An Empirical Comparison of Mutant Selection Approaches

Rahul Gopinath
Oregon State University
gopinath@eecs.orst.edu

Amin Alipour
Oregon State University
alipour@eecs.orst.edu

Iftekhar Ahmed
Oregon State University
ahmedi@eecs.orst.edu

Carlos Jensen
Oregon State University
cjensen@eecs.orst.edu

Alex Groce
Oregon State University
agroce@gmail.com

ABSTRACT

Mutation analysis is a well-known method for measuring the quality of test suites. However, it is computationally intensive compared to other measures, which makes it hard to use in practice. Choosing a smaller subset of mutations to run is a simple approach that can alleviate this problem. Mutation operator selection has been heavily researched. Recently, researchers have found that sampling mutants can achieve accuracy and mutant reduction similar to operator selection. However, the empirical support for these conclusions has been limited, due to the small number of subject programs investigated. The best sampling technique is also an open problem.

Our research compares a large number of sampling and operator selection criteria based on their ability to predict the full mutation score as well as the consistency of mutation reduction ratios achieved. Our results can be used to choose an appropriate mutation reduction technique by the reduction and level of fidelity to full mutation results required.

We find that all sampling approaches perform better than operator selection methods, when considering ability to predict the full mutation score as well as the consistency of mutation reduction ratios achieved.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging Testing Tools

General Terms

Measurement, Verification

Keywords

Test frameworks, empirical analysis, mutation operators

1. INTRODUCTION

Mutation analysis is a method for evaluating the quality of test suites. It involves producing a family of *mutants*, programs with small differences from the original program, and evaluating the effectiveness of test suites against these mutants [15, 2]. Previous research [3] suggests that mutations thus introduced behave in a fashion similar to real faults, with respect to the difficulty of detection.

One of the impediments to wider adoption of mutation analysis is its high computational cost. The set of simple mutants for even a moderate sized program can be very large, making mutation analysis prohibitively time consuming.

A major strain of research into cost-reduction of mutation analysis is to choose a smaller, representative, set of mutants [23, 14] — often called the *do fewer* approach. This approach can be generally divided into selective strategies and sampling strategies.

Selective mutation strategies attempt to select a representative subset of mutation *operators* based on heuristics and statistical analysis, and apply this subset of operators to generate mutants instead of applying the whole set of mutation operators [19, 27]. Recent work suggests that using statement deletion alone can be an effective approach [27].

Sampling strategies seek to randomly select a set of representative mutants. This was investigated first by Acree [1] and Budd [6], who proposed using only $x\%$ of all mutants produced. Wong and Mathur found that random sampling with ratio as low as 10% could provide accurate results [14].

Recent work [31, 30] has investigated the relative merits of random sampling strategies and operator selection. Random sampling can perform as well as or better than operator selection, in these studies; and a strategy of either sampling based on program elements or one combining both program element-based sampling and operator selection was best.

However, as pointed out by Zhang et al. [30], the field has a serious lacuna in large scale research, both in the size of the programs studied, and in the number and diversity of programs, which reduces our confidence in all results. This is true for selective mutation studies, sampling studies, and also for comparatively newer studies that attempt to combine methods. This is particularly worrisome if mutation analysis is to gain wider acceptance among testing professionals. Further, quite a few of the influential studies [22, 20, 28, 17] were conducted on older programming languages such as Fortran, with operators specific to the language, and are not directly applicable to newer languages such as Java. Finally, with bytecode based mutation engines like PIT [7] and Javalanche [25] (a *do faster* approach for eliminating the compilation step to gain execution speed), operators based on source code modification are no longer applicable, and their equivalents in bytecode need to be identified and compared with other approaches. A detailed discussion of these issues can be found in Section 2.

We have attempted to rectify this situation with a large scale study of real world programs. As detailed in Section 3, we sample 188 Java programs from Github, with size ranging from 50 LOC to 100 KLOC, which should allow widely applicable statistical inferences to be made. We provide the results from running both the original project's (presumably human-generated) test suites and also automatically generated test suites using random testing, increasing the applicability of our findings to both testers in traditional development who are interested in applying mutation analysis to suite evaluation and to researchers who seek to compare test suites quickly to evaluate testing techniques.

One note about our, and other researchers, evaluation criteria is in order. We, like others, use the ability of a sample to correlate well to the score over all mutants as a measure of sampling quality. The underlying reason for this is that, while many mutants have little ability to distinguish between good and bad test suites, some smaller set of mutants show subtle deficiencies in suites that are not clear from, say, code coverage. The full set of mutants obviously contains these mutants, and correlating with (rather than matching) the full mutation score is a strong indicator that many of these discriminatory mutants have been preserved in a sample.

Section 3 discusses the sampling and operator selection strategies we study in detail, and the results of our experiment are given in Section 5, followed by detailed discussion of what these results imply in Section 6. The specific contributions of this paper are:

- Our study is the largest so far in terms of both the size of programs involved (50 LOC to 100 KLOC), and the number of programs analyzed (188 unique open source projects) for mutant reduction strategies. This allows for stronger and more widely applicable conclusions about effectiveness.

- We compare a much larger number of mutant reduction strategies than previous studies. We compare with respect to the predictive power (correlation with the full mutation score), expected mutant reduction, and the stability of the amount of mutant reduction expected. Our sampling strategies include some that were suggested by other researchers, as well as a few novel strategies not previously considered.

- Our recommendations are useful for both testing researchers who might be comparing automatically generated test suites, and software testers in real world working to produce and evaluate manual test suites.

- Most importantly, we find that random sampling strategies perform better than operator selection strategies in consistently predicting the final mutation score while obtaining stable computational reduction.

2. RELATED WORK

There are several approaches to reducing the cost of mutation analysis. These were categorized by Offutt and Untch [23] into three approaches: do *fewer*, *smarter*, and *faster*. The *do fewer* approaches include selective mutation and mutant sampling, while weak mutation, parallelization of mutation analysis, and space/time trade-offs are grouped under the umbrella of *do smarter*. Finally *do faster* approaches include mutant schema generation methods, code patching etc.

The idea of using a subset of mutants was conceived along with mutation analysis itself. Budd [6] and Acree [1] showed that even 10% sampling can achieve 99% accuracy for the final score. The idea was further investigated by Mathur [16], Wong et al. [29, 28], and Offutt et al. [22] using the Mothra [9] mutation operators for Fortran. Mathur [16, 28] suggested constrained mutation where only two operators were used.

A number of studies in the past have looked at the relative merits of operator selection and random sampling criteria. Wong et al. [28] compared x% selection of each mutant type with operator selection using just two mutation operators, and found that both achieved similar accuracy and reduction (80%).

Mresa et al.[17] used the cost of detection of mutants as a means of selection to define a set of operators. They found that if very high mutation score (close to 100%) is required, x% selective mutation is better than operator selection, and, conversely, for lower scores, operator selection would be better if the cost of mutants is considered.

Zhang et al. [32] compared operator based mutant selection techniques to random mutant sampling. They found that none of the selection techniques are superior to random sampling, with the same number of mutants. They also found that uniform sampling of mutations is more effective for larger subjects compared to equal sampling of mutation operators and the reverse is true for smaller subjects.

Recently, Zhang et al. [30] confirmed that sampling as few as 5% of mutants was sufficient for a very high correlation (99%) with full mutation score, while sampling even fewer mutants has good potential for retaining a high accuracy of prediction. They investigated eight sampling strategies on top of operator-based mutant selection and found that sampling strategies based on program components (methods in particular) performed best.

