99.62	0.61	99.53	0.74	99.68	0.56	99.69	0.50	99.64	0.70	99.73	0.46	99.73	0.45	99.68	0.56
	CommonsLang	99.90	0.13	99.89	0.15	99.89	0.14	99.92	0.10	99.90	0.14	99.89	0.14	99.90	0.12	99.89	0.13
	JodaTime	99.91	0.12	99.91	0.11	99.91	0.11	99.92	0.09	99.92	0.11	99.91	0.11	99.91	0.10	99.91	0.12
	JMeter	99.71	0.53	99.73	0.52	99.76	0.43	99.76	0.37	99.72	0.48	99.72	0.51	99.77	0.44	99.67	0.59
	Avg.	**99.44**	-	**99.42**	-	**99.43**	-	**99.51**	-	**99.50**	-	**99.41**	-	**99.52**	-	**99.46**	-
10%	TimeMoney	99.61	0.61	99.66	0.47	99.71	0.43	99.75	0.37	99.67	0.48	99.67	0.48	99.69	0.45	99.72	0.43
	JDepend	99.21	1.17	99.27	1.03	99.35	0.90	99.40	0.89	99.43	0.84	99.32	0.92	99.38	0.91	99.32	0.87
	JTopas	99.67	0.61	99.62	0.65	99.74	0.44	99.66	0.54	99.65	0.65	99.64	0.61	99.76	0.46	99.75	0.43
	Barbecue	99.45	0.79	99.36	0.95	99.42	0.82	99.60	0.64	99.56	0.62	99.54	0.72	99.47	0.88	99.49	0.74
	Mime4J	99.91	0.17	99.90	0.16	99.93	0.13	99.91	0.15	99.91	0.15	99.92	0.13	99.92	0.13	99.92	0.14
	Jaxen	99.85	0.19	99.85	0.20	99.87	0.17	99.87	0.17	99.87	0.17	99.85	0.19	99.86	0.18	99.87	0.16
	XStream	99.95	0.08	99.94	0.08	99.94	0.08	99.96	0.06	99.96	0.07	99.95	0.09	99.95	0.07	99.95	0.09
	XmlSecurity	99.88	0.23	99.86	0.27	99.88	0.22	99.86	0.23	99.88	0.20	99.86	0.26	99.86	0.25	99.88	0.25
	CommonsLang	99.96	0.06	99.96	0.06	99.96	0.06	99.97	0.04	99.96	0.07	99.95	0.06	99.96	0.06	99.96	0.06
	JodaTime	99.96	0.05	99.96	0.05	99.96	0.06	99.97	0.04	99.97	0.04	99.96	0.06	99.97	0.05	99.97	0.05
	JMeter	99.86	0.27	99.87	0.25	99.90	0.21	99.92	0.17	99.88	0.23	99.88	0.27	99.89	0.24	99.88	0.25
	Avg.	**99.76**	-	**99.75**	-	**99.79**	-	**99.81**	-	**99.79**	-	**99.78**	-	**99.79**	-	**99.79**	-
15%	TimeMoney	99.81	0.30	99.81	0.30	99.82	0.29	99.88	0.19	99.86	0.23	99.84	0.27	99.86	0.24	99.81	0.30
	JDepend	99.42	0.84	99.50	0.82	99.69	0.55	99.63	0.53	99.74	0.48	99.63	0.57	99.59	0.59	99.62	0.60
	JTopas	99.82	0.32	99.82	0.35	99.80	0.32	99.84	0.28	99.87	0.24	99.85	0.26	99.83	0.32	99.79	0.37
	Barbecue	99.68	0.50	99.73	0.40	99.70	0.47	99.74	0.36	99.70	0.46	99.72	0.47	99.74	0.41	99.69	0.52
	Mime4J	99.95	0.10	99.95	0.10	99.96	0.08	99.96	0.09	99.96	0.08	99.94	0.10	99.97	0.08	99.95	0.10
	Jaxen	99.90	0.15	99.91	0.12	99.92	0.11	99.92	0.10	99.92	0.12	99.92	0.10	99.92	0.12	99.92	0.11
	XStream	99.97	0.05	99.97	0.05	99.97	0.06	99.98	0.04	99.98	0.04	99.97	0.05	99.97	0.04	99.97	0.06
	XmlSecurity	99.91	0.19	99.94	0.12	99.93	0.16	99.92	0.15	99.94	0.11	99.92	0.14	99.95	0.11	99.93	0.15
	CommonsLang	99.97	0.05	99.98	0.04	99.97	0.04	99.99	0.02	99.98	0.03	99.98	0.03	99.98	0.03	99.98	0.04
	JodaTime	99.98	0.03	99.98	0.03	99.98	0.03	99.99	0.02	99.98	0.02	99.98	0.03	99.98	0.03	99.98	0.03
	JMeter	99.93	0.16	99.93	0.17	99.93	0.18	99.96	0.12	99.94	0.15	99.94	0.17	99.96	0.14	99.94	0.15
	Avg.	**99.85**	-	**99.87**	-	**99.88**	-	**99.89**	-	**99.90**	-	**99.88**	-	**99.89**	-	**99.87**	-
20%	TimeMoney	99.89	0.19	99.88	0.21	99.88	0.20	99.91	0.15	99.91	0.16	99.88	0.20	99.90	0.21	99.89	0.18
	JDepend	99.73	0.45	99.74	0.45	99.71	0.52	99.78	0.38	99.83	0.33	99.80	0.34	99.78	0.41	99.73	0.48
	JTopas	99.89	0.20	99.89	0.20	99.86	0.26	99.90	0.20	99.89	0.20	99.89	0.22	99.85	0.31	99.87	0.27
	Barbecue	99.80	0.38	99.72	0.44	99.81	0.31	99.84	0.28	99.82	0.31	99.80	0.33	99.81	0.33	99.78	0.39
	Mime4J	99.97	0.07	99.97	0.07	99.98	0.05	99.97	0.05	99.98	0.04	99.97	0.06	99.97	0.06	99.97	0.07
	Jaxen	99.95	0.07	99.94	0.09	99.94	0.09	99.95	0.08	99.94	0.08	99.95	0.07	99.94	0.08	99.94	0.09
	XStream	99.98	0.03	99.98	0.04	99.98	0.03	99.99	0.02	99.98	0.03	99.98	0.03	99.99	0.03	99.98	0.03
	XmlSecurity	99.95	0.09	99.94	0.13	99.95	0.10	99.96	0.08	99.97	0.07	99.94	0.15	99.95	0.10	99.95	0.10
	CommonsLang	99.99	0.02	99.98	0.03	99.99	0.02	99.99	0.02	99.99	0.02	99.98	0.03	99.99	0.02	99.99	0.03
	JodaTime	99.99	0.02	99.99	0.02	99.98	0.02	99.99	0.01	99.99	0.01	99.99	0.02	99.99	0.02	99.99	0.02
	JMeter	99.95	0.15	99.95	0.13	99.95	0.14	99.98	0.09	99.96	0.12	99.97	0.10	99.98	0.07	99.94	0.16
	Avg.	**99.92**	-	**99.91**	-	**99.91**	-	**99.93**	-	**99.93**	-	**99.92**	-	**99.92**	-	**99.91**	-

