Do Mutation Reduction Strategies Matter?

Rahul Gopinath[*], Amin Alipour[†], Iftekhar Ahmed[‡], Carlos Jensen[§], and Alex Groce[¶]
Department of EECS, Oregon State University
Email: [*]gopinath@eecs.orst.edu, [†]alipour@eecs.orst.edu, [‡]ahmed@eecs.orst.edu, [§]cjensen@eecs.orst.edu, [¶]agroce@gmail.com

***Abstract*—Mutation analysis is a well-known, but computationally intensive, method for measuring test suite quality. While multiple strategies have been proposed to reduce the number of mutants, there is inconclusive evidence for their utility due to the limited number and size of programs used for validation, and a lack of comprehensive comparative studies. Traditional evaluation criteria for mutation reduction also rely on mutation-adequate suites, which are rare in practice.**

We propose novel criteria for evaluating reduction strategies for non-mutation-adequate test suites, directly linked to the actual use of mutation analysis during development — to ensure that tests check for many different possible faults. We evaluate using both these criteria and the traditional criteria with 201 real-world projects, and show that the popular strategies — operator selection, and stratified sampling (on operators or program elements) — are at best marginally better than random sampling, and are often worse.

I. Introduction

Mutation analysis is the best approach for evaluating the quality of test suites. It involves producing a family of *mutants* (programs with small differences from the original program) used to evaluate the effectiveness of test suites by attempting to detect these mutants [1], [2]. Studies by Andrews et al. [3], [4] and more recently by Just et al. [5] suggest that mutations resemble and can simulate the behavior of real faults. However, mutation analysis of test suites has not been widely adopted as an actual software engineering practice [6], despite the need for tools able to evaluate tests [7]. A major impediment to wider adoption is its high computational cost; the set of mutants for even a moderate sized program can be very large, and their evaluation prohibitively time consuming.

A major approach to cost-reduction of mutation analysis is choosing a smaller, representative set of mutants [8], [9] — often called the *do fewer* approach. This approach can be divided into selective strategies and sampling strategies. The selective mutation strategy is to select a smaller representative subset of mutation *operators* based on heuristics and statistical analysis, and use these in the mutation analysis instead of the entire set of mutation operators [10], [11]. Sampling strategies seek to *randomly* select a representative subset of mutants to use in the same way. Acree [12] and Budd [13] proposed using only a small fraction of randomly selected mutants to evaluate a test suite. Wong and Mathur found that randomly sampling as few as 10% of mutants could provide representative results [14]. Recent studies have found that the sample size required to achieve adequacy in mutation sampling increases at a rate of $O(n^{\frac{1}{4}})$ as program size (n) increases [15].

While some of the recent studies [16], [17] have examined the relative merits of random sampling and operator selection strategies, Zhang et al. [17] point out that the field suffers from a lack of large scale studies, both in terms of the size of programs studied, and in the number and diversity of programs studied. This reduces our confidence in existing results. Most studies in this area are based on experiments with at most a dozen programs, which casts doubts on the generalizability of the results. We also note that several seminal studies on mutation reduction techniques [14], [18]–[20] use older programming languages such as Fortran, with operators specific to these languages. Further, the findings based on source-based mutants may not be directly applicable to mutants based on bytecode, such as those produced by Javalanche [21] and PIT [22], which are becoming common in both research [17], [23], [24] and industry.

Given these limitations, this paper studies different mutant reduction strategies with an emphasis on helping practicing software testers. For this reason, we evaluated different reduction strategies on a large number of Java programs using the popular bytecode-based mutation tool PIT [22]. We sampled 201 open source Java projects with between 50-116K lines of code (excluding comments and test cases) from Github and the Apache Foundation. This set of projects includes many popular Java projects regularly used in research and industry (such as Apache Common Lang, Common Math and Joda Time). For our mutation analysis, we chose to use PIT, a fast and easy to use Java mutation tool, used by researchers in the past [23], [24] due to its wide range of operators. In total, this study examines 799,028 mutants and 67,881 JUnit test cases with 200,110 asserts. We investigated five classic mutation selection techniques: Constrained [14], E-Selective [19], Javalanche [21], Variable Reduction [25], and N-Selection [18], and two sampling approaches: element-based [17] and operator-based sampling [14]. We compare the effectiveness of each against pure random sampling.

While the test suites of many open source programs are far from adequate,[1] they should satisfy a different requirement: namely, each test was almost always added through considerable manual labor [26], and was at least believed to be useful (the number of test cases correlates with the quality of software [26]). Therefore, *any test omitted* creates a potential for missed faults. An effective mutation reduction strategy should therefore identify the smallest possible set of *non-redundant* mutants to exercise the largest possible *non-redundant* test suite[2], and perform better than random selection. The cardinality of minimal test suite (which is same as the cardinality of the corresponding minimal mutant set) was suggested by Ammann et al. [27] as a measure of quality of a test suite.

[1] Mutation adequate test suites are suites with maximal mutation coverage; usually much less than 100% after discounting equivalent mutants.

[2] We only approximate a minimal suite, with greedy methods [27].

All test cases are not created equal, so we use assertion counts as well as the raw number of test cases, as the assertion count is a proxy for the number of features a test case verifies, shown to be correlated with fault detection by Fraser et al. [28]. Our basic claim is that an easy way to evaluate a mutant reduction strategy is to see if it promotes test suites with a large number of non-redundant test cases, or at least test suites with a large number of asserts.

The traditional mutation reduction strategy evaluation [20] involves finding the minimal adequate test suite for the reduced set of mutants suggested by the reduction strategy, and computing the ratio of mutants killed by this test suite and the original set of mutants killed[3]. However, since this criteria is not applicable for real-world programs which almost always are non-mutation adequate [29], we modify the criteria to suit non-mutation-adequate test suites. To ensure that our modification does not introduce any errors, we evaluate our criteria on four large projects with adequate test suites.

There could be other ways to evaluate mutation reduction strategies; for example one could imagine a criteria that encourages hardest to detect — yet not-equivalent mutants, or another based on the cost of evaluation of mutants. However, such criteria would not be useful for the basic purpose of mutation analysis — as an adequacy measure of the test suites targeting all kinds of bugs, not just hard to find bugs, or the easiest tests to evaluate.

Our results indicate that none of the reduction strategies evaluated provide any practical advantage over pure random sampling.[4]

Contributions:

- Our study is the largest so far in terms of both the size of programs involved (50-116K lines, excluding comments and tests), and the number of programs analyzed, (201 open source projects, totaling $1,241$K lines of code) for evaluating mutant reduction strategies under representative conditions.
- We examine a larger number of mutant reduction strategies than previous studies, including all the common and influential strategies for operator selection, and strata-based sampling.
- We use multiple evaluation criteria, applicable to both real-world non-adequate test suites, and traditional mutation adequate test suites.
- We find that current mutation reduction strategies seldom perform better than random sampling of mutants.