Some studies have tried to find the set of *sufficient mutation operators* that reduce the cost of mutation but maintain correlation with the full mutation score. Offutt et al. [22] suggested an n-selective approach with step-by-step removal of operators with most numerous mutations. Barbosa et al.[5] provided a set of guidelines for selecting such mutation operators. Namin et al.[18, 26] formulated the problem as a variable reduction problem, and found that just 28 out of 108 operators in Proteum were sufficient.

Using only the statement deletion operator was first suggested by Untch [27], who found that it had the highest correlation ($R^2 = 0.97$) with the full mutation score compared to other operator selection methods, while generating the smallest number of mutants. This was further reinforced by Deng et al. [10] who defined deletion for different language elements, and found that an accuracy of 92% is achieved while reducing the number of mutants by 80%.

Our work is most closely related to that of Zhang et al. [30]. We extend the scope of their study with a much wider range of mutation approaches and base our results on a much larger set of real-world projects.

3. METHODOLOGY

Our selection of programs was driven by a few overriding concerns. Our primary requirement was that our results had to be as widely applicable as possible for real-world programs. Secondly, we strived for a statistically significant

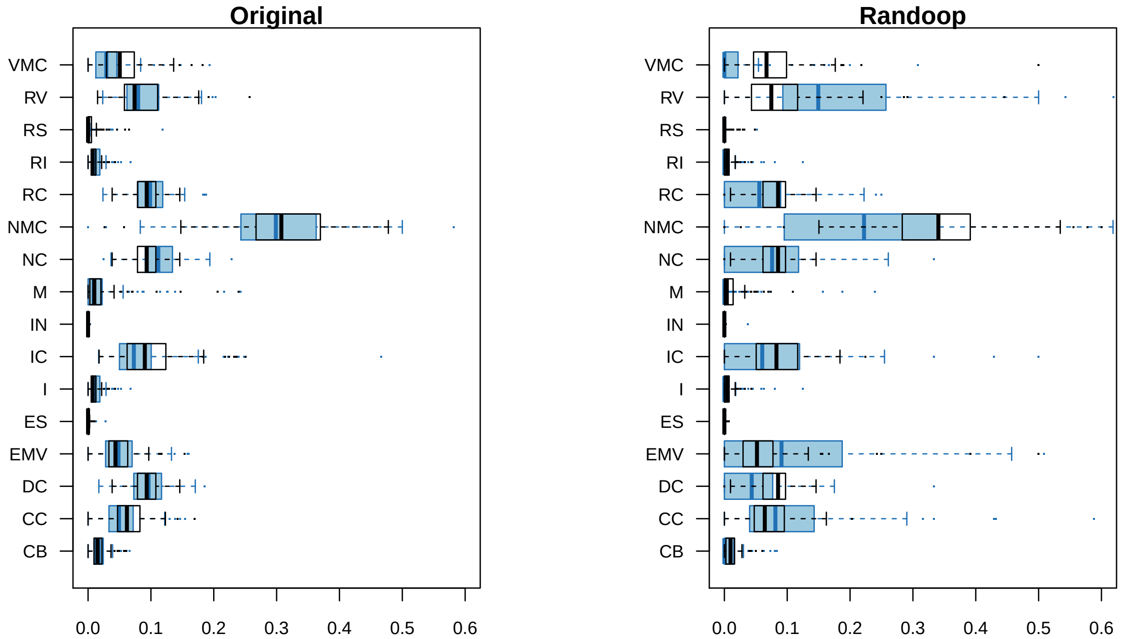

Figure 1: The relative contribution of mutation operators in terms of fraction of mutants produced and detected. The produced mutants are shown with black outline while detected mutants have blue border and shade.

result, reducing the number of variables present in the experiments. For this reason we chose a random sample of Java projects from Github [11] that use the popular maven [4] build system, following the methodology we used previously in an ICSE 2014 paper [13]. As in that work, we utilize two kinds of of test suites: the original test suites present in a limited number of projects, and automatically generated test suites from Randoop [24]. From this set of test suites, After eliminating those that took too long to finish, we sampled 107 suites each for both original and Randoop generated test suites, Since there was some overlap between projects with original test suites, and those with generated test suites, we had 81 unique projects each in both sets. The remaining 26 projects had both flavors of test suites.

We ran mutation analysis on this set of projects using PIT[7]. However, since the operators provided by PIT are limited, we extended PIT to provide newer operators that are similar to operators provided by other mutation systems. The set of operators that we used is provided in Table 1. Figure 1 shows the distribution of mutants and the relative detection rates of each mutation operator for our projects. In the figure, there are two box-plots corresponding to each mutation operator on the Y-axis. The black box represents the relative frequency of the particular mutation operator in the total mutants *produced*. For example, for projects in the (Randoop) generated set, the operator NMC contributed about 34% of the total number of mutants produced while RV contributed about 7%. Similarly, the blue shaded box-plot represents the relative frequency of particular mutation operator in the total mutants *detected*. For example, NMC contribution in the detected mutants was 23% while that of RV was 15%.

For a detailed description of each mutation operator, please refer to PIT documentation [8]. PIT was also modified to provide random selection of mutants according to various criteria. Finally, to remove random noise, each criteria described was run four times, and the results averaged to produce the final result.

IN	Remove negative sign from numbers
RV	Mutate return values
M	Mutate arithmetic operators
VMC	Remove void method calls
NC	Negate conditional statements
CB	Modify boundaries in logical conditions
I	Modify increment and decrement statements
NMC	Remove non-void method calls, returning default value
CC	Replace constructor calls, returning null
IC	Replace inline constants with default value
RI	Remove increment and decrement statements
EMV	Replace member variable assignments with default value
ES	Modify switch statements
RS	Replace switch labels with default (thus removing them)
RC	Replace boolean conditions with true
NC	Replace boolean conditions with false

Table 1: PIT Mutation Operators (We have used abbreviations instead of operator names.)

3.1 Sampling Criteria

We used several different sampling criteria, some of which has been suggested in the literature before, some which are variants of previously suggested criteria, and a few novel ones. For each sampling criteria, we sampled mutants on a decreasing power scale, sampling *1/2, 1/4, 1/8, 1/16, 1/32, 1/64* of the total mutants.

3.1.1 x% selection

The simplest sampling approach consisted of using x% selection as suggested by Budd [6]. In this criteria, we choose a specific fraction of the complete set of mutants. This criteria also serves as a baseline for verifying the effectiveness of other criteria.

3.1.2 Sampling over program elements and variants

Following the suggestion of Zhang et al. [30], we extended x% selection criteria to sample from within different program elements. We sampled in increasing order of scope, — *line*, *method* and *class* (*project* scope is just x% selection).

We used the formula by Zhang et al. [30],

$$sample(x) = \lfloor x + random(0..1) \rfloor$$

to correctly sample decimal numbers. Next, we slightly modified this criteria, and instead of *sample()* which uses probability to manage the decimal numbers, we applied *round()* to obtain the nearest whole number, which was used as the number of samples to be chosen from the population. The new strategy ignores program elements with small number of mutants. This strategy can potentially guide mutation generation toward more complex program elements. That is, if the number of mutants dropped below a threshold determined by the mutant reduction ratio, that program element would not contribute any mutants to the final result. Conceptually the idea is that mutants of more complex code have more discriminatory power (and may correspond better to real faults — though we do not evaluate this concept).