data points. The value of R^2 is between 0 and 1, where a higher value indicates a better goodness of fit.

We calculate R^2 for each triple (P, S, r) consisting of a subject program, strategy, and ratio. For each sample strategy S, we sample mutants at each ratio r and measure sampling mutation score for the same set of randomly constructed test suites (Section III-B3). We repeat sampling 20 times to obtain sampling and selected mutation scores for a variety of samples. We then calculate how well the sampling mutation scores from all the 20 sampling runs predict the selected mutation scores by calculating R^2 values[2]. Note that we calculate R^2 for all 20 sampling runs at once (which gives a more robust result than calculating R^2 for individual runs and averaging the result over 20 sampling runs). To illustrate, Figure 1 shows scatter plots of the sampling mutation score and the selected mutation score for CommonsLang. In each of the three subfigures, the x-axis shows the sampling mutation scores achieved by the test suites on various sampling runs, while the y-axis shows the selected mutation score for the same test suites. There are 20*100=2,000 points on each plot. From the three subfigures, we can see that a higher sampling ratio (r) leads to more stable data points, which can also be seen by smoother splines. However, note that the sampling mutation scores on all sampling runs are close to their selected mutation scores even when $r = 5\%$.

The left part of Table III shows R^2 values for all strategies with the sampling ratio of 5% on all subjects. (Due to the space limit, the detailed results for the other ratios are not shown but can be found on the project webpage [39].) Column "Subjects" lists the name of the subjects, and columns 2-9 include R^2 values for all 8 sampling strategies. The higher the R^2 value is, the better predictor the sampling strategy is. We find that the R^2 results at the 5% ratio level are

[2]We use R language for statistical computing.