Organization. Section II describes previous research in mutation reduction strategies. Section III discusses the sampling and operator selection strategies we study in detail. The analysis we used is explained in Section IV, and results of experiments are detailed in Section V. A detailed discussion is provided in Section VI. Threats to validity are in Section VII. Section VIII summarizes our findings and conclusion.

II. Related Work

According to Mathur [31], the idea of mutation analysis was first proposed by Richard Lipton, and formalized by DeMillo et al. [32] A practical implementation of mutation analysis was done by Budd et al. [33] in 1980.

Mutation analysis subsumes different coverage measures [13], [34], [35]; the faults produced are similar to real faults in terms of the errors produced [36] and ease of detection [3], [4]. Just et al. [5] investigated the relation between mutation score and test case effectiveness using 357 real bugs, and found that the mutation score increased with effectiveness for 75% of cases, which was better than the 46% reported for structural coverage.

Performing a mutation analysis is usually costly due to the large number of test runs required for a full analysis [9]. This is one of the chief barriers to more widespread adoption of the technique. There are several approaches to reducing the cost of mutation analysis, categorized by Offutt and Untch [8] as: do *fewer*, do *smarter*, and do *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, the *do faster* approaches include mutant schema generation, code patching, and other methods.

The idea of using only a subset of mutants was conceived along with mutation analysis itself. Budd [13] and Acree [12] showed that even 10% sampling approximates the full mutation score with 99% accuracy. This idea was further explored by Mathur [37], Wong et al. [14], [38], and Offutt et al. [18] using Mothra [39] for Fortran.

A number of studies have looked at the relative merits of operator selection and random sampling criteria. Wong et al. [14] 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. [20] used the cost of detection as a means of operator selection. They found that if a 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 detecting mutants is considered.

Zhang et al. [16] compared operator-based mutant selection techniques to random sampling. They found that none of the selection techniques were superior to random sampling. They also found that uniform sampling is more effective for larger programs compared to strata sampling on operators[5], and the reverse is true for smaller programs. Recently, Zhang et al. [17] confirmed that sampling as few as 5% of mutants is sufficient for a very high correlation (99%) with the full mutation score, with even fewer mutants having a good potential for retaining high accuracy. 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 a set of *sufficient mutation operators* that reduce the cost of mutation but maintain corre-

[3] Mresa et al. call this the *operator mutation score* [20].

[4] We reached a similar conclusion when we used another traditional measure of comparison — correlation between full and selected mutation scores. See [30] for more details.

[5] The authors choose a random operator, and then a mutant of that operator. This is in effect strata sampling on operators given equal operator priority.

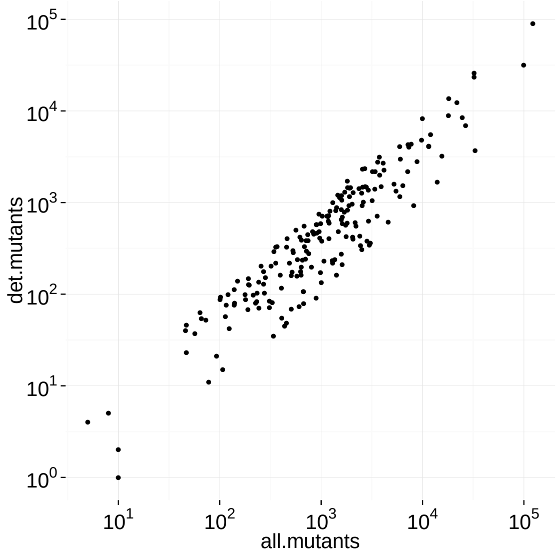


Fig. 1: The detected mutants plotted against the total number of mutants. This suggests that the mutation scores were not biased with respect to project size.

lation with the full mutation score. Offutt et al. [18] suggested an n-selective approach with step-by-step removal of operators that produce the most numerous mutations. Barbosa et al. [40] provided a set of guidelines for selecting such mutation operators. Namin et al. [25], [41] formulated the problem as a variable reduction problem, and found that just 28 out of 108 operators in Proteum were sufficient for accurate results.

Using only the statement deletion operator was first suggested by Untch [11], 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. [42] who defined deletion for different language elements, and found that an accuracy of 92% is achieved while reducing the number of mutants by 80%.

In operator and mutant subsumption, operators or mutants that do not significantly differ from others are eliminated. Kurtz et al. [43] found that a reduction up to 24 times can be achieved using subsumption alone, even though the result is based on an investigation of a single program, `cal`. Research into subsumption of mutants also includes Higher Order Mutants (HOM), whereby multiple mutations are introduced into the same set of mutants, reducing the number of individual mutants by subsuming component mutants. HOMs were investigated by Jia et al. [44], [45], who found that they can reduce the number of mutants by 50%.

Our work is an extension of previous work on comparison of mutation reduction strategies [16], [17], with a wider range of mutation approaches, and a larger set of large real-world projects. This makes our work more generalizable and usable by practicing testers.

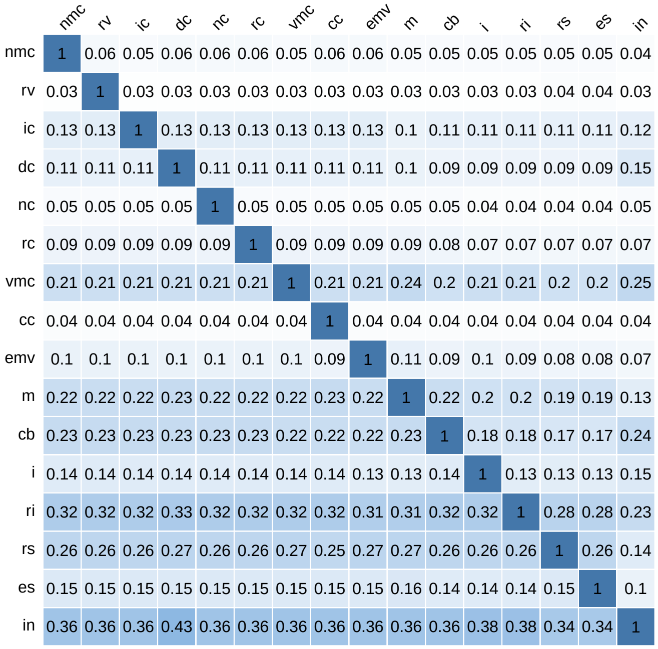
III. Methodology

Our selection of sample programs was driven by a few overriding concerns. Our primary requirement was that our results should be as representative as possible of real-world programs – our results should be applicable for both non-adequate test suites (which occur frequently during development), and stable well tested projects with adequate test suites. Secondly, we strove for a statistically significant result, reducing the number of variables present in the experiments.

