On The Limits Of Mutation Reduction Strategies

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

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

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

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

Alex Groce
Oregon State University
agroce@gmail.com

ABSTRACT

Although mutation analysis is considered the best way to evaluate the effectiveness of a test suite, hefty computational cost often limits its use. To address this problem, various mutation reduction strategies have been proposed, all seeking to gain efficiency by reducing the number of mutants while maintaining the representativeness of an exhaustive mutation analysis. While research has focused on the efficiency of reduction, the effectiveness of these strategies in selecting representative mutants, and the limits in doing so has not been investigated.

We investigate the practical limits to the effectiveness of mutation reduction strategies, and provide a simple theoretical framework for thinking about the absolute limits. Our results show that the limit in effectiveness over random sampling for real-world open source programs is 13.078% (mean). Interestingly, there is no limit to the improvement that can be made by addition of new mutation operators.

Given that this is the maximum that can be achieved with perfect advance knowledge of mutation kills, what can be practically achieved may be much worse. We conclude that more effort should be focused on enhancing mutations than removing operators in the name of selective mutation for questionable benefit.

Categories and Subject Descriptors

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

General Terms

Measurement, Verification

Keywords

test frameworks, evaluation of coverage criteria, statistical analysis

1. INTRODUCTION

The quality of software is a pressing concern for the software industry, and is usually determined by comprehensive testing. However, tests are themselves programs, (usually) written by human beings, and their quality needs to be monitored to ensure that they in fact are useful in ensuring software quality (e.g., it is important to determine if the tests are also a quality software system).

Mutation analysis [10, 37] is currently the recommended method [4] for evaluating the efficacy of a test suite. It involves systematic transformation of a program through introduction of small syntactical changes, each of which is evaluated against the given test suite. A mutant that can be distinguished from the original program by the test suite is deemed to have been *killed* by the test suite, and the ratio of all such mutants to the set of mutants identified by a test suite is its mutation (kill) score, taken as an effectiveness measure of the test suite.

Mutation analysis has been validated many times in the past. Andrews et al. [3,4], and more recently Just et al. [33], found that faults generated through mutation analysis resemble real bugs, their ease of detection is similar to that of real faults, and most importantly for us, a test suite's effectiveness against mutants is similar to its effectiveness against real faults.

However, mutation analysis has failed to gain widespread adoption in software engineering practice due to its substantial computational requirements — the number of mutants generated needs to be many times the number of tokens in order to achieve exhaustive coverage of even first order mutants (involving one syntactic change at a time), and each mutant needs to be evaluated by a potentially full test suite run.

A number of strategies have been proposed to deal with the computational cost of mutation analysis. These have been classified [44] orthogonally into *do faster*, *do smarter*, and *do fewer* approaches, corresponding to whether they improve the speed of execution of a single mutant, parallelize the evaluation of mutants, or reduce the number of mutants evaluated.

A large number of *do fewer* strategies — mutation reduction methods that seek to intelligently choose a smaller, representative, set of mutants to evaluate — have been investigated in the past. They are broadly divided into operator selection strategies, which seek to identify the smallest subset of mutation operators that generate the most useful mutants [43,46], and strata sampling [1,8] techniques, which seek to identify groups of mutants that have high similarity between them to reduce the number of mutants while

maintaining representativeness and diversity [55, 56]. Even more complex methods using clustering [19], static analysis [29, 35] and other intelligent techniques [49] are under active research [20].

These efforts raise an important question: What is the actual effectiveness of a perfect mutation reduction strategy over the baseline – random sampling, given any arbitrary program?

We define the **efficiency** of a selection technique as the amount of reduction achieved, and the **effectiveness** as the selection technique's ability to choose a representative reduced set of mutants that capture all the behavior of the original set of mutants. The ratio of effectiveness of a technique to that of random sampling is taken as the **utility** of the technique.

We approach these questions from two directions. First, we consider a simple theoretical framework in which to evaluate the improvement in effectiveness for the best mutation reduction possible, using a few simplifying assumptions, and given oracular knowledge of mutation kills. This helps set the base-line. Second, we empirically evaluate the best mutation reduction possible for a large number of projects, given post hoc (that is, oracular) detection knowledge. This gives us practical (and optimistic) limits given common project characteristics.

Our contributions are as follows:

- We find a theoretical upper limit for the effectiveness of mutation reduction strategies of 58.2% for a uniform distribution of mutants — the distribution most favorable for random sampling. We later show that for real world programs, the impact of distribution is very small (4.467%) suggesting that uniform distribution is a reasonable approximation.
- We find an empirical upper limit for effectiveness through the evaluation of a large number of open source projects, which suggests a maximum practical utility of 13.078% on average, and for 95% of projects, a maximum utility between 12.218% and 14.26% (*one sample u-test* $p < 0.001$)[1].
- We show that even if we consider a set of mutants that are distinguished at least by one test (thus discounting the impact of skew in redundant mutants) we can expect a maximum utility of 17.545% on average, and for 95% of projects, a maximum utility between 16.912% and 18.876% (*one sample u-test* $p < 0.001$).

What do our results mean for the future of mutation reduction strategies? Any advantage we gain over random sampling is indeed an advantage, however small. However, our understanding of mutant semiotics[2] is as yet imperfect, and insufficient to infer whether the kind of selection employed is advantageous. In fact, our current research [23] shows that current operator selection strategies seldom provide any advantage over random sampling, and even strata sampling based on program elements never achieves more than a 10% advantage over pure random sampling. Our results suggest that the effort spent towards improving mutant selection mechanisms should be carefully weighed against the potential maximum utility, and the risks associated with actually making things worse through biased sampling.