(a) 5% ratio (b) 10% ratio (c) 15% ratio

Fig. 1: Sampling mutation score vs. Selected mutation score, with best fit line (black color) and smoothing spline line (red color), for CommonsLang subject, Meth strategy, and three different sampling ratios

TABLE III: R^2 and τ correlation values between mutation scores on sampled 5% mutants and on mutants before sampling

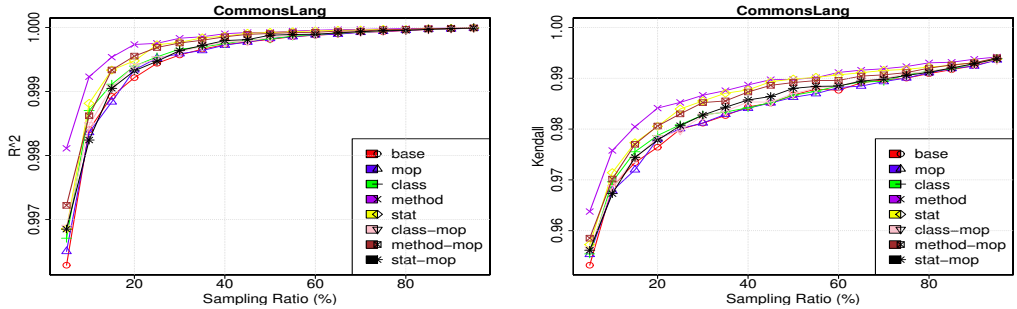
Subjects	R^2 correlation								Kendall's τ correlation							
	Base	MOp	Class	Meth	Stmt	Class -MOp	Meth -MOp	Stmt -MOp	Base	MOp	Class	Meth	Stmt	Class -MOp	Meth -MOp	Stmt -MOp
TimeMoney	0.971	0.972	0.975	0.977	0.976	0.975	0.975	0.972	0.874	0.872	0.881	0.888	0.882	0.877	0.880	0.872
JDepend	0.936	0.928	0.927	0.946	0.948	0.933	0.934	0.920	0.784	0.782	0.774	0.791	0.796	0.766	0.789	0.790
JTopas	0.932	0.923	0.909	0.945	0.944	0.943	0.922	0.922	0.845	0.834	0.826	0.856	0.851	0.850	0.836	0.831
Barbecue	0.956	0.951	0.958	0.970	0.964	0.961	0.962	0.946	0.844	0.833	0.849	0.867	0.861	0.854	0.857	0.832
Mime4J	0.991	0.992	0.993	0.994	0.992	0.992	0.993	0.991	0.936	0.937	0.938	0.943	0.936	0.938	0.939	0.933
Jaxen	0.985	0.987	0.983	0.990	0.984	0.986	0.986	0.982	0.894	0.896	0.886	0.910	0.898	0.902	0.901	0.890
XStream	0.993	0.994	0.996	0.996	0.995	0.996	0.996	0.995	0.942	0.946	0.954	0.956	0.949	0.953	0.954	0.948
XmlSecurity	0.982	0.984	0.986	0.986	0.983	0.987	0.984	0.982	0.888	0.892	0.904	0.900	0.894	0.906	0.899	0.896
CommonsLang	0.996	0.996	0.997	0.998	0.997	0.997	0.997	0.997	0.953	0.955	0.955	0.964	0.957	0.956	0.958	0.956
JodaTime	0.996	0.996	0.997	0.998	0.996	0.996	0.997	0.996	0.950	0.950	0.954	0.957	0.952	0.950	0.953	0.949
JMeter	0.982	0.982	0.988	0.989	0.985	0.985	0.988	0.980	0.915	0.917	0.931	0.935	0.925	0.924	0.931	0.913
Avg.	**0.975**	**0.973**	**0.974**	**0.981**	**0.979**	**0.977**	**0.976**	**0.971**	**0.893**	**0.892**	**0.896**	**0.906**	**0.900**	**0.898**	**0.900**	**0.892**

already extremely high, e.g., ranging from 0.945 (on JTopas) to 0.998 (on CommonsLang) for the S_{meth} strategy. This further confirms our findings for adequate test suites—the sampling ratio of 5% can be effective for mutation testing in practice. In addition, similar to our findings for adequate test suites, the sampling strategies are less effective for smaller subjects, e.g., the R^2 for the S_{meth} strategy ranges from 0.945 to 0.977 for the first four subjects below 6,000 LOC, while it is over 0.98 for the other seven larger subjects. Furthermore, although all the strategies perform well, the S_{meth} strategy slightly outperforms S_{base} and S_{mop} for all the 11 subjects, indicating again that sampling across different program elements can be a better choice than sampling purely randomly from all mutants or sampling across different mutation operators.