For this reason, we conducted our study in two parts. In the first part we evaluated the effectiveness of different mutation reduction strategies on a large set of real-world projects containing non-adequate test suites. In the second part, we evaluated the same strategies using four large real-world mature projects with adequate test suites. This allowed us to have confidence in the wide applicability of our results.

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
DC	Replace boolean conditions with false

TABLE I: PIT Mutation Operators (We use abbreviations instead of operator names.)



	nmc	rv	ic	dc	nc	rc	vmc	cc	emv	m	cb	i	ri	rs	es	in
nmc	1	0.06	0.05	0.06	0.06	0.06	0.05	0.06	0.06	0.05	0.05	0.05	0.05	0.05	0.05	0.04
rv	0.03	1	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.04	0.04	0.03
ic	0.13	0.13	1	0.13	0.13	0.13	0.13	0.13	0.13	0.1	0.11	0.11	0.11	0.11	0.11	0.12
dc	0.11	0.11	0.11	1	0.11	0.11	0.11	0.11	0.11	0.1	0.09	0.09	0.09	0.09	0.09	0.15
nc	0.05	0.05	0.05	0.05	1	0.05	0.05	0.05	0.05	0.05	0.05	0.04	0.04	0.04	0.04	0.05
rc	0.09	0.09	0.09	0.09	0.09	1	0.09	0.09	0.09	0.09	0.08	0.07	0.07	0.07	0.07	0.07
vmc	0.21	0.21	0.21	0.21	0.21	0.21	1	0.21	0.21	0.24	0.2	0.21	0.21	0.2	0.2	0.25
cc	0.04	0.04	0.04	0.04	0.04	0.04	0.04	1	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04
emv	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.09	1	0.11	0.09	0.1	0.09	0.08	0.08	0.07
m	0.22	0.22	0.22	0.23	0.22	0.22	0.22	0.23	0.22	1	0.22	0.2	0.2	0.19	0.19	0.13
cb	0.23	0.23	0.23	0.23	0.23	0.23	0.22	0.22	0.22	0.23	1	0.18	0.18	0.17	0.17	0.24
i	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.14	0.13	0.13	0.14	1	0.13	0.13	0.13	0.15
ri	0.32	0.32	0.32	0.33	0.32	0.32	0.32	0.32	0.31	0.31	0.32	0.32	1	0.28	0.28	0.23
rs	0.26	0.26	0.26	0.27	0.26	0.26	0.27	0.25	0.27	0.27	0.26	0.26	0.26	1	0.26	0.14
es	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.16	0.14	0.14	0.14	0.15	1	0.1
in	0.36	0.36	0.36	0.43	0.36	0.36	0.36	0.36	0.36	0.36	0.36	0.38	0.38	0.34	0.34	1

Fig. 2: Subsumption rate between operators. Note that subsumption is not a symmertical relation. No operators come close to full subsumption. This suggests that **none of the operators studied are redundant**.

projects	mutants	muscore	tests	asserts
commons-lang3	32323	79.89	2828	16270
commons-math	122484	73.21	6743	17956
commons-configuration	18198	74.92	5216	11184
jodatime	32293	72.29	5120	22330

TABLE II: Projects with adequate test suites that are used for second part of analysis

For the first part, we chose a large random sample of Java projects from Github [46][6] and the Apache Software Foundation [47] that use the popular maven [48] build system. From an initial $1,800$ projects, we eliminated aggregate projects, and projects without test suites, which left us with 796 projects. Out of these, 326 projects compiled (common reasons for failure included unavailable dependencies, compilation errors due to syntax, and bad configurations). Next, projects that did not pass their own test suites were eliminated since the analysis requires a passing test suite. We also removed all projects with trivial test suites, leaving only those that had at least a 10% mutation score, leaving us with 201 projects. The distribution of detected mutants (y-axis) against that of the total mutants (x-axis) is given in Figure 1. This shows both the size of projects under consideration and the mutation scores obtained, and a reasonably non-biased distribution at least as far as mutation scores and project sizes are concerned. Our set of projects, as well as the complete data set for replication of this study is available for download[7].

We ran our mutation analysis on this set of projects using PIT [22], a tool used in several previous studies [23], [24], [49], [50]. However, since the operators provided by PIT are limited, we extended PIT to provide new operators similar to those provided by other mutation systems. The set of operators that we used is provided in Table I. In order to ensure that the operators that we added were not redundant, we computed their operator subsumption matrix, presented in the Figure 2. In the figure, the amount of subsumption is indicated by the shading, with same operators subsuming themselves entirely. This suggests that the operators we used were non-redundant, with a maximum subsumption of only 43%. Note that the subsumption relation is not symmetric, since it captures the non-overlapping portions of any pair of mutants. For a detailed description of each mutation operator, please refer to the PIT documentation [51].

We also modified PIT to report the entire test matrix of $tests \times failures$ rather than just the first test to fail. To remove the effects of random noise, results for each criteria were averaged over ten runs.

For the second part, we selected the four largest well tested projects from our sample, that had an adequate test suite. These are given in Table II. We follow the standard practice in mutation research [16], [17], [41] of assuming undetected mutants in well tested projects to be equivalent. We also note that these projects have been used previously in similar research [17]. The same set of operators and sampling criteria as in the first part were used on these projects.

A. *Sampling Criteria*

We used several sampling criteria suggested in the literature. For each sampling criteria we sampled mutants on a decreasing power scale, sampling fractions $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, and $\frac{1}{64}$ of the total mutants.

1) Uniform random sampling: The simplest sampling approach suggested by [13], where a fraction of the complete set of mutants is chosen at random, and serves as a baseline.

2) Stratified random sampling over mutation operators: First suggested by Wong et al. [14], this strategy samples the same proportion of mutants from each operator. While Wong seems to treat this as equivalent to x% selection, this sampling is subtly different from pure random sampling in that it provides a stratified sampling based on mutation operators.

3) Stratified random sampling over program elements: Following the suggestion of Zhang et al. [17], we extended x% selection 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 from Zhang et al. [17] for sampling fractional values.

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

B. *Operator Selection*

For selective methods, we evaluated the mutation operators suggested by Wong et al. [14], Offutt et al. [19], [42], and Namin et al. [41]. Since Javalanche [21] uses operator selection mechanisms, we included operators suggested by Javalanche separately. Note that all of these techniques except for Javalanche have targeted C programs. Thus, some operators may make sense in C but not in Java. For example, deletion of the `return` statement is tolerated by C compilers, but not in Java. Moreover, there were a few operators that could only be partially implemented in PIT (as below).