Our research is also an endorsement of the need for further research into new mutators. It suggests that addition of new mutators and then randomly sampling the same number of mutants as that of the original set, is only subject to a similar maximum disadvantage ($\frac{0.189 \times 100}{1-0.189} = 23.268\%$ upper limit for 95% projects), while having essentially *no upper bound* on advantage due to increase in effectiveness.

The asymmetry between improvement obtained by operator removal and operator addition is caused by the difference in population from which the random comparison sample is drawn. For operator selection, the perfect set remaining after removal of operators is a subset of the original population. Since the random sample is drawn from the original population, it can potentially contain a mutant from each strata in the perfect set. For operator addition, the new perfect set is a superset of the original population, with as many new strata as new mutants (no bounds on the number of new strata). Since the random sample is constructed from the original population, it does not contain the newly added strata.

Our results suggest a higher payoff in **finding newer categories** of mutations, than in trying to reduce the mutation operators already available.

In the interests of easy replication, our research is organized and reproducible using *Knitr*. The raw *Knitr* source of our paper along with the *R* data set required to build the paper, and the instructions to do so, are available [22].

2. RELATED WORK

According to Mathur [39], the idea of mutation analysis was first proposed by Richard Lipton, and formalized by DeMillo et al. [17] A practical implementation of mutation analysis was done by Budd et al. [9] in 1980.

Mutation analysis subsumes different coverage measures [8, 40, 45]; the faults produced are similar to real faults in terms of the errors produced [14] and ease of detection [3, 4]. Just et al. [33] 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 [32]. 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 [44] 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

[1] We use the non-parametric Mann-Whitney *u-test* as it is more robust to normality assumption, and to outliers. We note that a *t-test* also gives similar results.

[2] Here semiotics is the relation between a syntactic change and its semantic impact.

along with mutation analysis itself. Budd [8] and Acree [1] showed that even 10% sampling approximates the full mutation score with 99% accuracy. This idea was further explored by Mathur [38], Wong et al. [51, 52], and Offutt et al. [43] using Mothra [16] for Fortran.

A number of studies have looked at the relative merits of operator selection and random sampling criteria. Wong et al. [51] 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. [41] 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. [56] 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[3], and the reverse is true for smaller programs. Recently, Zhang et al. [55] 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 correlation with the full mutation score. Offutt et al. [43] suggested an n-selective approach with step-by-step removal of operators that produce the most numerous mutations. Barbosa et al. [7] provided a set of guidelines for selecting such mutation operators. Namin et al. [42, 48] 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 [50], 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. [18] who defined deletion for different language elements, and found that an accuracy of 92% is achieved while reducing the number of mutants by 80%.

A similar mutation reduction strategy is to cluster similar mutations together [20,27], which has been attempted based on domain analysis [29] and machine learning techniques based on graphs [49].

In operator and mutant subsumption, operators or mutants that do not significantly differ from others are eliminated. Kurtz et al. [36] found that a reduction of 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. [30, 31], who found that they can reduce the number of mutants by 50%.

Ammann et al. [2] observes that the set of minimal mutants corresponding to a minimal test suite have the same cardinality as the test suite, and provides a simple algorithm for finding both a minimal test suite and a corresponding minimal mutant set. Further, they also suggest this minimal mutant set as a way to evaluate the quality of a mutation reduction strategy. Our work is an extension of Ammann et al. [2] in that we provide a theoretical and empirical bound to the amount of improvement that can be expected by *any* mutation reduction strategy.

3. THEORETICAL ANALYSIS

The ideal outcome for a mutation reduction strategy is to find the minimum set of mutants that can represent the complete set of mutants. A mutation reduction strategy accomplishes this by identifying redundant mutants and grouping them together so that a single mutant is sufficient to represent the entire group. The advantage of such a strategy over random sampling depends on two characteristics of the mutant population. First, it depends on the number of redundant mutants in each group of such mutants. Random sampling works best when these groups have equal numbers of mutants in them (uniform distribution), while any other distribution of mutants (skew) results in lower effectiveness of random sampling. However, this distribution is dependent on the program being evaluated. Since our goal is to find the mean advantage for a perfect strategy for an arbitrary program, we use the conservative distribution (uniform) of mutants for our theoretical analysis (We show later that the actual impact of this skew is limited to less than 5% for real world mutants).

The next consideration regards the minimum number of mutants required to represent the entire population of mutants. If a mutant can be distinguished from another in terms of tests that detect it, then we consider both to be *distinguishable* from each other in terms of faults they represent, and we pick a representative from each pair of distinguishable mutants. Note that, in the real world, the population of distinguishable mutants is often larger than the minimum number of mutants required to select a minimum test suite able to kill the entire mutant population. This is because while some mutants are distinguishable from others in terms of tests that detect them, there may not be any test that uniquely kills them[4]. Since this is external to the mutant population, and also because such a minimum set of mutants does not represent the original population fully (we can get away with a lower number only because the test suite is inadequate), we assume that distinguishable mutants are uniquely identified by test cases. We note however, that having inadequate test suites favors random sampling, and hence lowers the advantage for a perfect mutation reduction strategy, because random sampling can now miss the mutant without penalty. We derive the limits of mutation reduction for this system using the best strategy possible, given oracular knowledge of mutant kills.

[3]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.

[4] Consider the mutant×test matrix (1 implies the test kills the mutant) $\{\{1,1,0\},\{1,0,1\},\{0,1,1\}\}$. While all the mutants are distinguishable, just two test cases are sufficient to kill them all.

Impact of deviations of parameters:
Skew: The presence of skew reduces the effectiveness of random sampling, and hence increases the utility of the perfect strategy.
Distinguishability: Any distinguishable mutant that is not chosen by the strategy (due to not having a unique detecting test case) decreases the effectiveness of the selection strategy, decreasing its utility.