Next, we explored in the other direction, by forcing the program elements to return at least one element by using *ceil()*. That is, irrespective of the fraction being sampled, the simplest elements always contributed at least one mutant to the total sample, ensuring coverage but also giving priority to complex elements.

3.1.3 Lines per element and variants

Our previous research [13] found that statement coverage was highly correlated with mutation score for a project. This immediately suggests that perhaps choosing one mutant per line may be sufficient to achieve a close approximation of the final mutation score. Further, statement deletion has been researched previously [27, 10] and has been found to reduce the number of mutants well, with negligible decrease in effectiveness. This also provides a nice comparison with similar numbers of mutants between operator selection and random sampling of mutants.

We extended sampling to levels coarser than methods, i.e. method, class, and project. We sampled n mutants from a method (class or project), where n is the number of statements in the method (class or project).

This provided a test of the hypothesis of how important a mutation sampling's relationship to simple code coverage is. This was extended to class and project scope also.

For the first variant, we applied the x% sampling to the result of the first sampling based on line counts. That is, in the case of methods, we first selected *linecount* mutants each from each method, and applied the x% selection to this result. For example, for 1/2 sampling on methods if one method had 10 lines and another had 20, we sampled 10 mutants from the first, and 20 from the second, and from the combined 30 mutants, we sampled 15. For the second variant, we did the sampling in one go, where we chose *linecount/x* number of mutants from each program element. That is, in the previous example, only 5 mutants from the first and 10 mutants from the second method would be chosen.

3.1.4 One per element

Another simple strategy of sampling we experimented with is to just choose one mutant per program element, in the order *line*, *method*, *class*.

3.1.5 x% selection per operator

This strategy, first suggested by Wong et al. [28] samples an equal percentage of mutants from each operator.

3.2 Operator Selection

For selective methods, we tried mutation operators suggested by Wong et al. [28], Offutt et al. [20, 10], and, Namin et al. [26]. Since Javalanche [25] utilized operator selection mechanisms, we also compared the Javalanche operators for operator selection. Note that all of these techniques except Javalanche have targeted C programs. Thus, some of these operators may be sensible in C but not in Java. For example, deletion of return statement is tolerated in C, not in Java. Moreover, there were a few operators that were not supported by the PIT, and could not be implemented easily (as mentioned below).

3.2.1 Constrained Mutation

Wong et al. [28]. They selected ROR and ABS from Mothra for mutation analysis. We used operators CB and NC to model ROR operator. There are no comparable operators to ABS in PIT.

3.2.2 E-Selective

Offut et al. [20]. They selected ABS, UOI, LCR, AOR, ROR from Mothra operators. We have used IN, M, CB, and NC. PIT does not have any operator comparable to LCR.

3.2.3 Javalanche

Javalanche [25]. Javalanche uses Negate Jump Condition, Omit Method Call, Replace Arithmetic Operator, and Replace Numerical Constant operators. We have used NC, VMC, NMC, M, ICand EMVto model them.

3.2.4 Variable Reduction

Namin et al. [26]. They have tried to reduce the mutation operators of Proteum analysis tool which is for C programs. They suggest 28 operators which many of them are not applicable in Java, and some not in PIT. We used by IN, M, Iand NC.

3.2.5 N-selection

Offut et al. [22]. They suggested removal of n most numerous operators. In our experiment, the order of operators was NMC, NC, RC, DC, RV, IC, CC, EMV, VMC, M, CB, I, RI, RS, ES, and IN. We discarded one at each step and evaluated the effectiveness at each n.

3.2.6 Statement Deletion

The basic statement deletion was modeled on the work by Deng et al. [10]. The operations on single statements were modeled using VMC, NMC, CC, EMV, and RI for simple statements, and using RC for control structures. RC replaces boolean conditions with *false*, resulting in removal of the conditional block. The operator for return values was modeled using RV, which is similar. The operators for *while*, *for*, and *if* statements were modeled using DC, which replaced the boolean condition with *true*, which removed the effect of conditional. The *switch* statement deletion was modeled using RS which replaced the first 100 labels with a *default* label, resulting in the switch element being deleted. Due to the constraints of the architecture of PIT only the first 100 labels were replaced. Deleting *try/catch* was not necessary at bytecode level. Finally, to get an accurate estimation of the effect of true statement deletion, we grouped the mutants by line, and considered each line a single virtual mutant (that is, the total number of mutants is equal to the total number of lines). Further, killing any mutant from a line resulted in marking the virtual mutant for that line as killed. This gave us the mutation score for the virtual statement deletion operator. We note that the approximation of simple statement deletion, especially when arithmetic operators are involved, is not complete. However, as seen in Figure 1, the number of mutants produced by M operator is very small. We also note that our approximation does not account for the increase in ease of detection when multiple mutations are combined together due to the coupling effect. This also means that not all the lines may be mutated ,since there may be no applicable operators. However, given the constraints of a bytecode based mutation system, we believe that our procedure is reasonable.

4. ANALYSIS

For the purposes of comparing between the criteria, we computed the linear correlation between sampled mutation score and full mutation score. Since for the baseline sampling criteria, no mutants sampled implies none generated, our model was

$$\mu\{M_{all}|M_{reduced}\} = \beta_1 \times M_{reduced}$$

where M_{all} is the mutation score of original mutation analysis and $M_{reduced}$ is the mutation score of the sample.

We reported the correlation for this linear regression. We also report the expected multiplier factor β_1 for the sampled mutation score. This factor determines the ease of detecting mutants in the sampling criteria. That is, if the multiplier factor is small, then the sample mutation score was larger than the full mutation score, and hence it was easier to detect mutants in the sample.

Next, for each criterion, we computed the reduction of mutants given by the ratio between sampled mutants and total number of mutants. We also compute standard deviation of the mutant reduction, which determines the consistency of results. A large standard deviation suggests that the mutant reduction ratio achieved is very much dependent on the project. Therefore, as a heuristic, we suggest looking at only those methods which have a mean reduction ratio at least twice the standard deviation.

A few projects did not return any results for some large reduction factors. Thus the number of valid results from projects is also collected in the result tables.

5. RESULTS

Our results for random sampling over program elements using $sample(x)$ are given in Table 2. The result of the variation using $round(x)$ is given in Table 3 and that using $ceil(x)$ is given in Table 4. The first column in each table provides the scope of sampling: per project, per class, per method, or per line. The second column ($\div$) provides the fraction involved. The third column contains the R^2 value obtained between the mutation score of the sampling criteria used and the full mutation score. The fourth column (μ_{red}) the mutant reduction factor, which is the average of total mutants divided by the number of mutants sampled. The fifth column (σ_{red}) is the standard deviation of the same. As we explained in the analysis, the sixth column contains the valid responses obtained. The seventh column (β_1) is the multiplier factor (coefficient reported by the regression.)

Table 5 provides the results for line count criteria where the sampling was conducted after combining program elements, and Table 6 contains the results for line count criteria where the sampling was conducted inside each program element. Similarly Table 8 contains the results for the criteria of one mutant per program element.