1) Constrained Mutation: Wong et al. [14] selected ROR and ABS from Mothra for selective mutation. The ABS operator was chosen because it forces users to consider all parts of the input domain, and ROR because it forces users to consider values of predicates. ROR mutates relational operators, while ABS replaces variables and expressions by their positive or negative absolute value, or zero. CB and NC from PIT are a good mapping for ROR. Similarly, IN is able to partially cover the ABS functionality.

2) E-Selective: Proposed by Offutt et al. [19]. Mothra supports three main classes of operators; Replacement (operand) mutators, Expression (operator) mutators, and Statement mutators. The operator selections used in this paper are groupings of these operators: ES, ER, RE, RS, E.

The best strategy identified by Offutt et al [19]. was the E-Selective strategy, which chooses only those mutators that modify operators. For Mothra, these were ABS, UOI, LCR, AOR, and ROR. UOI operates by incrementing or decrementing arithmetic expressions by 1, LCR changes the relational operators, and AOR mutates arithmetic operators.

[6] Github allows us access only a subset of projects using their search API. We believe this should not confound our results.

[7] We hope that new mutation reduction strategies will make use of our collection of programs, and the detection matrix of individual test cases against individual mutants. The test matrix sufficient to test a new approach on our set of programs (without running mutation analysis) is available for download at http://eecs.osuosl.org/rahul/ase2015/.

To accomplish the same with PIT, we divided the PIT operators similarly. Operand mutators are IC, EMV, and IN. Operator mutators are M, CB, NC, RC, and DC. Statement mutators are given by RV, I, VMC, NMC, CC, RI, ES, RS. We report the results of all combinations: ES, ER, RS, E, R, and S.

3) Javalanche: Javalanche [21], [52] adapts for Java bytecode, the E-Selective operator set suggested by Offutt et al. [19] for Fortran and implemented in C by Andrews et al. [3]. The original operators adapted by Andrews were 1) replacing an integer constant by its predecessor, successor, or by a small constant, 2) replacing arithmetic or boolean operators by an operator of the same class, 3) negating boolean conditions in control flow, and 4) statement deletion.

This translated [21] to 1) replace numerical constant operators. 2) replace arithmetic operator, and 3) negate jump condition. The last operator, 4) the omit method call, was added later [52].[8] These map directly to PIT operators IC, M, NC, and VMC.

4) Variable Reduction: Proposed by Namin et al. [41]. Namin framed the question as a statistical problem of finding the minimum set of operators that can best predict the final mutation score. That is, given that M is the final mutation score, and $m_1, m_2, \ldots, m_n$ are mutation scores given by n mutation operators, Namin wanted to find the smallest set of mutations that can predict M from the set of $m_{1..n}$. This boils down to finding the linear regression model.

Emulating *variable reduction* methodology for our experiment, we took advantage of the limited set of operators to run a complete subset model comparison to obtain the best model given by

$$\mu M_s = 0 + .55nmc + .2rc + .1dc + .2rv + .1cc + .7emv + .02m + .02ri$$

with $R^2 = 0.96$. This suggests that the variables we are interested in are NMC, RC, DC, RV, CC, EMV, M, and RI.

5) N-selection: Offutt et al. [18] suggested removal of the n most numerous operators. In our experiment, the order of operators was NMC, RV, IC, DC, NC, RC, VMC, CC, EMV, M, CB, I, RI, RS, ES, and IN. We discarded one at each step and evaluated the effectiveness at each n.

6) Statement deletion emulation: Statement deletion based operator selection is based on the work by Deng et al. [42]. 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 the 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 the bytecode level.

Note that we are not attempting to evaluate statement deletion mutation directly. Rather, we have chosen a set of operators that could be involved in deletion of statements. This means that in order to translate the results from our experiment back to the original statement deletion operator, we rely on some assumptions. We rely on a coupling effect: if a test is able to kill a mutant in this set, then it should kill it even when it is in combination with other mutants of this set (resulting in the deletion of the statement in question). That is, since statement deletion is a higher order mutant, according to the coupling hypothesis, it should fail more often than its component mutants, and should result in a lesser number of tests selected than the component faults taken separately, and hence a lower test utility. (If all tests detected all deleted statements, only a single test would be present in the minimal test suite).

Finally, reported results of statement deletion are based on component mutants involved in the emulation of true statement deletion. While this has no impact on the utility measures and strategy effectiveness, the mutation share differs between true statement deletion, and emulated statement deletion, and only the emulated mutation share is reported[9].

IV. Analysis

For the purpose of comparing different mutation reduction strategies, we computed the average *test utility*, and *assert utility* of different strategies, defined below.

To evaluate a mutation reduction strategy, we use the strategy to select a subset of mutants. We then collect all test cases that killed any of the selected mutants. Next, we compute the minimal, non-redundant test suite that detects all of these mutants. This is repeated and averaged over multiple runs, and we report the average.

M and $M_{strategy}$ denote the original set of mutants and the reduced set of mutants, respectively. A test suite composed of test cases that are able to kill a mutant m where $m \in M_{strategy}$ is in $T_{strategy}$, and the total number of `asserts` in $T_{strategy}$ is given by $asserts(T_{strategy})$. We find the minimal test suite that is capable of killing all mutants in $M_{strategy}$ using a greedy algorithm iteratively choosing the test cases with the largest amount of kills of remaining mutants. We denote this the "minimal" set $min(T_{strategy})$. Similarly, for a baseline, we build a set of randoml mutants M_{random}, such that $|M_{random}| = |M_{strategy}|$. Given any set of tests T, the total number of mutants that can be killed by the test cases in T is given by $kill(T)$.

Test utility (U_t) approximates the extra tests a selection strategy requires, compared to a random sampling, to kill the same number of mutants. The result is reported as a percentage of the non-redundant tests above the random sample (the baseline). That is, the test utility is given by:

$$U_t = \frac{|min(T_{strategy})|}{|min(T_{random})|} - 1$$

[8] We have already given a translation of the original operators suggested by Offutt as they apply to PIT. Here, we are evaluating how the translation implemented by Javalanche works. Javalanche has since this publication, added more operators to the default set. However it is not clear if they belong to a selected set under some criteria or if Javalanche is simply attempting to increase its repertoire of mutations.

[9] If n mutants in a statement were needed to emulate the statement deletion, the mutation share is reported based on n rather than based on the single statement deletion mutation that was emulated

Positive values show that the strategy requires *more* tests than random selection (it is better than random selection), and a negative test utility indicates that the strategy needs fewer test cases (it is worse than random testing). Values close to zero denote that the strategy tested performed similar to random selection. Note that the comparison here is between the size of tests and does not imply any subset relationship between test suites.

Since the assertions in a test were found to have a significant (albeit medium – 0.35) correlation with fault detection and mutation kill rate [28], we also compute the number of assertions in the test cases required by a strategy. If a test case does not have any assertions, we assume its number of assertions to be one (to account for uncaught exceptions and other kinds of failure).