Before establishing a theoretical framework for utility of mutation reduction strategies, we must establish some terminology for the original and reduced mutant sets and their related test suites.

Terminology: Let M and $M_{strategy}$ denote the original set of mutants and the reduced set of mutants, respectively. The mutants from M killed by a test suite T are given by $kill(T, M)$ (We use M_{killed} as an alias for $kill(T, M)$). Similarly the tests in T that kill mutants in M are given by $cover(T, M)$.

$$\text{kill} : \mathbb{T} \times \mathbb{M} \rightarrow \mathbb{M}$$
$$\text{cover} : \mathbb{T} \times \mathbb{M} \rightarrow \mathbb{T}$$

The test suite $T_{strategy}$ can kill all mutants in $M_{strategy}$. That is, $kill(T_{strategy}, M_{strategy}) = M_{strategy}$. If it is minimized with respect to the mutants of the strategy, we denote it by $T^{min}_{strategy}$ (a minimal test suite with respect to a set of mutants is the smallest test suite that can kill all mutants in the set).

Two mutants m and m' are distinguished if the tests that kill them are different. $cover(T, \{m\}) \neq cover(T, \{m'\})$.

We use M^{uniq}_{killed} to denote the set of distinguished mutants from the original set such that $\forall_{m,m' \in M} cover(T, \{m\}) \neq cover(T, \{m'\})$.

The *utility* ($U_{strategy}$) of a strategy is improvement in effectiveness due to using that strategy compared to the baseline (the baseline is random sampling of the same number[5] of mutants). That is,

$$U_{strategy} = \left| \frac{kill(T^{min}_{strategy}, M)}{kill(T^{min}_{random}, M)} \right| - 1$$

Note that $T^{min}_{strategy}$ is minimized over the mutants **selected** by the strategy, and it is then applied against the **full set** of mutants (M) in $kill(T^{min}_{strategy}, M)$.

This follows the traditional evaluation of effectiveness, which goes as follows: start with the original set of mutants, and choose a subset of mutants according to the strategy. Then select a minimized set of test cases that can kill all the selected mutants. This minimized test suite is evaluated against the full set of mutants. If the mutation score obtained is greater than 99%, then the reduction is deemed to be effective. Note that we compare this score against the score of a random set of mutants of the same size, in order to handle the case where the full suite itself is not mutation adequate (or even close to adequate). Our utility answers the question: does this set of mutants better represent the test adequacy criteria represented by the full set of mutants than a random sample of the same size, and if so, by how much?

The strategy that can select the perfect set of representative mutants (a set of mutants as small as possible that does not change either the number of mutants killed or the subset of minimal tests for a suite) is called the *perfect strategy*, with its utility denoted by $U_{perfect}$[6].

We now show how to derive an expression for the maximum $U_{perfect}$ for the idealized system with the following assumptions.

1. Every distinguished mutant can be killed uniquely by a test case.
2. We assume that we have an equal number of redundant mutants for each distinguished mutant.

From here on, we refer to a set of non-distinguished mutants as a *stratum*, and the entire population is referred to as the *strata*. Given any population of detected mutants, the mutation reduction strategy should produce a set of mutants such that if a test suite can kill all of the reduced set, the same test suite can kill all of the original mutant set (remember that $T_{strategy}$ kills all mutants in $M_{strategy}$). Hence,

$$kill(T_{perfect}, M) = kill(T, M)$$

The quality of the test suite thus selected is dependent on the number of unique mutants that we are able to sample. Since we have supposed a uniform distribution, say we have x elements per stratum, and total n mutants. Our sample size s would be $p \times k$ where p is the number of samples from each stratum, and is a natural number; i.e. the sample would contain elements from each stratum, and those would have equal representation. Note that there will be at least one sample, and one strata: i.e., $s \geq 1$. Since our strata are perfectly homogeneous by construction, in practice $p = 1$ is sufficient for perfect representation, and as we shall see below, ensures maximal advantage over random sampling.

Next, we evaluate the number of different (unique) strata expected in a random sample of the same size s.
Let X_i be a random variable defined by:

$$X_i = \begin{cases} 1 & \text{if strata } i \text{ appears in the sample} \\ 0 & \text{otherwise.} \end{cases}$$

Let X be the number of unique strata in the sample, which is given by: $X = \sum_{i=1}^{k} X_i$, and the expected value of X (considering that all mutants have equal chance to be sampled) is given by:

$$E(X) = E(\sum_{i=1}^{k} X_i) = \sum_{i=1}^{k} E(X_i) = k \times E(X_1)$$

Next, consider the probability that the mutant 1 has been selected, where the sample size was $s = p \times k$:

$$P[X_i = 1] = 1 - \left(\frac{k-1}{k}\right)^{pk}$$

The expectation of X_i:

$$E(X_1) = 1 \times P(X_i = 1)$$

[5]For the rest of the paper, we require that efficiency of random sampling is the same as that of the strategy it is compared to, i.e. $|M_{strategy}| = |M_{random}|$.

[6]Where unambiguous, we shorten the subscript such as p for *perfect*, and r for *random*.