Finally, Table 9 contains the results when only a single operator is used, and Table 10 contains the results for specific operator selection criteria. The first columns in these two tables contain the particular operator or operator selection applied, and the fraction column ($\div$) is absent. The remaining columns are same as previous tables. The TSDL row is the result of applying only the higher order mutant equivalent to *statement deletion.* All the *N-sel* rows are the results of N-selective operator selection strategies [19], using the order of operators from Figure 1. The complete data (only for original test suites) is visualized in Figure 3.

Our data will be available for replication on publication [12]

6. DISCUSSION

From our data, we see that for original and generated test suites, the mutant reduction ratio and standard deviation were similar for stable strategies (where the standard deviation of the reduction achieved was less than half of the reduction achieved). This suggests that our analysis is equally applicable for both original test suites and generated ones.

Analyzing the scores further, we note that the correlation (R^2) achieved by sampling is generally lower for generated test suites. As the sampling denominator increases, the correlation reduces drastically for generated test suites while correlation remains strong for original test suites. Secondly, we also observe that the multiplier factor β_1 is smaller for generated test suites than original test suites. This observation holds good for all sampling and operator selection strategies examined. Both these factors together suggest that generated test suites have a higher variability in their sampled mutation scores than original test suites. We note that since β_1 was generally smaller for generated test suites than original test suites, the full mutation score corresponding to a sampled mutation score for generated test suites would be lower than what would be expected from the original test suite. Further, the amount of mutant reduction possible such that the sampled mutation scores still has a high enough correlation with the total score is lower for generated test suites.

Considering different sampling strategies, based on their stability and mutation reduction achieved, we note that using simple sampling (*sample(x)*) produced the most stable results. For both generated and original test suites, this strategy produced valid results for all different scopes and sampling ratios. It also was able to achieve a highest mutation reduction of 72.63, with high stability. That is, only 1.37% of the original mutants were needed to produce a highly correlated ($R^2 = 0.97$) mutation score, with minimal deviation ($\sigma = 6.36$) for original test suites. While the correlation is weaker for generated test suites it is still quite strong, with $R^2 = 0.90$

Our *round(x)*, the strategy of ignoring low complexity elements, did not pay off, with line scope having a tendency to become unstable and low correlated at low sampling ratios. At very low sampling ratio (1/64), the strategy with

Original							Generated						
scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1	scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
line	1/2	0.99	2.00	0.05	106	0.99	line	1/2	0.95	2.00	0.08	107	0.87
method	1/2	0.99	1.99	0.02	107	0.99	method	1/2	0.97	2.00	0.08	107	0.92
class	1/2	0.99	1.99	0.02	107	0.99	class	1/2	0.97	1.99	0.06	107	0.91
project	1/2	0.99	2.29	0.44	99	0.99	project	1/2	0.96	2.20	0.16	107	0.86
line	1/4	1.00	4.00	0.14	106	0.99	line	1/4	0.97	3.98	0.31	107	0.86
method	1/4	0.99	3.99	0.07	107	0.99	method	1/4	0.96	4.01	0.33	107	0.89
class	1/4	0.99	3.99	0.04	107	0.98	class	1/4	0.96	3.98	0.12	107	0.87
project	1/4	0.99	4.50	0.21	99	0.99	project	1/4	0.97	4.38	0.25	107	0.91
line	1/8	0.99	8.02	0.44	107	0.97	line	1/8	0.92	8.15	1.07	105	0.84
method	1/8	0.99	8.01	0.30	107	0.99	method	1/8	0.94	7.87	0.65	107	0.82
class	1/8	0.99	7.98	0.16	107	0.98	class	1/8	0.94	7.99	0.48	107	0.85
project	1/8	0.99	9.11	0.81	99	0.98	project	1/8	0.95	8.75	0.52	107	0.85
line	1/16	0.98	16.34	1.81	106	0.97	line	1/16	0.87	16.14	2.59	106	0.74
method	1/16	0.99	16.00	0.73	107	0.99	method	1/16	0.91	16.36	2.24	107	0.85
class	1/16	0.99	15.98	0.37	107	0.99	class	1/16	0.85	15.97	1.39	106	0.94
project	1/16	0.99	18.00	1.11	99	0.98	project	1/16	0.91	17.44	1.48	106	0.84
line	1/32	0.98	33.16	9.61	107	0.96	line	1/32	0.86	33.04	15.20	107	0.75
method	1/32	0.98	32.10	4.21	107	0.98	method	1/32	0.88	31.61	5.75	105	0.80
class	1/32	0.97	31.87	1.59	107	0.97	class	1/32	0.88	32.34	5.72	105	0.75
project	1/32	0.98	36.11	2.26	99	1.00	project	1/32	0.89	35.38	3.41	107	0.82
line	1/64	0.96	67.25	24.62	107	0.92	line	1/64	0.67	68.54	26.21	103	0.66
method	1/64	0.97	62.67	10.19	107	0.95	method	1/64	0.58	67.33	22.35	103	0.64
class	1/64	0.95	64.92	7.60	107	0.99	class	1/64	0.70	66.56	17.99	104	0.63
project	1/64	0.97	72.63	6.36	99	0.95	project	1/64	0.90	69.95	8.05	106	0.76

Table 2: The sample(x)% random selection criteria results.

Original							Generated						
scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1	scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
line	1/2	0.99	1.61	0.08	107	0.99	line	1/2	0.98	1.57	0.14	107	0.89
method	1/2	0.99	1.89	0.05	107	0.99	method	1/2	0.97	1.84	0.15	107	0.89
class	1/2	0.99	1.97	0.02	107	0.98	class	1/2	0.96	1.97	0.07	107	0.92
project	1/2	0.98	2.24	0.07	99	0.99	project	1/2	0.96	2.19	0.16	107	0.88
line	1/4	0.98	3.77	0.33	107	0.99	line	1/4	0.90	3.92	0.95	106	0.93
method	1/4	0.99	3.88	0.19	107	0.99	method	1/4	0.89	4.01	0.88	107	0.90
class	1/4	0.99	3.94	0.05	107	0.98	class	1/4	0.96	3.92	0.22	107	0.92
project	1/4	0.99	4.50	0.25	99	0.98	project	1/4	0.98	4.36	0.25	107	0.92
line	1/8	0.98	11.32	2.88	107	0.97	line	1/8	0.87	12.90	6.71	103	0.82
method	1/8	0.99	8.24	0.47	107	0.98	method	1/8	0.89	8.44	0.94	105	0.84
class	1/8	0.99	7.90	0.15	107	0.98	class	1/8	0.94	7.83	0.50	107	0.85
project	1/8	0.98	8.98	0.44	99	0.99	project	1/8	0.93	8.74	0.49	107	0.91
line	1/16	0.93	71.82	48.12	105	0.90	line	1/16	0.78	75.67	58.83	95	0.79
method	1/16	0.99	17.86	2.62	107	0.96	method	1/16	0.89	19.29	6.65	101	0.99
class	1/16	0.99	16.11	0.77	107	0.99	class	1/16	0.96	16.15	1.98	105	0.91
project	1/16	0.97	17.97	0.94	99	0.97	project	1/16	0.90	17.36	1.83	107	0.82
line	1/32	0.83	879.29	811.90	67	0.85	line	1/32	0.23	852.04	808.62	44	0.36
method	1/32	0.97	44.31	14.50	107	0.95	method	1/32	0.79	44.91	14.21	99	0.85
class	1/32	0.98	33.07	2.93	107	0.97	class	1/32	0.91	35.46	7.54	103	0.82
project	1/32	0.98	36.05	2.26	99	0.99	project	1/32	0.90	35.31	3.48	104	0.85
line	1/64	0.52	2032.47	1106.57	20	0.66	line	1/64	0.04	4214.24	6367.58	13	0.29
method	1/64	0.93	163.19	144.71	99	0.91	method	1/64	0.54	158.37	121.07	84	0.60
class	1/64	0.97	72.54	12.59	107	0.94	class	1/64	0.82	76.29	23.25	100	0.72
project	1/64	0.96	71.83	6.72	99	0.94	project	1/64	0.80	70.36	9.55	103	0.80

Table 3: The round(x)% random selection criteria results.

method scope also becomes unstable and low correlated. On the whole *round(x)* did not perform as well as *sample(x)*

The strategy of ensuring coverage with *ceil(x)* also did not perform well. While the results were generally stable, the reduction ratios achieved were lower than *sample(x)*, with similar correlations.