The *assert utility* (U_a) is computed as the difference between the number of assertions in the selected non-redundant test cases and the number of assertions in the random sample. As before, it is reported as a percentage of the asserts of the non-redundant tests of the random sample:

$$U_a = \frac{|asserts(min(T_{strategy}))|}{|asserts(min(T_{random}))|} - 1$$

The *baseline effectiveness* (E_r) is computed by getting the number of mutants selected by the strategy under test, and selecting the same number of mutants randomly. We then collect the test cases that kill any of these mutants, and apply the same test cases against the original (complete) set of mutants. The result is then divided by the original number of detected mutants:

$$R_e = \frac{|kill(T_{random})|}{|kill(M)|}$$

The traditional mutation reduction criteria *strategy effectiveness*[10](E_s) is computed by collecting all the test cases that detect any of the mutants selected by the strategy under test, and applying these to the complete set of mutants. The score obtained is divided by the original number of detected mutants, and the effectiveness above that of baseline is reported:

$$E_s = \frac{|kill(T_{strategy})|}{|kill(M)|} - R_e$$

All values are reported as percentage (multiplied by 100).

It has to be noted that having a good test utility does not preclude a reduction strategy from having a poor strategy effectiveness or vice versa. It is possible for a strategy to select mutants such that there are a number of independent tests killing each mutant; however, if the tests kill no other mutants than the strategy selected ones, the strategy will have very poor strategy effectiveness. A similar argument applies for the inverse — a strategy selects a small number of very strong tests, which are able to kill most other mutants. However, we would expect a strong test that kills a much larger number of mutants than its peers to be distinguished by a larger number of assert statements. By computing the assert utility, we guard against such a possibility. We require only a strong positive utility in any one of the criteria to judge a strategy to be useful. However, a negative or inconsequential results with all the three criteria is a strong statement on the non-utility of the strategy in question.

V. Results

This section presents the results of our experiments. Each experiment was repeated ten times and averages were used to avoid random noise in results. We analyze results in detail, and then summarize overall patterns.

A. All Projects

We evaluated our criteria on 201 projects.

1) Operator Selection: in Table III, looking at test and assert utility, we see that there are a few selective strategies that perform best. Notably, the statement modification strategy *S-Selective* does about 1.7% better than random selection for test utility, and *RE-Selective* 1.07% in assert utility. It also seems *S-Selective* has a slight advantage of 0.24% in strategy effectiveness.

For N-selection, given in Table IV, the best test utility was for removal of NMC, RV, IC, DC, and NC, which resulted in an advantage of 2.76%. Assert utility was best for just removing NMC, (1.45%). The best strategy effectiveness (0.36%) was for removal of operators until VMC.

2) Stratified sampling over operators: For stratified sampling over operators (Table V), the best test utility appears to be at $\frac{1}{4}$, which gives an advantage of 1.12% over baseline, and also has best assert utility (1.08%), while strategy effectiveness was best for $\frac{1}{2}$ (0.01%).

3) Stratified sampling over program elements: For stratified sampling over program elements (Table VI), there appears to be a small but consistent advantage for most sample fractions. The best test utility achieved was 3.57% for line-based $\frac{1}{8}$ sampling, and best assert utility was for method-based $\frac{1}{32}$ sampling (4.38%). The $\frac{1}{32}$ class-based sampling had the best strategy effectiveness of 1.13%.

Summary. Operator selection strategies produce mutant sets that has slight positive utility at best (the utility is often negative) over random selection. Strata-based sampling tends to produce a positive utility when the elements sampled produces large enough mutants to provide a representative sample for the given sampling fraction.

B. Selected Projects

We evaluated our criteria on four projects: commons-lang3, commons-math, commons-configuration, jodatime.

1) Operator Selection: in Table VII, looking at test and assert utility, we see that there are a few strategies that perform best. Notably, the statement modification strategy *S-Selective* does about 4.43% better than random selection for test utility, and has an advantage of 1.83% in assert utility. It also seems to have a slight advantage of 0.15% in strategy effectiveness.

For N-selection, given in Table VIII, the best test utility was for removal of NMC, resulting in a test utility advantage of 0.89%, and best assert utility of 0.94% . This also has a strategy

[10] also called operator mutation score by Mresa et al. [20]

The operator selection results for all projects.

Strategy	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
Constrained	-7.47	-6.04	12.93	-0.71	94.03
E-Selective	-6.12	-5.11	32.89	-1.65	96.86
S-Selective	**1.70**	0.92	53.38	**0.24**	99.61
R-Selective	-2.35	0.44	13.73	-0.20	95.60
ES-Selective	-0.05	-0.11	86.27	0.01	99.99
RS-Selective	-3.59	-2.39	46.62	-0.23	98.63
RE-Selective	1.58	**1.07**	67.11	0.11	99.76
Javalance	-3.39	-2.25	61.43	-0.02	99.30
VarReduction	0.77	0.56	73.41	0.01	99.96
SDL	0.88	0.01	65.02	0.02	98.93

TABLE III: The operator selection strategy

Removed	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
rm.nmc	1.84	**1.45**	70.38	0.03	99.94
rm.rv	0.25	-0.52	59.90	0.10	98.84
rm.ic	0.52	-0.27	52.10	0.06	98.82
rm.dc	-0.16	-1.09	43.77	0.18	98.70
rm.nc	**2.76**	0.70	32.30	0.22	98.55
rm.rc	1.64	-2.08	23.11	0.24	98.20
rm.vmc	1.44	-3.07	19.00	**0.36**	97.91
rm.cc	-4.94	-4.62	12.93	-0.30	96.17
rm.emv	-13.15	-12.52	6.99	-4.57	81.55
rm.m	-14.67	-15.85	4.49	-6.92	78.07
rm.cb	-16.68	-19.40	3.03	-7.23	73.18
rm.i	-20.50	-28.69	1.80	-14.36	68.27
rm.ri	-25.12	-20.84	0.58	-6.73	27.34
rm.rs	-6.25	-7.58	0.15	-0.84	21.43
rm.es	-10.08	-13.35	0.02	-0.81	4.12

TABLE IV: The N-selective strategy. Each row removes the named mutation operator from the preceding row

Fraction	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2	0.35	0.27	50.18	**0.01**	99.51
1/4	**1.12**	**1.08**	24.98	-0.03	98.35
1/8	0.43	0.79	12.40	-0.11	96.77
1/16	-1.70	-1.61	6.17	-0.85	94.07
1/32	-2.04	-1.28	3.07	-2.14	89.36
1/64	-4.13	-3.67	1.53	-3.24	80.61