To show how the correlation varies when the sampling ratio changes, Figure 2a shows the R^2 values for all 8 strategies when the sampling ratio increases from 5% to 95% for the subject CommonsLang. The plots for the other subjects look similar and are available on the project webpage [39]. We can draw the following conclusions. First, S_{meth} is slightly better than the other strategies across all sampling ratios, further demonstrating the benefits of sampling mutants across program elements. Second, more importantly, all sampling strategies predict the selected mutation score very well. Across all the programs, strategies, and ratios, the minimum R^2 was 0.909 (for JTopas). Extremely high R^2 gives evidence that sampling mutation is valuable and can be used for evaluation of test suites. We believe that the results of our study can greatly impact the use of mutation testing in research practice; using sampling mutation testing makes it feasible to evaluate the quality of test suites for large-scale programs.

Comparing Testing Techniques and Test Suites. Mutation testing has also been extensively used in studies that compare testing techniques [40]–[42]. Commonly, a testing technique or a test suite that has a relatively higher mutation score than another testing technique or test suite is claimed to be better (regardless of the absolute mutation score that it achieves). We thus want to evaluate whether sampling mutation can be used for comparison of testing techniques and test suites, i.e., if a test suite T has a higher *sampling* mutation score than another test suite T', does T have a higher *selected* mutation score than T'? Similar to a previous study [5], we calculate Kendall's τ and Spearman's ρ rank correlation coefficients, which measure the strength of the agreement between two rankings. Both τ and ρ can take values between -1 and 1, where 1 indicates perfect agreement, and -1 indicates perfect disagreement.

To illustrate how τ is computed, consider all the pairs of sampling and selected mutation scores; two pairs (MS_1, MS'_1) and (MS_2, MS'_2) are said to be concordant if $(MS_1 > MS_2 \wedge MS'_1 > MS'_2) \vee (MS_1 < MS_2 \wedge MS'_1 < MS'_2)$ and discordant if $(MS_1 < MS_2 \wedge MS'_1 > MS'_2) \vee (MS_1 > MS_2 \wedge MS'_1 < MS'_2)$; otherwise, the pair is neither concor-



(a) R^2 (b) τ

Fig. 2: Correlation values for CommonsLang subject, all strategies, and all rates

TABLE IV: Selective and sampling mutation testing time

Subject	All Mutants (mmm:ss)	5% Sampled Mutants (mm:ss) Min.	Max.	Avg.	(Pct.)
TimeMoney	7:13	0:54	0:57	0:55	(12.89%)
JDepend	3:02	0:31	0:33	0:32	(17.66%)
JTopas	34:59	1:01	1:12	1:03	(3.02%)
Barbecue	7:35	2:42	2:56	2:46	(36.62%)
Mime4J	181:20	6:44	9:24	8:09	(4.50%)
Jaxen	48:21	2:49	4:03	3:15	(6.75%)
XStream	132:02	4:37	9:49	6:07	(4.64%)
XmlSecurity	53:04	3:17	4:03	3:46	(7.10%)
CommonsLang	74:35	5:12	6:51	6:01	(8.08%)
JodaTime	196:28	12:28	19:03	14:57	(7.61%)
JMeter	57:32	3:42	5:26	4:28	(7.77%)
Avg.	72:22	-	-	4:43	(6.54%)

dant nor discordant. Kendall's τ is calculated as the ratio of difference between the number of concordant and discordant pairs over total number of pairs. In this paper we use τ_b, which has a more complex computation because it takes ties into consideration.

We calculate Kendall's τ_b for each triple (P, S, r), following the same procedure as for R^2. Similar with the R^2 measure, we show the τ_b measure for all the strategies on all subjects with the sampling ratio of 5% in the right part of Table III. We also show Kendall's τ_b values for CommonsLang subject, all sampling strategies, and all sampling ratios in Figure 2b. The plots for the other examples look similar and are available on the project webpage [39]. Across all the subjects, all strategies, and all ratios, the minimal value for τ_b in our study was 0.766 (for JDepend).