Hence, the expected number of unique strata appearing in a random sample is:

$$k \times E(X_1) = k - k \times \left(\frac{k-1}{k}\right)^{pk}$$

We already know that the number of *unique* strata appearing in each strata-based sample is k (because it is perfect, so each strata is unique). Hence, we compute the utility as the difference divided by the baseline.

$$U_{max} = \frac{k - \left(k - k \times \left(\frac{k-1}{k}\right)^{pk}\right)}{k - k \times \left(\frac{k-1}{k}\right)^{pk}} = \frac{1}{(\frac{k}{k-1})^{pk} - 1} \quad (1)$$

This converges to[7]

$$\lim_{k \to \infty} \frac{1}{(\frac{k}{k-1})^{pk} - 1} = \frac{1}{e^p - 1} \quad (2)$$

and has a maximum value when $p = 1$.

$$U_{max} = \frac{1}{e-1} \approx 58.2\% \quad (3)$$

Note that this is the *mean* improvement expected over random sampling for *uniform distribution* of redundant mutants in strata (and with oracular knowledge). That is, individual samples could still be arbitrarily advantageous (after all, the perfect strata sample itself is one potential random sample), but on average this is the expected gain over random samples.

How do we interpret this result? If you have a robust set of test cases that is able to uniquely identify distinguishable mutants, then given an arbitrary program, you can expect a perfect strategy to have at least a mean 58.2% advantage over random sample of the same efficiency in terms of effectiveness. However, if the program produces redundant mutants that are skewed, then the advantage of perfect strategy with oracular knowledge will increase (depending on the amount of skew). Similarly, if the tests are not sufficient to identify distinguishable mutants uniquely, we can expect the advantage of the perfect strategy to decrease. Finally, strategies can rarely be expected to come close to perfection in terms of classifying mutants in terms of their behavior without post hoc knowledge of the kills. Hence the advantage held by such a strategy would be much much lower (or it may not even have an advantage).

4. EMPIRICAL ANALYSIS

The above analysis provides a theoretical framework for evaluating the advantage a sampling method can have over random sampling, with a set of mutants and test suite constructed with simplifying assumptions. It also gives us an expected limit for how good these techniques could get for a uniform distribution of mutants. However, in practice, it is unlikely that real test suites and mutant sets meet our assumptions. What advantage can we expect to gain with real software systems, even if we allow our hypothetical method to make use of prior knowledge of the results of mutation analysis? To find out, we examine a large set of real-world programs and their test suites.

[7]While we can expect k to be finite for mutation testing, we are looking at the maximum possible value for this expression.

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

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

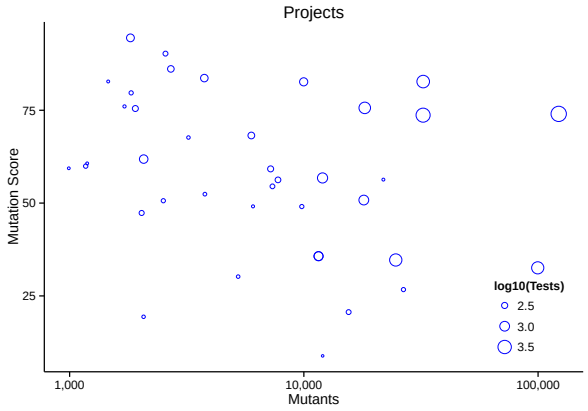

Figure 1: Distribution of number of mutants and test suites. It shows that we have a reasonable non-biased sample with both large programs with high mutation scores, and also small low scoring projects.

Our selection of sample programs for this empirical study of the limits of mutation reduction was driven by a few overriding concerns. Our primary requirement was that our results should be as representative as possible of real-world programs. Second, we strove for a statistically significant result, therefore reducing the number of variables present in the experiments for reduction of variability due to their presence.

We chose a large random sample of Java projects from Github [21][8] and the Apache Software Foundation [5] that use the popular Maven [6] 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. Tests that timed out for particular mutants were assumed to have not detected the mutant. The tests that completely failed to detect any of the mutants were eliminated as well, as these were redundant to our analysis. We also removed all projects with trivial test suites, leaving only those that had at least 100

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

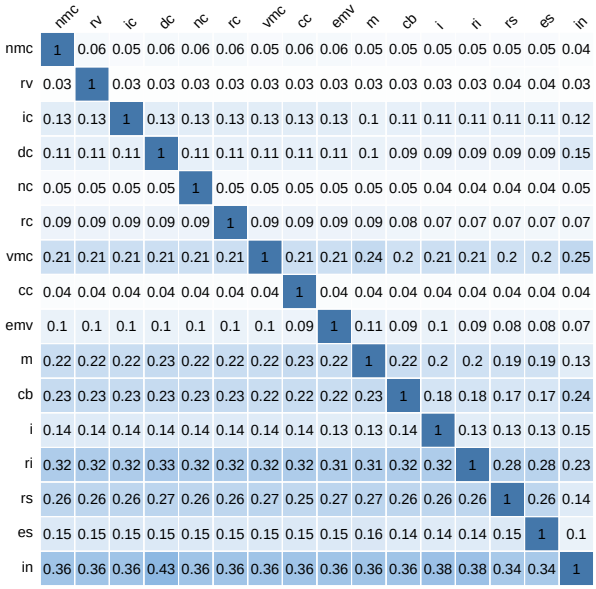

Figure 2: Subsumption rate between operators. Note that subsumption is not a symmetrical relation. No operators come close to full subsumption. This suggests that **none of the operators studied are redundant**.

test cases. This left us with 39 projects. The projects are given in Table 2.

Note that we have a much larger set of large projects (39 projects with a mean 7720 LOC) than previous studies such as Namin et al. [48] (7 projects with a mean 312 LOC), Zhang et al. [56] (7 projects with a mean 312 LOC), Zhang et al. [55] (7 projects with a mean 15083 LOC), and Zhang et al. [54] (12 projects with a mean 6209 LOC).