TABLE V: The operator-based x% sample strategy

Fraction Elt	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2 line	1.42	1.11	50.06	0.14	99.52
1/4 line	3.37	2.97	25.08	0.16	98.49
1/8 line	**3.57**	3.71	12.67	0.12	96.90
1/16 line	-0.19	-0.14	6.23	-0.83	94.01
1/32 line	-2.02	-1.70	3.14	-2.05	89.01
1/64 line	-2.91	-1.25	1.55	-2.90	79.67
1/2 method	1.16	0.98	49.91	0.04	99.46
1/4 method	2.91	2.52	25.09	0.29	98.38
1/8 method	2.29	1.96	12.42	0.21	96.81
1/16 method	2.68	2.79	6.17	0.04	94.21
1/32 method	3.01	**4.38**	3.13	0.68	88.54
1/64 method	0.52	0.61	1.60	-1.62	80.18
1/2 class	0.96	0.63	49.96	0.02	99.44
1/4 class	1.34	1.04	24.97	0.02	98.49
1/8 class	2.84	3.02	12.53	0.57	96.75
1/16 class	1.94	2.68	6.22	0.66	94.06
1/32 class	3.50	3.61	3.15	**1.13**	89.13
1/64 class	0.55	1.87	1.53	-0.55	80.28

TABLE VI: The element-based x% sample strategy

Format: The test and assert utility shows how good the mutation strategy is in selecting non-redundant test cases as percentage difference. The mutation share is the fraction of mutants selected by the strategy compared to the full set. The strategy effectiveness shows the total mutants caught by a test suite selected by the strategy mutants, and is provided as comparison to baseline effectiveness in percentage.

The operator selection results for selected projects.

Strategy	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
Constrained	-15.69	-11.62	13.92	-0.24	98.89
E-Selective	-12.76	-10.59	37.10	-0.98	99.86
S-Selective	**4.43**	**1.83**	51.16	**0.15**	99.85
R-Selective	-12.64	-7.76	11.74	-0.88	98.52
ES-Selective	0.01	-0.20	88.26	0.00	100.00
RS-Selective	-10.48	-8.07	48.84	-0.06	99.92
RE-Selective	3.43	1.77	62.90	0.05	99.94
Javalance	-4.98	-3.49	58.68	0.03	99.94
VarReduction	1.64	1.12	72.08	0.03	99.97
SDL	2.84	0.67	61.55	0.04	99.95

TABLE VII: The operator selection strategy

Removed	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
rm.nmc	**0.89**	**0.94**	74.04	0.01	99.99
rm.rv	-4.31	-4.06	63.67	-0.02	99.97
rm.ic	-4.45	-4.36	53.11	-0.01	99.95
rm.dc	-5.18	-5.44	48.55	0.01	99.93
rm.nc	0.34	-1.72	36.94	0.06	99.85
rm.rc	-3.90	-10.56	21.75	-0.78	99.33
rm.vmc	-8.95	-13.66	18.64	-0.60	99.09
rm.cc	-24.96	-21.26	14.62	-1.51	98.66
rm.emv	-28.60	-24.92	11.73	-4.07	97.56
rm.m	-29.96	-25.25	6.04	-5.31	96.12
rm.cb	-37.74	-36.51	3.87	-7.90	93.95
rm.i	-42.07	-37.40	2.32	-14.75	90.21
rm.ri	-44.36	-52.22	0.77	-12.74	74.21
rm.rs	-16.61	-15.33	0.27	**8.59**	52.88
rm.es	-27.85	-16.68	0.19	-6.72	38.74

TABLE VIII: The N-selective strategy. Each row removes the named mutation operator from the preceding row

Fraction	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2	-0.03	-0.06	50.00	0.00	99.93
1/4	0.10	0.16	25.00	0.00	99.61
1/8	**0.34**	0.40	12.50	0.06	98.73
1/16	-0.15	-0.18	6.25	**0.22**	96.59
1/32	-0.38	-0.51	3.13	-0.27	93.42
1/64	0.29	**2.84**	1.56	-0.20	87.77

TABLE IX: The operator-based x% sample strategy

Fraction Elt	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2 line	2.95	2.00	50.01	0.04	99.93
1/4 line	4.41	3.18	25.03	0.13	99.63
1/8 line	3.82	3.31	12.45	0.35	98.77
1/16 line	3.57	3.62	6.22	0.34	96.77
1/32 line	1.53	1.61	3.12	0.57	93.22
1/64 line	1.83	2.41	1.56	0.66	87.83
1/2 method	2.18	1.41	49.98	0.04	99.93
1/4 method	3.95	3.22	25.00	0.13	99.62
1/8 method	5.71	4.90	12.53	0.39	98.75
1/16 method	6.57	6.26	6.25	1.06	96.59
1/32 method	**7.02**	**6.44**	3.12	**1.36**	93.25
1/64 method	4.52	5.03	1.57	1.11	88.21
1/2 class	0.75	0.36	50.01	0.00	99.94
1/4 class	0.96	0.76	25.01	0.01	99.63
1/8 class	1.16	1.20	12.50	0.08	98.75
1/16 class	2.50	2.80	6.25	0.11	96.80
1/32 class	2.41	3.71	3.13	0.07	93.52
1/64 class	4.59	6.30	1.56	1.02	87.65

TABLE X: The element-based x% sample strategy

Format: The test and assert utility shows how good the mutation strategy is in selecting non-redundant test cases as percentage difference. The mutation share is the fraction of mutants selected by the strategy compared to the full set. The strategy effectiveness shows the total mutants caught by a test suite selected by the strategy mutants, and is provided as comparison to baseline effectiveness in percentage. (The format is same as previous page).

effectiveness (0.01%). However, the best strategy effectiveness was given by removal of NMC, RV, IC, DC, NC, RC, VMC, CC, EMV, M, CB, I, RI, and RS (8.59%).

2) Stratified sampling over operators: For stratified sampling over operators (Table IX), the best test utility appears to be at $\frac{1}{8}$, with 0.34% over baseline, and best assert utility was 2.84% for $\frac{1}{64}$ sampling. The best strategy effectiveness obtained was 0.22% for $\frac{1}{16}$ sampling.

3) Stratified sampling over program elements: For stratified sampling over program elements (Table X), there appears to be a small consistent advantage for most sample fractions. The best test utility achieved was 7.02% for method-based $\frac{1}{32}$ sampling, and an assert utility of 6.44% for $\frac{1}{64}$ and strategy effectiveness of 1.36% for $\frac{1}{32}$ sampling.

Summary. As in Subsection V-A, operator selection strategies produce mutant sets that have slight positive utility at best over random selection. Similarly, strata-based sampling tends to produce a positive utility when the elements sampled produce enough mutants to provide a representative sample for the given sampling fraction (which is smaller than with projects in Subsection V-A, on account of the large average project size).