Considering Table III and Figure 2b, we can draw similar conclusions as from the R^2 correlation measures. First, all sampling strategies provide very similar result for Kendall's τ. In addition, S_{meth} slightly outperforms S_{base} and S_{mop} for all 11 subjects. Second, all the values are very high, which indicates very strong agreement between rankings. The results for Spearman's ρ show even stronger agreement (details can be found on the project webpage [39]). Based on our study, we believe that the comparison of test suites or testing techniques can be done using sampling mutation.

3) Savings Obtained by Mutation Sampling: Table IV shows the selected mutation testing time for all the mutants generated by Javalanche (recall that Javalanche uses operator-based selection), and the sampling mutation testing time for the sampling ratio of 5% and our S_{meth} strategy. Column "Subject" lists the subjects, column "All Mutants" shows the mutant generation and execution times for all the mutants generated by Javalanche, and columns 3-5 list the minimum/maximum/average mutant generation and execution times for the sampling mutation with the sampling ratio of 5% across 20 sampling runs. In column 6 ("Pct."), we also show the ratio of the sampling mutation testing time over the selected mutation testing time. Note that we include the mutant generation time of all selected mutants for both selected mutation and sampling mutation, because our current implementation requires Javalanche to generate all the mutants before sampling. The results show that the sampling mutation testing time, with sampling ratio of 5%, is close to 5% of the selected mutation testing time. We further noticed that the sampling mutation testing time on small subjects tends to be longer than expected 5% of selected mutation time, because for small subjects the tool setup time and the mutant generation time (rather than the mutation execution time) can dominate the total mutation testing time. However, for the seven larger subjects, the tool setup time and the mutant generation time take insignificant time compared to the total mutation testing time, leading to sampling mutation time from 4.50% to 8.08% of the selected mutation time. On average across all the 11 subjects, the sampling mutation testing time is less than 5 minutes; in contrast, the original Javalanche time is much more and exceeds 70 minutes.

D. Below 5%

Our experimental results show that it is possible to greatly reduce the number of mutants (e.g., sampling only 5% mutants) while still preserving the mutation score. However, it was not clear whether we can use sampling ratio below 5%. Thus, we additionally collected the experimental results for sampling fewer than 5% mutants. Table V shows the results for the S_{base} and S_{meth} strategies. The detailed results for all the 8 strategies can be found online [39]. In the table, Column 1 lists all the studied sampling ratios, columns 2-4 list the average selected mutation scores for adequate test suites as well as the average R^2 and the τ correlation values for

TABLE V: Results of sampling below 5% of selected mutants

Ra.	Base			Meth		
	MS.	R^2	τ	MS.	R^2	τ
0.5%	92.24	0.806	0.716	92.02	0.803	0.722
1.0%	96.55	0.896	0.792	96.32	0.905	0.799
1.5%	97.27	0.926	0.819	97.71	0.939	0.837
2.0%	98.21	0.944	0.843	98.48	0.952	0.858
2.5%	98.68	0.955	0.859	98.82	0.964	0.872
3.0%	98.94	0.964	0.874	99.00	0.973	0.885
3.5%	99.07	0.968	0.877	99.22	0.977	0.895
4.0%	99.22	0.974	0.888	99.34	0.978	0.899
4.5%	99.38	0.974	0.893	99.46	0.982	0.907

inadequate test suites by the S_{base} strategy across all subjects. Similarly, columns 5-7 list the corresponding results for the S_{meth} strategy.

The results show that it is possible to have a fairly reliable mutation score even when sampling fewer than 5% mutants. However, by the "rule of 99%" [3], which would require the sampling mutation score to be 99% or higher, the ratio of 3-3.5% is on the borderline for our set of programs and tests and may not generalize to other programs and tests. In the future, we plan to evaluate whether advanced techniques (e.g., search-based mutant selection [13], [43]) could achieve even smaller sampling ratios. In addition, the results show that S_{meth} outperforms S_{base} in terms of all the three metrics with sampling ratio of greater than 1%, further demonstrating the benefits of our proposed sampling based on program elements.

E. Threats to Validity

Threats to construct validity. The main threat to construct validity for our study is the set of metrics used to evaluate the mutant sampling strategies. To reduce this threat, we use two widely used metrics, the mutation score metric for adequate test suites [3], [4], [7], [20], [22] and the correlation analysis for non-adequate test suites [5]. Our study still inherits a major threat to construct validity: as in those previous studies, we considered all mutants not killed by the original test pool to be equivalent due to the lack of precise techniques for detecting equivalent mutants.