For line count strategies, only the external sampling produced consistent results, which were again not as good as *round(x)* for similar reduction ratios. Secondly, we had hypothesized that coverage was a significant contributor to having a high correlation with the full mutation score. However we could find no evidence in support of this hypothesis with line, method, class, and project scope showing similar correlation and consistency at similar reduction ratios.

The one per element strategy did not have stable results in general except for at the scope of line (at which point it was same as the count-per-element strategy).

While the x% per operator strategy produced consistent results, its consistency was not as good as *sample(x)*, and was either on par with or worse than *sample(x)* in R^2.

Considering mutation operators, only three operators *NC*, *RC*,and *DC* had consistent results (and only for original test suites). The n-selection strategy had high correlation and consistency (though low reduction ratio) until 7-selection (removing the seven most numerous operators) for both generated and original test suites. Javalanche did not have a high mutation reduction ratio, though it had high R^2 for both kinds of test suites. The other operator selection meth-

Original

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
line	1/2	0.99	1.61	0.08	107	0.99
method	1/2	0.99	1.89	0.05	107	0.99
class	1/2	0.99	1.97	0.02	107	0.99
project	1/2	0.99	2.24	0.08	98	0.99
line	1/4	0.99	2.24	0.23	107	0.99
method	1/4	0.99	3.36	0.27	107	1.00
class	1/4	0.99	3.88	0.15	107	0.99
project	1/4	0.98	4.50	0.26	99	0.99
line	1/8	0.99	2.65	0.43	107	0.99
method	1/8	0.98	5.50	0.83	107	1.00
class	1/8	0.99	7.43	0.43	107	1.00
project	1/8	0.97	8.94	0.39	99	0.99
line	1/16	0.98	2.74	0.51	107	0.98
method	1/16	0.98	7.98	1.98	107	0.99
class	1/16	0.98	13.65	1.37	107	0.99
project	1/16	0.98	17.76	0.97	99	0.97
line	1/32	0.98	2.75	0.52	107	0.98
method	1/32	0.97	10.22	3.59	107	0.99
class	1/32	0.98	23.71	4.64	107	0.99
project	1/32	0.98	34.87	2.24	99	0.97
line	1/64	0.99	2.75	0.52	107	0.98
method	1/64	0.96	11.55	5.15	107	0.99
class	1/64	0.96	36.95	11.23	107	0.98
project	1/64	0.97	67.92	4.73	99	0.97

Generated

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
line	1/2	0.97	1.57	0.14	107	0.87
method	1/2	0.97	1.84	0.15	107	0.88
class	1/2	0.98	1.97	0.07	107	0.93
project	1/2	0.95	2.18	0.11	107	0.88
line	1/4	0.96	2.12	0.34	107	0.86
method	1/4	0.94	3.20	0.55	107	0.79
class	1/4	0.96	3.80	0.25	107	0.86
project	1/4	0.97	4.34	0.26	107	0.89
line	1/8	0.93	2.46	0.55	107	0.81
method	1/8	0.89	5.14	1.42	107	0.70
class	1/8	0.95	7.18	0.89	107	0.88
project	1/8	0.94	8.62	0.68	107	0.83
line	1/16	0.94	2.53	0.62	107	0.84
method	1/16	0.86	7.48	3.03	107	0.64
class	1/16	0.91	12.69	2.60	107	0.81
project	1/16	0.90	16.79	1.83	107	0.80
line	1/32	0.94	2.54	0.62	107	0.83
method	1/32	0.82	9.71	5.21	107	0.59
class	1/32	0.89	20.97	6.47	107	0.78
project	1/32	0.89	32.20	5.86	107	0.86
line	1/64	0.94	2.54	0.62	107	0.83
method	1/64	0.81	11.10	6.90	107	0.57
class	1/64	0.86	31.99	14.03	107	0.77
project	1/64	0.86	61.26	14.11	107	0.82

Table 4: The ceil(x)% random selection criteria results.

Original

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
line	1/1	0.99	2.75	0.52	107	0.98
method	1/1	0.98	2.75	0.52	107	0.98
class	1/1	0.99	2.75	0.52	107	0.98
project	1/1	0.99	2.76	0.53	99	0.98
line	1/2	0.99	5.34	0.99	107	0.98
method	1/2	0.99	5.34	0.99	107	0.98
class	1/2	0.99	5.34	0.99	107	0.98
project	1/2	0.99	5.51	1.08	99	0.99
line	1/4	0.98	10.67	2.12	107	0.97
method	1/4	0.99	10.67	2.12	107	0.97
class	1/4	0.99	10.67	2.12	107	0.99
project	1/4	0.99	10.99	2.10	99	0.98
line	1/8	0.98	21.72	4.27	107	0.98
method	1/8	0.98	21.72	4.27	107	1.00
class	1/8	0.98	21.72	4.27	107	0.99
project	1/8	0.98	22.13	4.40	99	1.00
line	1/16	0.98	46.54	11.53	107	0.94
method	1/16	0.97	46.54	11.53	107	0.96
class	1/16	0.97	46.54	11.53	107	0.95
project	1/16	0.98	44.30	9.19	99	0.99

Generated

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
line	1/1	0.95	2.55	0.63	107	0.87
method	1/1	0.95	2.54	0.62	107	0.86
class	1/1	0.95	2.54	0.62	107	0.90
project	1/1	0.97	2.55	0.63	107	0.90
line	1/2	0.94	4.89	1.22	107	0.85
method	1/2	0.95	4.89	1.22	107	0.84
class	1/2	0.94	4.89	1.22	107	0.84
project	1/2	0.96	5.05	1.26	107	0.84
line	1/4	0.92	9.89	2.65	107	0.86
method	1/4	0.92	9.88	2.64	107	0.84
class	1/4	0.85	9.89	2.65	107	0.81
project	1/4	0.97	10.14	2.48	107	0.87
line	1/8	0.87	20.15	5.25	106	0.72
method	1/8	0.90	20.15	5.25	106	0.78
class	1/8	0.87	20.15	5.25	106	0.83
project	1/8	0.93	20.21	5.27	107	0.86
line	1/16	0.91	44.95	12.84	102	0.86
method	1/16	0.93	44.95	12.84	102	0.83
class	1/16	0.92	44.95	12.84	102	0.82
project	1/16	0.88	40.81	10.97	105	0.76

Table 5: The line count selection criteria results.