VI. Discussion

An important concern for a software tester during development is whether a newly added test contributes towards the effectiveness of a test suite. Not all tests are useful — some are redundant — recall that we use *minimal test suites* for our test utility and assert utility measures, averaged over multiple runs. Even if tests are not equal, a new test will improve a test suite if it increases the *average size* of a *minimal test suite*. Hence, the *average size* of a minimal test suite is a reasonable measure for the utility of a set of mutants.

The second question is whether the test suite selected by a subset of mutants is similarly effective to the test suite selected by the full set of mutants. This is the question answered by the traditional criteria of strategy effectiveness.

It is possible for our criteria to return contrary results to the traditional criteria. For a pathological example, consider a set of test cases with a single strong test case, and a large number of weak test cases. This can result in a high strategy effectiveness if the strong test is included, and a low test utility due to the very low number of non-redundant test cases. Similarly, if the strong test case is excluded, it can result in a high test utility, while having a low strategy effectiveness if the mutants discarded by the strategy are same ones that are killed by the strong test. However, we consider a mutation strategy useful if it has some utility for *at least one* of these criteria.

It is very rare for a project under development to have a mutation-adequate test suite. Hence it is important to ensure that the criteria we use are tested on real-world programs with non-adequate test suites, across the whole range of mutation scores. For this reason, the first part of our analysis is done on a large number of real-world projects with a range of mutation scores. The second part of our analysis concentrates on large real-world projects with adequate test suites in order to ensure that our results remain valid even for such projects: developers want a strategy that works throughout a project's life cycle.

The interesting thing to note here is that there is *no* consistent winning strategy. That is, there is no strategy that provides an advantage over all others for both kinds of project. Second, operator selection strategies provide little benefit (or even decrease performance) over random selection for even strategy effectiveness (the traditional criteria), which is surprising given previous research in the field.

A similar conclusion can be drawn for N-Selection, an operator selection strategy we considered separately. While it provided a small advantage in strategy effectiveness for the removal of several operators, the advantage or disadvantage is rather inconsistent, and when the mutation share of the remaining mutants drops to less than 10%, it is almost always disadvantageous compared to random selection.

The results indicate that operator selection strategies in general tend to be either disadvantageous (sometimes by a large difference), or where advantageous, this is by a very small margin compared to the baseline. There is a possible reason for this result: the three measures that we examined encourage mutation reduction strategies that result in mutants with the largest amount of variation in terms of tests detecting them, and any selection process that results in a reduction in variation tends to get penalized. It is possible that the operators chosen during operator selection produced mutants that were more similar to each other than the full set of mutants, and hence resulted in reduced variation. We see no reason why reduced variation is a benefit.

The strata-based random selection strategies fare a little better. While they are mostly advantageous, the advantage gained is always rather small — below 10%. Strata-based selection is founded upon a simple assumption; mutants within strata are more similar to each other than to those outside, and strata-based selection works well for approximating full mutation scores [17], [30] when this assumption is met. Our results indicate that while there exists a small advantage, this advantage is almost always less than 1% compared to the baseline for strategy effectiveness. One factor to consider in strata-based sampling is that it is effective only as long as all the elements of a given strata can provide samples for a given fraction. A $\frac{1}{64}$ fraction sample for a statement generating 10 mutants is effectively zero, and hence statement-based strata may no longer be useful for $\frac{1}{64}$ samples. This effect can be seen in the negative utility and strategy effectiveness for smaller fractions for various strata.

A general pattern seems to be that none of the mutation reduction strategies provide a practical large benefit over the baseline of random sampling — the simplest strategy of all — and they likely do not provide enough benefit to justify the additional complexity involved in implementing the various reduction strategies. Importantly, this finding seems to be true irrespective of the criteria used to judge; both test utility and strategy effectiveness provide similar results. This suggests that, at least from the perspective of a software tester wishing to use mutation analysis as an evaluation criteria during development (either to judge which tests to add, or to determine if a stopping criteria has been met), *random sampling is the easiest, and most effective, way to reduce the computational requirement of mutation analysis.*

This result merits another question. Should we attempt to find better reduction methods? What is the best we can do? Our preliminary [53] analysis shows that a maximum strategy effectiveness of up to 58.2% is possible (in theory) against random sampling on average using the full mutation results. However, this may not be practically achievable as it relies on perfect stratification and the knowledge of actual kills[11].

While it is tempting to draw further conclusions from the data we collected, the lack of a clear winner across the board and the rather small differences from the baseline in the best case make it difficult to isolate real patterns from random noise.

VII. Threats to Validity

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

Threats due to sampling bias: To ensure representativeness of our samples, we opted to use search results from the Github repository of Java projects that use the `Maven` build system. We picked *all* projects that we could retrieve given the Github API, and selected from these only based on constraints of building and testing. However, our sample of programs could be biased by skew in the projects returned by Github.

Projects of small size and low coverage: Since we used real-world projects with real test suites, the size, coverage, and hence the mutation scores are representative of real world projects. Unfortunately, given that a large majority of these projects are in the process of development, some of them had low coverage and mutation score, and very few had adequate test suites. However, our analysis remains the same even with well-tested high-coverage projects.

Bias due to tool used: As our focus was on the practical advantages of different mutation reduction strategies for a practicing tester, we relied on a popular mutation tool used in industry — PIT. However, PIT does have some drawbacks such as an incomplete repertoire of mutation operators and an imperfect mapping to source level mutants. While we have done our best to extend PIT to provide a reasonably sufficient set of mutation operators, and have tried to map the source level mutants to bytecode level mutants, some imperfections may still exist. However, given that we have captured the original reasoning behind the strategies, and also that previous research on same area has used Javalanche, which operates under similar constraints, we believe that the influence on our results is minimal.

VIII. Conclusion

Our analysis suggests that while some of the various more complex mutant reduction strategies produce slightly advantageous mutant sets, the advantage gained is usually very small. None of the common operator selection strategies performed notably and consistently well against random sampling, irrespective of the criteria used for evaluation. While some individual operators had a mean positive test utility or a mean positive strategy effectiveness, the effect itself was too small (on the order of one or two mutants or tests) to be of much help for a tester in the real world. A similar result is seen for operator-based x% selection and for element-based x% strategies, which only show a marginal advantage in test utility. This may not be worthwhile, considering the additional complexity of implementation required. This suggests that the best approach for a working tester is to rely on the simplest scheme of all — random selection of mutants.

It might be asked, why do we endorse pure random sampling when there are advantages (however small) for some more complex methods? First, there is the fact that these *are* “more complex” methods: given that random sampling is extremely simple to implement and very likely to be bug-free, it is therefore reasonable to prefer it to other methods, in the absence of compelling advantages to justify the added complexity. Our data suggests the gains from complexity are very small indeed.