Similarly, our test suites are at least comparable to other studies, (mean=750.564 test cases, sd=1184.335) compared to previous studies — Namin et al. [48] (mean=3115.286, sd=1572.038), Zhang et al. [56] (mean=3115.286, sd=1572.038), Zhang et al. [55] (mean=3115.286, sd=1572.038), and Zhang et al. [54] (mean=81, sd=29.061).

In terms of mutation scores, our sample (mean=0.58, sd=0.214), is also comparable to previous studies such as Namin et al. [48] (mean=0.322, sd=0.318), Zhang et al. [56] (mean 0.831, sd=0.055), Zhang et al. [55] (mean 0.831, sd=0.055), and Zhang et al. [54] (mean 0.529, sd=0.256). In summary, our set of projects is fairly large, has large test suites, and has comparable mutation scores to previous studies that handled similar sized projects. Other studies of a similar nature had either smaller test suites, smaller sized subjects, or a much smaller number of subjects. Further, our study includes both low and high mutation scores, as required to demonstrate the effectiveness of our technique.

We ran our mutation analysis on this set of projects using PIT [12], a tool popular in industry and used in several previous studies [15,25,28,47]. Since the operators provided by PIT were limited, we extended PIT to provide new operators similar to those of other mutation systems (which were later accepted into the PIT mainline). In doing so, we made sure that the new operators introduced were non-redundant with PIT's existing operators. The set of operators that we used is provided in Table 1, and the redundancy matrix for the full operator set is given in Figure 2. A mutant *m1* is deemed to subsume another, say *m2* if any tests that kills

Table 2: The projects mutants and test suites

Project	$\|M\|$	M_{killed}	M^{uniq}_{killed}	$\|T\|$	$\|T_{min}\|$
events	1171	702	59	180	33.87
annotation-cli	992	589	110	109	38.97
mercurial-plugin	2069	401	102	138	61.77
fongo	1461	1209	175	113	70.73
config-magic	1188	721	204	112	74.55
clazz	5242	1583	151	140	64.00
ognl	21852	12308	2990	114	85.43
java-api-wrapper	1715	1304	308	125	107.04
webbit	3780	1981	325	147	116.93
mgwt	12030	1065	168	101	90.65
csv	1831	1459	411	173	117.97
joda-money	2512	1272	236	173	128.48
mirror	1908	1440	532	301	201.21
jdbi	7754	4362	903	277	175.57
dbutils	2030	961	207	224	141.53
cli	2705	2330	788	365	186.24
commons-math1-l10n	6067	2980	219	119	109.02
mp3agic	7344	4003	730	206	146.79
asterisk-java	15530	3206	451	214	196.32
pipes	3216	2176	338	138	120.00
hank	26622	7109	546	171	162.88
java-classmate	2566	2316	551	215	196.57
betwixt	7213	4271	1198	305	206.35
cli2	3759	3145	1066	494	303.86
jopt-simple	1818	1718	589	538	158.37
faunus	9801	4809	553	173	146.11
beanutils2	2071	1281	465	670	181.00
primitives	11553	4125	1365	803	486.71
sandbox-primitives	11553	4125	1365	803	488.56
validator	5967	4070	759	383	264.35
xstream	18030	9163	1960	1010	488.25
commons-codec	9983	8252	1393	605	444.69
beanutils	12017	6823	1570	1143	556.67
configuration	18198	13766	4522	1772	1058.36
collections	24681	8561	2091	2241	938.32
jfreechart	99657	32456	4686	2167	1696.86
commons-lang3	32323	26741	4479	2456	1998.11
commons-math1	122484	90681	17424	5881	4009.98
jodatime	32293	23796	6920	3973	2333.49

m1 is guaranteed to kill *m2*. This is extended to mutation operators whereby the fraction of mutants in *o1* killed by test cases that kills all mutants in *o2* is taken as the degree of subsumption of *o1* by *o2*. The matrix shows that the maximum subsumption was just 43% — that is, none of the operators were redundant. For a detailed description of each mutation operator, please refer to the PIT documentation [13]. 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. The mutation scores along with the sizes of test suites are given in Figure 1.

It is of course possible that our results may be biased by the mutants that PIT produces, and it may be argued that the tool we use produces too many redundant mutants, and hence the results may not be applicable to a better tool that reduces the redundancy of mutants. To account for this argument, we run our experiment in two parts, with similar procedures but with different mutants. For the first part, we use the detected mutants from PIT as is, which provides us with an upper bound that a practicing tester can expect to experience, now, using an industry-accepted tool. For the second part, we choose only distinguishable mutants [2] from the original set of detected mutants. What this does is to reduce the number of samples from each stratum to 1, and hence eliminate the skew in mutant population. Note that this requires post-hoc knowledge of mutant kills (not just that the mutants produce different failures, but also that

available tests in the suite can distinguish between both), and is the best one can do for the given projects to enhance the utility of any strategy against random sampling. We provide results for both the practical and more theoretically interesting distinguishable sets of mutants. Additionally, for the few projects that have plausibly mutation-adequate test suites, we computed the possible advantage, in case adequacy is required for large gain over random sampling.

4.1 Experiment

Our task is to find the $U_{perfect}$ for each project. The requirements for a perfect strategy are simple:

1. The mutants should be representative of the full set. That is,
$$kill(T_p, M) = kill(T, M)$$
2. The mutants thus selected should be non redundant. That is,
$$\forall_{m \in M_p} kill(T_p, M_p \setminus \{m\}) \subset kill(T_p, M_p)$$

The minimal mutant set suggested by Ammann et al. [2] satisfies our requirements for a perfect strategy, since it is representative — a test suite that can kill the minimal mutants can kill the entire set of mutants — and it is non-redundant with respect to the corresponding minimal test suite.