Original

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
method	1/2	0.98	6.88	1.47	106	0.96
class	1/2	0.99	5.34	0.99	107	0.99
method	1/4	0.98	18.44	9.72	107	0.95
class	1/4	0.99	10.67	2.12	107	0.99
method	1/8	0.94	61.61	40.25	104	0.92
class	1/8	0.98	21.72	4.27	107	0.99
method	1/16	0.91	238.66	214.95	81	0.90
class	1/16	0.97	46.54	11.53	107	0.96

Generated

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
method	1/2	0.84	6.68	1.78	105	0.94
class	1/2	0.92	4.89	1.22	107	0.82
method	1/4	0.79	17.56	6.88	102	1.04
class	1/4	0.87	9.89	2.65	107	0.82
method	1/8	0.72	57.28	37.09	96	0.86
class	1/8	0.83	20.15	5.25	106	0.78
method	1/16	0.50	174.92	107.33	67	0.58
class	1/16	0.94	44.95	12.84	102	0.89

Table 6: The line count selection criteria results (within).

ods performed poorly on generated test suites, while they had high correlation and reasonable stability on original test suites. However, for comparable mutation reduction levels, they all performed worse than random sampling strategies.

Considering different sampling strategies, we see a few general patterns. The first is that sampling strategy can predict the final score with very minimal loss up to a fraction of $\frac{1}{64}$ of the total number of mutants. We did not cut off with the 99% rule as mentioned byZhang et al. [30] (originally proposed by Offutt et al. [21] for a different measurement), and instead observe that we obtained $R^2 = 0.97$ which provides a high confidence in the accuracy of prediction of the

Original

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
block	1/1	0.99	3.89	0.75	107	1.00
line	1/1	0.99	2.75	0.52	107	0.98
method	1/1	0.96	12.28	6.74	107	0.99
class	1/1	0.95	79.63	81.64	107	0.97
line	1/2	0.99	5.34	0.99	107	0.98
method	1/2	0.96	22.10	12.25	107	0.98
class	1/2	0.94	79.68	81.61	107	0.96
line	1/4	0.98	10.67	2.12	107	0.97
method	1/4	0.96	45.98	32.33	107	0.97
line	1/8	0.98	21.72	4.27	107	0.98
method	1/8	0.95	109.23	61.94	106	0.95
line	1/16	0.98	46.54	11.53	107	0.94
method	1/16	0.90	322.67	235.35	94	0.89

Generated

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
block	1/1	0.90	4.04	1.21	107	0.76
line	1/1	0.95	2.55	0.63	107	0.87
method	1/1	0.79	12.01	8.42	107	0.56
class	1/1	0.81	63.26	68.83	107	0.69
line	1/2	0.94	4.89	1.22	107	0.85
method	1/2	0.80	21.25	15.06	107	0.56
class	1/2	0.80	63.26	68.83	107	0.70
line	1/4	0.92	9.89	2.65	107	0.86
method	1/4	0.73	44.05	35.81	107	0.54
line	1/8	0.87	20.15	5.25	106	0.72
method	1/8	0.75	114.93	97.44	98	0.48
line	1/16	0.91	44.95	12.84	102	0.86
method	1/16	0.72	295.94	259.87	75	0.40

Table 7: The one per element selection criteria results.

Original

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
operator	1/2	0.99	2.26	0.14	99	0.98
operator	1/4	0.99	4.47	0.15	99	0.99
operator	1/8	0.98	9.05	0.56	99	1.00
operator	1/16	0.99	17.98	1.16	99	0.98
operator	1/32	0.98	36.02	2.70	99	0.98
operator	1/64	0.97	71.67	10.42	99	0.97

Generated

scope	÷	R^2	μ_{red}	σ_{red}	valid	β_1
operator	1/2	0.96	2.18	0.13	107	0.87
operator	1/4	0.95	4.39	0.31	107	0.90
operator	1/8	0.89	8.82	0.63	107	0.86
operator	1/16	0.93	17.65	2.55	107	0.84
operator	1/32	0.80	34.70	4.03	106	0.80
operator	1/64	0.83	66.77	12.95	106	0.79

Table 8: The x% per operator criteria results.

Original

operator	R^2	μ_{red}	σ_{red}	valid	β_1
TSDL	0.95	6.85	5.47	101	0.82
IN	0.54	3542.79	2658.47	15	0.58
RV	0.95	15.12	9.73	107	0.94
M	0.79	184.75	252.49	92	0.82
VMC	0.85	32.33	41.63	105	0.94
NC	0.97	11.63	3.88	107	0.84
CB	0.88	97.00	115.39	106	0.83
I	0.89	178.87	203.51	103	0.71
NMC	0.97	3.88	3.71	107	1.00
CC	0.93	18.58	10.46	105	1.05
IC	0.94	14.46	11.92	107	0.99
RI	0.89	178.87	203.51	103	0.71
EMV	0.90	27.24	18.54	104	0.86
ES	0.70	1301.75	1523.32	47	0.56
RS	0.65	363.30	597.14	47	0.66
RC	0.97	11.63	3.88	107	0.95
DC	0.98	11.63	3.88	107	0.95

Generated

operator	R^2	μ_{red}	σ_{red}	valid	β_1
TSDL	0.91	5.78	6.83	105	0.72
IN	0.78	2767.00	2062.97	8	0.21
RV	0.74	19.35	17.35	106	0.50
M	0.44	195.86	250.58	72	0.61
VMC	0.26	20.52	20.43	105	0.81
NC	0.76	14.92	11.83	99	0.75
CB	0.72	117.35	135.85	86	0.70
I	0.50	233.23	253.06	76	0.49
NMC	0.89	3.44	3.62	106	1.02
CC	0.83	16.54	8.68	103	0.63
IC	0.75	15.78	11.59	101	0.87
RI	0.49	233.23	253.06	76	0.50
EMV	0.72	30.46	64.32	101	0.45
ES	0.01	1386.19	1612.68	27	0.13
RS	0.07	347.48	482.93	27	0.62
RC	0.74	14.92	11.83	99	0.85
DC	0.73	14.92	11.83	99	0.85

Table 9: Operators

Original

operator	R^2	μ_{red}	σ_{red}	valid	β_1
Wong01	0.98	9.83	3.24	107	0.86
Offutt96	0.97	8.58	3.12	107	0.86
Javalanche	0.99	1.62	0.15	107	0.98
Namin08	0.97	9.06	3.30	107	0.85
14-sel	0.72	1313.41	1533.80	52	0.60
13-sel	0.69	514.99	1146.97	52	0.68
12-sel	0.89	119.26	124.96	104	0.73
11-sel	0.90	66.27	65.26	104	0.73
10-sel	0.90	34.36	28.04	106	0.80
9-sel	0.92	27.00	25.76	107	0.82
8-sel	0.95	10.78	5.84	107	0.95
7-sel	0.96	7.08	3.19	107	0.95
6-sel	0.97	4.79	2.01	107	1.00
5-sel	0.98	3.28	0.93	107	1.01
4-sel	0.98	2.54	0.59	107	1.01
3-sel	0.99	2.02	0.32	107	1.01
2-sel	0.99	1.69	0.21	107	1.00
1-sel	0.99	1.46	0.17	107	0.98
0-sel	0.99	1.00	0.01	106	0.99