Second, we follow Hamlet’s principle, formulated in discussing random testing [54]: *in the absence of a rational basis for systematic methods, random methods are best at avoiding bias*. That is, we are not *aware* of any way that the various mutant reduction strategies are biased against finding certain kinds of bugs. But we also do not have the information to detect biases if these exist. Random selection, at least, should only be as biased as the full set of mutations produced, and is thus safer to use in the absence of a deeper understanding of how mutants and real faults relate.

There is a second possible point of interest in our results: as with coverage metrics [23], using a large body of real-world open source projects to perform experiments seems to favor simple and easily-implemented approaches more strongly than more limited academic research samples using only a few benchmark projects. We speculate that the inadequacies of real-world test suites may frequently be more simply predicted and evaluated than those of typical research subjects: these are typically smaller or involve less easily distinguished test suites, or in other cases only consider well-tested programs. There is also an inherent danger in using only a small sample of the same academic benchmarks over and over again. We may end up seeing patterns that are specific to the subjects, and not present in the wider population. The non-performance of the popular strategies against a more widely-drawn data set such as ours may be seen as a possible symptom of such over-fitting.

Our advice to mutation analysis researchers is to ensure that any reduction strategies proposed are validated using a large number of real-world projects, and to ensure that any reduction criteria proposed maintains the original amount of variation. We make available our collection of programs and mutation testing results from them for this purpose.

Our advice to mutation tool implementors is two fold: try to provide as much sources of variation as possible, and avoid questionable operator selection strategies that reduce overall variation. You can always reduce the number of mutants produced by random sampling of the mutants produced.

We give similar advice to the practicing tester: pure random sampling is the best method for mutation reduction for a generic project. Avoid operator selection unless the operators thus rejected are sure to be subsumed by others, and use strata-based sampling only when you can be sure that all strata elements can produce representative samples for a given sampling fraction.

[11] Minimal mutants [27] may provide another post-hoc measure.

REFERENCES

[1] R. J. Lipton, "Fault diagnosis of computer programs," Carnegie Mellon Univ., Tech. Rep., 1971.

[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 *International Conference on Software Engineering*. IEEE, 2005, pp. 402–411.

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

[5] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Hong Kong, China: ACM, 2014, pp. 654–665.

[6] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *International Symposium on Software Reliability Engineering*, Nov 2014, pp. 201–211.

[7] P. Runeson, "A survey of unit testing practices," *Software, IEEE*, vol. 23, no. 4, pp. 22–29, July 2006.

[8] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," in *Mutation testing for the new century*. Springer, 2001, pp. 34–44.

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

[10] A. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *International Conference on Software Engineering*, 1993, pp. 100–107.

[11] R. H. Untch, "On reduced neighborhood mutation analysis using a single mutagenic operator," in *Annual Southeast Regional Conference*, ser. ACM-SE 47. New York, NY, USA: ACM, 2009, pp. 71:1–71:4.

[12] A. T. Acree, Jr., "On mutation," Ph.D. dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

[13] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, New Haven, CT, USA, 1980.

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

[15] J. Zhang, M. Zhu, D. Hao, and L. Zhang, "An empirical study on the scalability of selective mutation testing," in *International Symposium on Software Reliability Engineering*. ACM, 2014.

[16] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 435–444.

[17] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *IEEE/ACM Automated Software Engineering*. ACM, 2013.

[18] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *International Conference on Software Engineering*. IEEE Computer Society Press, 1993, pp. 100–107.

[19] 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*, vol. 5, no. 2, pp. 99–118, 1996.

[20] E. S. Mresa and L. Bottaci, "Efficiency of mutation operators and selective mutation strategies: An empirical study," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 205–232, 1999.

[21] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2009, pp. 69–80.

[22] H. Coles, "Pit mutation testing," http://pitest.org/.

[23] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *International Conference on Software Engineering*. IEEE, 2014.

[24] L. Inozemtseva and R. Holmes, "Coverage Is Not Strongly Correlated With Test Suite Effectiveness," in *International Conference on Software Engineering*, 2014.

[25] A. S. Namin and J. H. Andrews, "Finding sufficient mutation operators via variable reduction," in *Workshop on Mutation Analysis*, 2006, p. 5.

[26] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *Software Engineering, IEEE Transactions on*, vol. 31, no. 3, pp. 226–237, March 2005.

[27] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 21–30.

[28] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated white-box test generation really help software testers?" in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 291–301.

[29] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2013.

[30] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "An empirical comparison of mutant selection approaches," Oregon State University, Tech. Rep., 2015. [Online]. Available: http://hdl.handle.net/1957/55691

[31] A. P. Mathur, *Foundations of Software Testing*. Addison-Wesley, 2012.

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

[33] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Theoretical and empirical studies on using program mutation to test the functional correctness of programs," in *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1980, pp. 220–233.

[34] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Software Testing, Verification and Reliability*, vol. 4, no. 1, pp. 9–31, 1994.

[35] A. J. Offutt and J. M. Voas, "Subsumption of condition coverage techniques by mutation testing," Technical Report ISSE-TR-96-01, Information and Software Systems Engineering, George Mason University, Tech. Rep., 1996.

[36] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1996, pp. 158–171.

[37] A. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Annual International Computer Software and Applications Conference, COMPSAC*, 1991, pp. 604–605.

[38] W. E. Wong, "On mutation and data flow," Ph.D. dissertation, Purdue University, West Lafayette, IN, USA, 1993, uMI Order No. GAX94-20921.

[39] R. A. DeMillo, D. S. Guindi, W. McCracken, A. Offutt, and K. King, "An extended overview of the mothra software testing environment," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 1988, pp. 142–151.

[40] 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.

[41] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *International Conference on Software Engineering*. ACM, 2008, pp. 351–360.

[42] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *IEEE 6th ICST*, Luxembourg, 2013.

[43] R. Kurtz, P. Ammann, M. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *Workshop on Mutation Analysis*, 2014.

[44] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, Oct. 2009.

[45] ——, "Constructing subtle faults using higher order mutation testing," in *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2008, pp. 249–258.

[46] GitHub Inc., "Software repository," http://www.github.com.

[47] Apache Software Foundation, "Apache commons," http://commons.apache.org/.

[48] ——, "Apache maven project," http://maven.apache.org.

[49] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 246–256.

[50] M. Delahaye and L. Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Software: Practice and Experience*, 2015.

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

[52] D. Schuler and A. Zeller, "Javalanche: Efficient mutation testing for java," in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, Aug. 2009, pp. 297–298.

[53] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "Mutant census: An empirical examination of the competent programmer hypothesis," Corvallis, OR: School of Electrical Engineering and Computer Science, Oregon State University, Tech. Rep., 2015.

[54] D. Hamlet, "When only random testing will do," in *Proceedings of the 1st International Workshop on Random Testing*, ser. RT '06. New York, NY, USA: ACM, 2006, pp. 1–9.