Threats to internal validity. The main threat to internal validity is the potential faults in the implementation of our sampling strategies or in our data analysis. To reduce this threat, the first two authors carefully reviewed all the code for mutant sampling and data analysis during the study.

Threats to external validity. The main threat to external validity is that our results from this study may not generalize to other contexts, including programs, tests, and mutants. To reduce this threat, we select 11 real-world Java programs with various sizes (from 2681 to 36910 lines of code) from various application domains. Note that our study includes more programs than any previous study on selective mutation testing for sequential code [3]–[5], [7], [20], [22]. In addition, the 11 programs used in our study are one to two orders of magnitude larger than programs used in similar previous studies.

IV. Related Work

Mutation testing was first proposed by Hamlet [1] and DeMillo et al. [2]. Since then, due to its cost and effectiveness, a large amount of research has been dedicated to reducing the cost of mutation testing and exploring the application of mutation testing. In this section, we first discuss the related work in reducing the cost of mutation testing. Then we discuss the existing applications of mutation testing. More details about existing work on mutation testing can be found in a recent survey by Jia and Harman [8].

A. Reducing The Cost of Mutation Testing

There are mainly three ways to reduce the cost of mutation testing – selecting a subset of all mutants (Section IV-A1), executing each mutant partially (Section IV-A2), and optimizing mutation generation and execution (Section IV-A3).

1) Selective Mutation Testing: Selective mutation testing, which was first proposed by Mathur [44], aims to select a representative subset of all mutants that can still achieve similar results as all mutants. Since its first proposal, a large amount of research effort has been put on operator-based mutant selection, which only generates mutants based on a subset of mutation operators. Wong and Mathur [22] investigated selection of two mutation operators among all the 22 mutation operators in Mothra [45], and found that mutants generated with the selected two mutation operators can achieve similar mutation testing results as all the 22 mutation operators. Offutt et al. [3], [20] then proposed five mutation operators, named *sufficient mutation operators*, through a set of experimental studies to ensure that the selected set of mutants achieves almost the same results as the entire mutant set. Barbosa et al. [4] proposed six guidelines to determine a set of 10 mutation operators. Namin et al. [5] used rigorous statistical analysis to determine 28 mutation operators from all the 108 mutation operators of Proteum [19]. Recently, Gligoric et al. [9] also investigated operator-based mutant selection for concurrent mutation operators.

In contrast to operator-based mutant selection, random mutant selection was less widely researched. The idea of random mutant selection was first proposed by Acree et al. [46] and Budd [47]. Wong and Mathur [22] empirically studied randomly selecting x% of *all mutants* generated by Mothra. Since then, researchers mainly used random mutant selection as a control technique when evaluating operator-based mutant selection [4]. However, a recent study by Zhang et al. [7] demonstrated that random mutant selection can be equally effective with operator-based selection when selecting the same number of mutants. The study used larger subjects (7 C programs from Siemens Suite [48] ranging from 137 to 513 lines of code) and more mutation operators (108 mutation operators implemented by Proteum [19]) than previous studies. However, the studied subjects are still relatively small. In addition, they did not demonstrate that random mutant selection can further be applied to *operator-based selected mutants*. In contrast, our study is the first to show that random mutant selection can be applied together with operator-based selected mutants, e.g., even sampling 5% of operator-based generated mutants can still achieve precise mutation score. In addition, we used 11 real-world Java programs from 2681 to 36910 lines

of code, which are orders of magnitude larger than subject programs used in previous studies on selective mutation [3]–[5], [7], [22].

2) Weakened Mutation Testing: Weakened mutation testing, which was first proposed by Howden [49], aims to provide a more efficient way to determine whether a mutant is killed by a test. More precisely, the traditional mutation testing considers a mutant as killed only when a test generates different final results for the mutant and the original program. On the contrary, the first work of weakened mutation testing, *weak mutation* [49], considers a mutant as *weakly* killed when a test triggers a different program *internal state* when executing the mutated statement on the mutant and on the original program. However, this approach is imprecise because some internal states triggered by a test may not be propagated to the final result. To better balance the cost and precision, Woodward and Halewood [50] proposed *firm mutation*, which is a spectrum of techniques between weak and strong mutation. Offutt and Lee [51] experimentally investigated the relationships between weak mutation and strong mutation. Our study is orthogonal to this line of work; our study aims to reduce the number of mutants while weakened mutation testing aims to reduce the execution cost for each mutant.