Generated

operator	R^2	μ_{red}	σ_{red}	valid	β_1
Wong01	0.78	13.15	11.64	99	0.77
Offutt96	0.78	11.92	11.27	99	0.79
Javalanche	0.97	1.54	0.17	107	1.00
Namin08	0.78	12.47	11.24	99	0.78
14-sel	0.07	1201.44	1534.04	28	1.02
13-sel	0.07	474.45	1230.40	28	0.67
12-sel	0.46	174.10	183.77	78	0.50
11-sel	0.48	93.75	94.34	78	0.51
10-sel	0.65	60.24	87.81	87	0.70
9-sel	0.62	42.17	41.90	92	0.72
8-sel	0.50	11.92	10.17	107	0.92
7-sel	0.84	6.63	3.04	107	0.76
6-sel	0.90	4.45	2.01	107	0.78
5-sel	0.92	3.17	0.89	107	0.84
4-sel	0.92	2.50	0.67	107	0.78
3-sel	0.93	2.05	0.40	107	0.78
2-sel	0.95	1.76	0.31	107	0.82
1-sel	0.95	1.55	0.28	107	0.83
0-sel	0.97	1.00	0.03	105	0.92

Table 10: Operator Selection

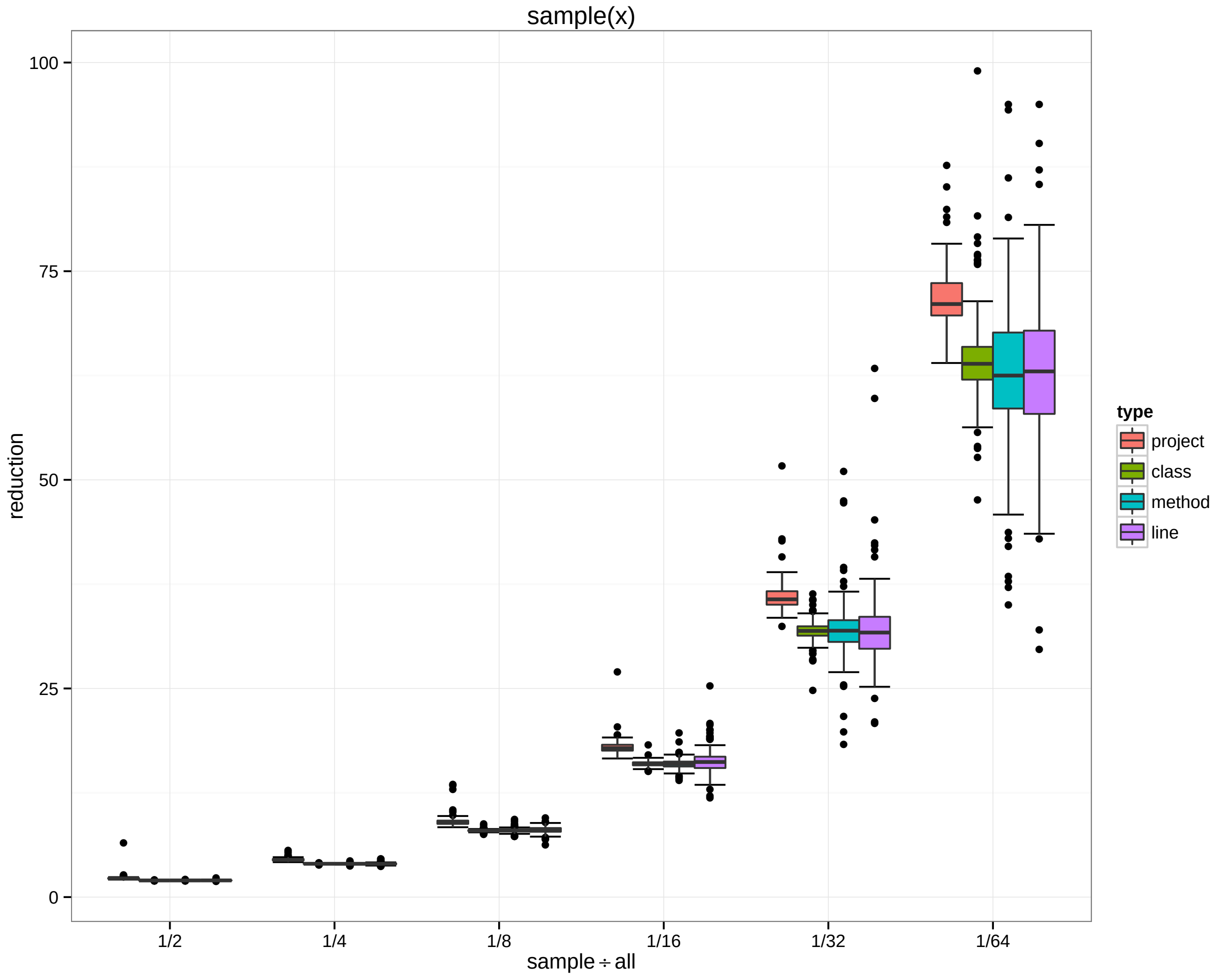

Figure 2: The effect of sampling reductions, Original set $R^2 > 0.90$. Notice the hierarchy of spread within boxplot for similar sampling ratios (÷) project < class < method < line i.e. project is better

full score.

The second pattern is that fractional sampling based on projects appears to be better than that based on classes, which in turn is better than methods and lines, with regard to the stability of the result (i.e. low standard deviation). This is not clear at lower levels of reduction (< 1/32), but becomes more pronounced for lower ratios (see Figure 2). There seems to exist a hierarchy of sampling, whereby as sampling becomes more fine grained stability decreases. We did not observe the pattern found by Zhang et al. [30] where method-based sampling had the best correlations. In fact, in our measurements, the opposite may be true, with project based sampling having the highest R^2 a majority of the time.

The third "pattern" is that the simplest sampling scheme — simple random sampling — performs best compared to sampling schemes that tried to provide higher weight to specific elements based on specific criteria. Additionally, operator selection seems to be generally worse off in predicting the final mutation score than any of the random selection schemes. In principle, if we could devise sampling methods that somehow captured only the most discriminative mutants for programs, they might perform better than any of these; our results suggest that most schemes that (we assume) aim at this goal do not in fact achieve the goal over a large body of programs, and so tend to perform worse than simply randomly sampling the whole pool of mutants, which has attractive simplicity in the absence of a better strategy.

7. THREATS TO VALIDITY

While we have taken every care to ensure that our results are valid, and to eliminate the effects of random noise, our results are subject to the following threats to validity.

Our results were observed on open source Java programs from the Github repository, using the maven build system. Further, only those projects that successfully completed mutation testing within a designated amount of time were chosen. This implies that if there is a confounding factor in play that affects the relation between the full mutation score and sampling mutation score, it might cause our results to be either wrong or less applicable.

Secondly, we had to rely on the PIT mutation testing tool, and had to extend its capabilities to some extent for our purposes. Software bugs are a fact of life. While every care has been taken to avoid them, there is still some possibility of some bugs having escaped us.