Ammann et al. [2] observed that the cardinality of a minimal mutant set is the same as the cardinality of the corresponding minimal test suite. That is,

$$|M^{min}_{perfect}| = |MinTest(T, M)| = |T^{min}_{all}|$$

Finding the true minimal test suite for a set of mutants is NP-complete[9]. The best possible approximation algorithm is Chvatal's [11], using a greedy algorithm where each iteration tries to choose a set that covers the largest number of mutants. This is given in Algorithm 1, and achieves an approximation ratio of $H(|M|)$[10]

Since it is an approximation, we average the greedily estimated minimal test suite size over 100 runs. The variability is given in Figure 3, ordered by the size of minimal test suite. Note that there is very little variability, and the variability decreases as the size of test suite increases. All we need now is to find the effectiveness of random sampling for the same number of mutants as produced by the *perfect* strategy.

Next, we randomly sample $|M^{min}_{perfect}|$ mutants from the original set M_{random}, obtain the minimal test suite of this sample T^{min}_{random}, and find the mutants from the original set that are killed by this test suite $kill(T^{min}_{random}, M)$, which is used to compute the utility of perfect strategy with respect to that particular random sample. The experiments were repeated 100 times for each project, and averaged to compute $U_{perfect}$ for the project under consideration.

5. RESULTS

[9]This is the *Set Covering Problem* [2] which is NP-Complete [34].

[10]$H(n)$ is the n-th *harmonic number*. It is given by

$$H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq lnn + 1$$

Algorithm 1 Finding the minimal test suite

```
function MinTest(Tests, Mutants)
    T ← Tests
    M ← kill(T, Mutants)
    T_min ← ∅
    while T ≠ ∅ ∨ M ≠ ∅ do
        t ← random(max_t |kill({t}, M)|)
        T ← T \ {t}
        M ← kill(T, Mutants)
        T_min ← T_min ∪ {t}
    end while
    return T_min
end function
```

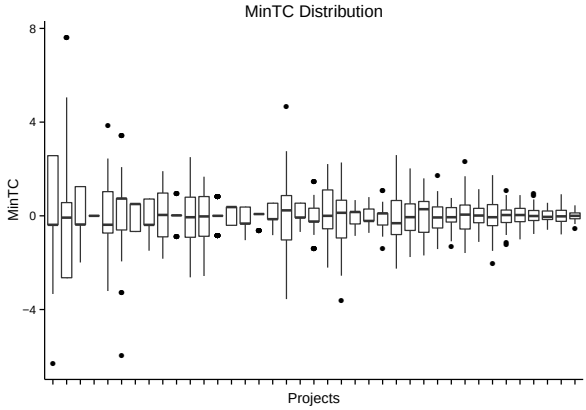

Figure 3: Distribution of variation of minimum test cases for each sample as a percentage difference from the mean ordered by mean minimum test suite size. There is very little variation, and the variation decreases test suite size.

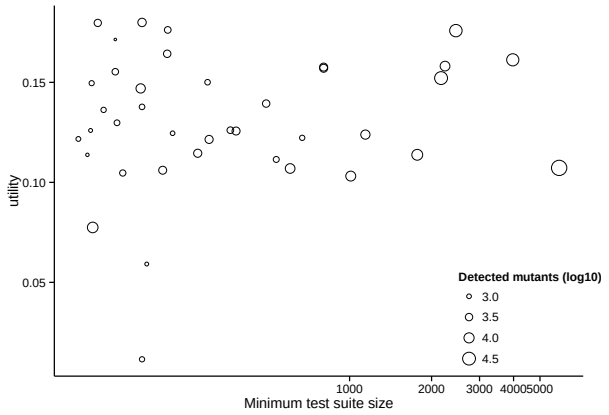

Figure 4: The figure plots utility (y-axis) against the minimum test suite size (log10). The size of bubbles represent the magnitude of detected mutants (log10). The figure suggests that there is no correlation between utility and minimum test suite size.

Table 3: The maximum utility achievable by a perfect strategy for each project

Project	$\|kill(T,M)\|$	$\|kill(T_r,M)\|$	U_{perf}
events	702	662.97	0.06
annotation-cli	589	529.51	0.11
mercurial-plugin	401	342.91	0.17
fongo	1209	1052.99	0.15
config-magic	721	640.91	0.13
clazz	1583	1402.39	0.13
ognl	12308	11426.09	0.08
java-api-wrapper	1304	1148.52	0.14
webbit	1981	1793.96	0.10
mgwt	1065	949.96	0.12
csv	1459	1282.93	0.14
joda-money	1272	1257.55	0.01
mirror	1440	1252.50	0.15
jdbi	4362	3914.73	0.11
dbutils	961	854.83	0.12
cli	2330	2069.84	0.13
commons-math1-l10n	2980	2527.66	0.18
mp3agic	4003	3620.41	0.11
asterisk-java	3206	2754.69	0.16
pipes	2176	1884.73	0.16
hank	7109	6200.08	0.15
java-classmate	2316	1969.76	0.18
betwixt	4271	3809.19	0.12
cli2	3145	2760.66	0.14
jopt-simple	1718	1546.21	0.11
faunus	4809	4078.22	0.18
beanutils2	1281	1141.73	0.12
primitives	4125	3565.83	0.16
sandbox-primitives	4125	3563.85	0.16
validator	4070	3616.71	0.13
xstream	9163	8307.12	0.10
commons-codec	8252	7455.50	0.11
beanutils	6823	6071.53	0.12
configuration	13766	12359.89	0.11
collections	8561	7392.63	0.16
jfreechart	32456	28171.19	0.15
commons-lang3	26741	22742.46	0.18
commons-math1	90681	81898.25	0.11
jodatime	23796	20491.96	0.16