3) Optimized Mutation Testing: Optimized mutation testing aims to explore efficient ways to generate, compile, and execute mutants. DeMillo et al. [52] extended a compiler to compile all mutants at once to reduce the cost of generating and compiling a large number of mutants. Similarly, Untch et al. [53] proposed the schema-based mutation approach, which generates one meta-mutant that encodes all the mutants and can be compiled by a traditional compiler. Researchers have also investigated various ways to run mutants in parallel to speed up mutation testing [54], [55]. Recently, we proposed approaches inspired by *regression testing* [56]–[58] to optimize the test execution for each mutant [10], [37]. More specifically, inspired by regression test selection, we proposed ReMT [10] to incrementally collect mutation testing results based on old mutation results; inspired by regression test prioritization and reduction, we proposed FaMT [37] to prioritize and reduce tests for each mutant to collect mutation testing results faster. Our sampling strategies, which further sample mutants over operator-based selected mutants, aim to reduce the cost of mutation testing at a different dimension.

B. Applications of Mutation Testing

Mutation testing was initially used to evaluate the quality of test suites. After achieving the mutation score for the test suite evaluation, the user can improve the quality of the test suite manually. Researchers have also proposed techniques that automatically generate tests to kill mutants. DeMillo and Offutt first proposed constraint-based testing (CBT) [59] to generate tests (each killing one mutant) based on static symbolic evaluation. Offutt et al. [15] further proposed the dynamic domain reduction technique to further refine CBT. Recently, researchers proposed more solutions for this area due to the growing computing power. Fraser and Zeller [60] used search-based software testing (SBST) to generate tests for mutant killing. Zhang et al. [16] and Papadakis et al. [17] used dynamic symbolic execution (DSE) to generate tests for mutant killing. Harman et al. [14] combined SBST and DSE techniques to generate tests that kill multiple mutants at a time. Our sampling strategies may make these test generation techniques more efficient by only generating tests that kill a sample of mutants.

Mutants generated by mutation testing can also be used to simulate real program faults to evaluate software testing techniques. The advantage compared with seeded or real faults is that mutation faults can be systematically generated, making the generation replicable and sufficient for statistical analysis. Andrews et al. [11], [12] empirically showed that the mutants, which are generated by selective operators, simulate real faults better than manually seeded faults, and is appropriated for evaluating testing techniques. Do et al. [61] also showed that it is suitable to use mutation faults to evaluate regression testing techniques. Recently, Zhang et al. [62] showed that mutation faults can be used to simulate the impacts of real faults. Thus, more and more software testing techniques are evaluated using mutation faults [41], [58], [63]–[65]. Currently, testing techniques are mainly evaluated using an arbitrary set of mutants due to the large number of mutants. Our study establishes rules for evaluating testing techniques – if technique t outperforms technique t' on 5% sampled mutants, we can predict with high confidence that t is better than t' on all mutants.

V. Conclusions and Future Work

This paper reports an empirical study to answer an important question for selective mutation testing: Can random mutant sampling be applied on top of operator-based mutant selection to further reduce the cost of mutation testing? We evaluate that question for various sampling strategies, for adequate and non-adequate test suites, and on 11 real-world Java programs. Surprisingly, the empirical results show that sampling only 5% of mutants generated by operator-based selection can still provide a highly precise mutation score. In addition, the study shows that our newly proposed random mutant sampling strategies based on program elements can be more effective than strategies based on mutation operators. Furthermore, the study shows that mutant sampling is more beneficial for larger programs, indicating a promising future for applying mutant sampling to larger projects.

In the future, we plan to consider more sophisticated mutant sampling based on dynamic test behavior (e.g., we may sample more mutants on some critical paths) or on combinations of random mutant sampling and search-based mutant selection.

Acknowledgments

We would like to thank Nathan Hirtz and Douglas Simpson for help with statistical analysis, and David Schuler for help with Javalanche. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CNS-0958231, CNS-0958199, CCF-0845628, and CCF-0746856.

REFERENCES

[1] R. G. Hamlet, "Testing programs with the aid of a compiler," *TSE*, vol. 3, 1977.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, 1978.

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

[4] 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.

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

[6] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proc. of ISSTA*, 2009.

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

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

[9] M. Gligoric, L. Zhang, C. Pereira, and G. Pokam, "Selective mutation testing for concurrent code," in *Proc. of ISSTA*, 2013, pp. 224–234.