8. CONCLUSION

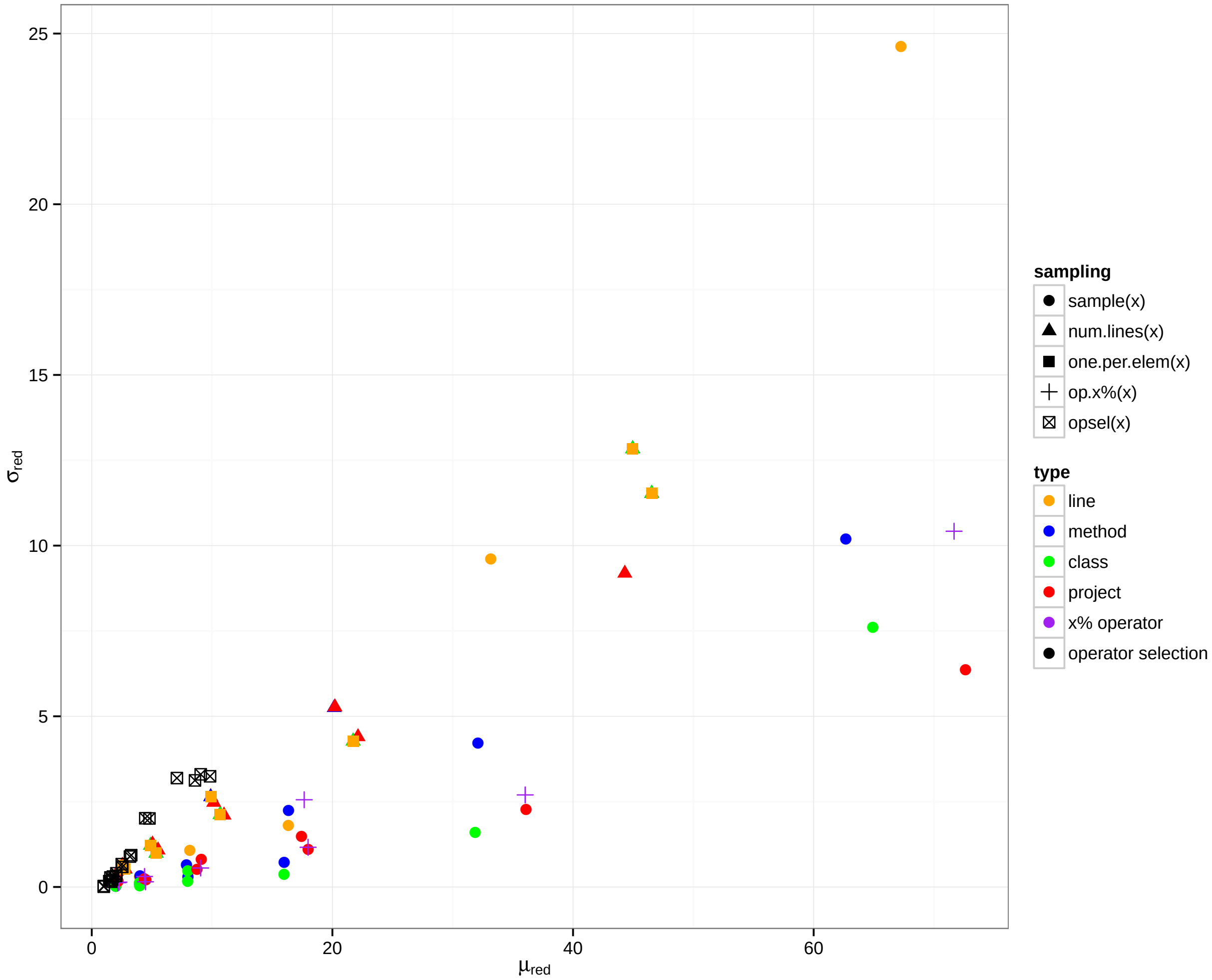

Figure 3: The mean reduction of mutants vs its standard deviation. Original set, main variations of sampling, only observations with $R^2 > 0.9$ and $2 \times \sigma_{red} < \mu_{red}$ are shown here. Notice that operator selection is has high σ for corresponding μ, and project based sampling is better for high reduction (note that x and y axis are in different scales)

Our analysis suggests that simple random sampling — that is, at the project level — works best when compared to more sophisticated schemes of sampling. Further, we also note that random sampling performs better in predicting final mutation score than operator selection.

This suggests that the best way to reduce the computational requirements of mutation analysis is fortunately easy to apply in almost any setting, given a tool that produces a good starting set of mutants. Namely, randomly sampling the total mutant population, even with very small sample sizes, can effectively predict the results of running all mutants, but at only a fraction of the computational cost. Not only is this method simple and effective, but it is also quite stable compared to some alternatives, with a low standard deviation in effectiveness across projects, for both original and random test suites.

There is a second possible point of interest here: as with coverage metrics used to predict mutation scores [13], using a large body of actual open source code to perform experiments seems to favor simple and easily-implemented approaches much more strongly than limited experiments on more academic subjects. We speculate that the inadequacies in real world test suites, or even in randomly generated tests for real world programs, may frequently be more simply predicted and evaluated than for typical research subjects, which may be smaller or involve less easily distinguished test suites.

9. REFERENCES

[1] A. T. Acree, Jr. *On Mutation*. PhD thesis, Atlanta, GA, USA, 1980. AAI8107280.

[2] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[4] Apache Software Foundation. Apache maven project. http://maven.apache.org.

[5] 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*, 11(2):113–136, 2001.

[6] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, New Haven, CT, USA, 1980. AAI8025191.

[7] H. Coles. Pit mutation testing. http://pittest.org/.

[8] H. Coles. Pit mutation testing: Mutators. http://pitest.org/quickstart/mutators.

[9] R. A. DeMillo, D. S. Guindi, W. McCracken, A. Offutt, and K. King. An extended overview of the mothra software testing environment. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 142–151. IEEE, 1988.

[10] L. Deng, J. Offutt, and N. Li. Empirical evaluation of the statement deletion mutation operator. In *IEEE 6th International Conference on Software Testing, Verification and Validation.*, Luxembourg, 2013.

[11] GitHub Inc. Software repository. http://www.github.com.

[12] R. Gopinath. Replication data for: A comparison of mutation approaches. http://dx.doi.org/10.7910/DVN/24936.

[13] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *36th International Conference on Software Engineering*, 2014.

[14] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[15] R. J. Lipton. Fault diagnosis of computer programs. Technical report, Carnegie Mellon Univ., 1971.

[16] A. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Computer Software and Applications Conference, 1991. COMPSAC '91., Proceedings of the Fifteenth Annual International*, pages 604–605, 1991.

[17] E. S. Mresa and L. Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing Verification and Reliability*, 9(4):205–232, 1999.

[18] A. S. Namin and J. H. Andrews. Finding sufficient mutation operators via variable reduction. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06)*, page 5, 2006.

[19] A. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Software Engineering, 1993. Proceedings., 15th International Conference on*, pages 100–107, 1993.

[20] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Trans. Softw. Eng. Methodol.*, 5(2):99–118, Apr. 1996.

[21] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[22] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*, pages 100–107. IEEE Computer Society Press, 1993.

[23] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[24] C. Pacheco and M. D. Ernst. Randoop random test generation. http://code.google.com/p/randoop.

[25] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 297–298, Aug. 2009.

[26] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*, pages 351–360. ACM, 2008.

[27] R. H. Untch. On reduced neighborhood mutation analysis using a single mutagenic operator. In *Proceedings of the 47th Annual Southeast Regional Conference*, ACM-SE 47, pages 71:1–71:4, New York, NY, USA, 2009. ACM.

[28] W. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185 – 196, 1995.

[29] W. E. Wong. *On mutation and data flow*. PhD thesis, Citeseer, 1993.

[30] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In *IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2013.

[31] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 435–444, New York, NY, USA, 2010. ACM.

[32] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 435–444. ACM, 2010.