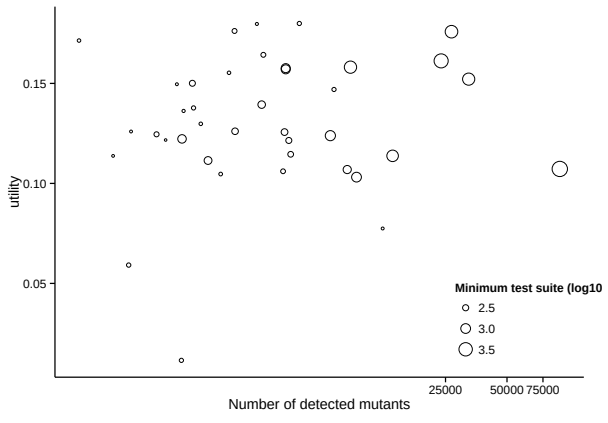

Figure 5: The figure plots utility (y-axis) against the number of detected mutants. The size of bubbles represent the magnitude of minimum test suite size (log10). The figure suggests that there is no correlation between utility and number of detected mutants.

Table 4: The maximum utility achievable by a perfect strategy for each project using distinguishable mutants

Project	$\|kill(T,M)\|$	$\|kill(T_r,M)\|$	U_{perf}
events	59	49.15	0.20
annotation-cli	110	93.68	0.18
mercurial-plugin	102	80.95	0.26
fongo	175	145.13	0.21
config-magic	204	171.60	0.19
clazz	151	129.24	0.17
ognl	2990	2835.77	0.05
java-api-wrapper	308	259.87	0.19
webbit	325	280.89	0.16
mgwt	168	140.60	0.20
csv	411	349.30	0.18
joda-money	236	230.76	0.02
mirror	532	444.17	0.20
jdbi	903	783.99	0.15
dbutils	207	170.60	0.21
cli	788	688.05	0.15
commons-math1-l10n	219	177.86	0.23
mp3agic	730	639.01	0.14
asterisk-java	451	372.25	0.21
pipes	338	288.41	0.17
hank	546	465.52	0.17
java-classmate	551	450.46	0.22
betwixt	1198	1055.30	0.14
cli2	1066	903.30	0.18
jopt-simple	589	514.36	0.15
faunus	553	467.03	0.18
beanutils2	465	392.30	0.19
primitives	1365	1155.09	0.18
sandbox-primitives	1365	1155.01	0.18
validator	759	647.36	0.17
xstream	1960	1691.84	0.16
commons-codec	1393	1192.29	0.17
beanutils	1570	1341.04	0.17
configuration	4522	3934.21	0.15
collections	2091	1750.05	0.19
jfreechart	4686	3910.15	0.20
commons-lang3	4479	3663.98	0.22
commons-math1	17424	15139.90	0.15
jodatime	6920	5801.10	0.19

Table 5: The correlation of utility for all mutants, killed mutants, mutation score, and minimum test suite size, based on both full set of mutants, and also considering only distinguished mutants

	R^2_{all}	$R^2_{distinguished}$
M	-0.02	-0.03
M_{kill}	-0.03	-0.01
M_{kill}/M	-0.02	-0.00
T^{min}	-0.01	-0.02

5.1 All Mutants

Our results are given in Table 3. We found that the largest utility achieved by the perfect strategy was 17.997%, for project *faunus*, while the lowest utility was 1.153%, for project *joda-money*. The mean utility of the perfect strategy was 13.078%. A one sample *u-test* suggests that 95% of projects have maximum utility between 12.218% and 14.26% ($p < 0.001$). The distribution of utility for each project is captured in Figure 6. Projects are sorted by average minimum test suite size.

One may wonder if the situation improves with either test suite size or project size. We note that the utility U_p has low correlation with total mutants, detected mutants (shown in Figure 5), mutation score, and minimum test suite size

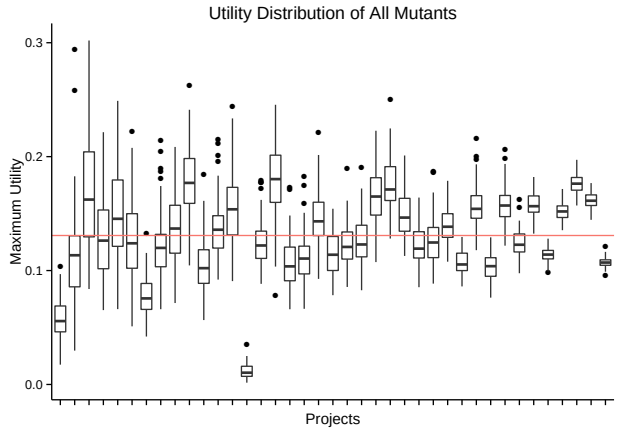

Figure 6: Using all mutants.

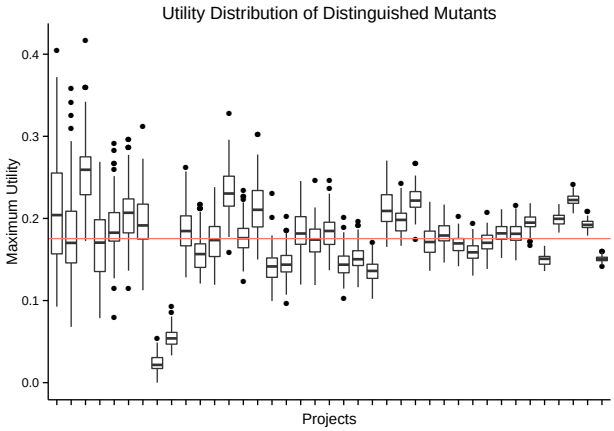

Figure 7: Using distinguished mutants.