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

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

[12] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[13] Y. Jia and M. Harman, "Higher order mutation testing," *IST*, vol. 51, no. 10, pp. 1379–1393, 2009.

[14] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. of FSE*. ACM, 2011, pp. 212–222.

[15] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Software-Practice and Experience*, vol. 29, no. 2, pp. 167–194, 1999.

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

[17] M. Papadakis, N. Malevris, and M. Kallia, "Towards automating the generation of mutation tests," in *Proc. of AST*, 2010, pp. 111–118.

[18] K. Pan, X. Wu, and T. Xie, "Automatic test generation for mutation testing on database applications," in *Proc. of AST*, 2013, p. to appear.

[19] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, "Proteum/im 2.0: An integrated mutation testing environment," in *Mutation testing for the new century*. Springer, 2001, pp. 91–101.

[20] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proc. of ICSE*, 1993.

[21] P. Ammann and J. Offutt, *Introduction to Software Testing*, 2008.

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

[23] "Time and Money home," http://timeandmoney.sourceforge.net/.

[24] "JDepend home," http://clarkware.com/software/JDepend.html.

[25] "JTopas home," http://jtopas.sourceforge.net/jtopas/index.html.

[26] "Barbecue home," http://barbecue.sourceforge.net/.

[27] "Apache Mime4J home," http://james.apache.org/mime4j/.

[28] "Jaxen home," http://jaxen.codehaus.org/.

[29] "XStream home," http://xstream.codehaus.org/.

[30] "Santuario home," http://santuario.apache.org/.

[31] "Apache Commons home," http://commons.apache.org/proper/commons-lang/.

[32] "Joda Time home," http://joda-time.sourceforge.net/.

[33] "Apache JMeter home," http://jmeter.apache.org/.

[34] D. Schuler and A. Zeller, "Javalanche: efficient mutation testing for Java," in *Proc. of FSE*, 2009, pp. 297–298.

[35] S. Zhang, C. Zhang, and M. D. Ernst, "Automated documentation inference to explain failed tests," in *Proc. of ASE*. IEEE, 2011, pp. 63–72.

[36] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proc. ICSE*, 2013, pp. 192–201.

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

[38] "JavaSourceMetric home," http://sourceforge.net/projects/jsourcemetric/.

[39] "Our project," https://webspace.utexas.edu/~lz3548/ase13support.html.

[40] M. Staats, G. Gay, and M. P. Heimdahl, "Automated oracle creation support, or: How i learned to stop worrying about fault propagation and love mutation testing," in *Proc. of ICSE*, 2012, pp. 870–880.

[41] L. Briand, Y. Labiche, and Y. Wang, "Using simulation to empirically investigate test coverage criteria based on statechart," in *Proc. of ICSE*, 2004, pp. 86–95.

[42] R. Sharma, M. Gligoric, A. Arcuri, G. Fraser, and D. Marinov, "Testing container classes: Random or systematic?" in *Proc. of FASE*, 2011, pp. 262–277.

[43] W. B. Langdon, M. Harman, and Y. Jia, "Multi objective higher order mutation testing with genetic programming," in *Proc. of TAIC PART*, 2009, pp. 21–29.

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

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

[46] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Mutation analysis." DTIC Document, Tech. Rep., 1979.

[47] T. A. Budd, "Mutation analysis of program test data[ph. d. thesis]," 1980.

[48] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

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

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

[51] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE TSE*, vol. 20, no. 5, pp. 337–344, 1994.

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

[53] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. of ISSTA*, 1993, pp. 139–148.

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

[55] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a MIMD computer," in *Proc. of International Conference on Parallel Processing*, 1992, pp. 257–266.

[56] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE TSE*, vol. 27, no. 10, pp. 929–948, 2001.

[57] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proc. of FSE*, vol. 29, no. 6, 2004, pp. 241–251.

[58] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, "On-demand test suite reduction," in *Proc. of ICSE*, 2012, pp. 738–748.

[59] R. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE TSE*, vol. 17, no. 9, pp. 900–910, 1991.

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

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

[62] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proc. of OOPSLA*, 2013, p. to appear.

[63] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Proc. of FASE*, 2012, pp. 409–424.

[64] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing JUnit test cases," *IEEE TSE*, vol. 38, no. 6, pp. 1258–1275, 2012.

[65] M. Gligoric, A. Groce, C. Zhang, R. Sharma, A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *Proc. of ISSTA*, 2013, pp. 302–313.