Figure 8: Distribution of mean utility using distinguished mutants across projects. The projects are ordered by the cardinality of minimum test suite. The red line indicates the mean of all observations.

(shown in Figure 4). The correlation factors are given in Table 5.

An analysis of variance (ANOVA) to determine significant variables affecting $U_{perfect}$ suggests that the variability due to project is a significant factor ($p < 0.001$) and interacts with $kill(T_{random}, M)$ strongly.

$$\mu\{U_p\} = project + kill(T_r, M) + project \times kill(T_r, M)$$

Project has a correlation of 0.682 with the $U_{perfect}$, and the combined terms have a correlation of 0.9995 with $U_{perfect}$.

5.2 Distinguishable Mutants

Our results are given in Table 4. We found that the largest utility achieved by the perfect strategy was 26.159%, for project *mercurial-plugin*, while the lowest utility was 2.283%, for project *joda-money*.

The mean utility of the perfect strategy was 17.545%. A one sample u-test showed that 95% of projects have a maximum utility between 16.912% and 18.876% ($p < 0.001$).

The utility distribution for each project is captured in Figure 7. The projects are sorted by the average minimum test suite size.

This situation does not change with either test suite or project size.

The utility U_p has low correlation with total mutants, detected mutants, mutation score, and minimum test suite size. The correlation factors are given in Table 5.

An analysis of variance (ANOVA) to determine significant variables affecting $U_{perfect}$ found that the variability due to project is a significant factor ($p < 0.001$) and strongly interacts with $kill(T_{random}, M)$.

$$\mu\{U_p\} = project + kill(T_r, M) + project \times kill(T_r, M)$$

Project has a correlation of 0.734 with the $U_{perfect}$, and the combined terms have a correlation of 0.9994 with $U_{perfect}$.

5.3 Adequate Mutants

Finally, one may ask if adequacy has an impact on the effectiveness of selection strategies. Following the industry practice of deeming well-tested projects adequate after discounting equivalent mutants [48, 54–56], we chose large well tested projects that had at least 10,000 mutants and a mutation score of at least 70% (in the range of similar studies above[11]) which were deemed adequate. We evaluated the utility for *configuration, commons-lang3, commons-math1, jodatime* and found that they have a mean maximum utility of 13.955%. These same projects have a distinguished mean maximum utility of 17.893%. This suggests that adequacy does not have a noticeable impact on the effectiveness of selection strategies.

6. DISCUSSION

Mutation analysis is an invaluable tool that is often difficult to use in practice due to hefty computational requirements. There is ongoing and active research to remedy this situation using different mutation reduction strategies. Hence, it is important to understand the amount by which one can hope to improve upon the simplest baseline strategy for reduction — pure random sampling.

Our theoretical analysis of a simple idealized system finds a mean improvement of 58.2% over random sampling for a mutation reduction strategy with oracular knowledge of mutation kills given a uniform distribution of mutants. This serves as an upper bound of what any known mutation reduction strategy could be expected to achieve (under the assumption that the mutant distribution is reasonably close to uniform).

Our empirical analysis using a large number of open source projects reveals that the practical limit is much lower, however, on average only 13.078% for mutants produced by PIT. Even if we discount the effects of skew, by using only distinguished mutants, the potential improvement is restricted to 17.545% on average.

It is important to distinguish the different questions that the theory and empirical analysis tackle. The theoretical limit shows the best that can be done by a perfect mutation strategy given the worst distribution of mutants one may encounter. On the other hand, the empirical analysis finds the average utility of a perfect strategy without regard to the distribution of mutants in different programs. However, given that the effects of skew were found to be rather weak (only 4.467%) the theoretical bound is reasonable for the empirical question too.

The empirical upper bounds on gain in utility are surpris-

[11] See footnote 8

ingly low, and call into question the effort invested into improving mutation reduction strategies. Of course, one can still point out that random sampling is subject to the vagaries of chance, as one can get arbitrarily good or bad samples. However, our results suggest that the variance of individual samples is rather low, and the situation improves quite a bit with larger projects — e.g. the variance of *commons-math1* is just 0.397%. Hence the chances for really bad samples are very low in the case of projects large enough to really need mutant reduction, and drop quickly as the number of test cases increases. One may wonder if the adequacy of test suites has an impact, but our analysis of projects with adequate test suites suggests that there is very little difference due to adequacy ($U_{perfect}$ =13.955%). In general, using accepted standard practices for statistical sampling to produce reasonably-sized random mutant samples should be practically effective for avoiding unusually bad results due to random chance. The added advantage is that random sampling is easy to implement and incurs negligible overhead.

We note that our framework is applicable not only to selective mutation, but also to mutation implementers looking to add new mutators. Say a mutation implementor has a perfect set of mutation operators such that their current set of mutants do not have any redundant mutants (practically infeasible given our shallow understanding of mutant semiotics). Even if we consider the addition of a new set of random mutants that *do not* improve the mutation set at all, in that they are redundant with respect to the original set (rare in practice, given that we are introducing new mutants), the maximum disadvantage thus caused is bounded by our limit (18.876% upper limit for 95% of projects). However, at least a few of the new mutants can be expected to improve the representativeness of a mutation set compared to the possible faults. Since we can't bound the number of distinguishable mutants that may be introduced, there is no upper bound for the maximum advantage gained by adding new mutation operators. Adding new operators is especially attractive in light of recent results showing classes of real faults that are not coupled to any of the operators currently in common use [33].

Our previous research [24] suggests that a constant number of mutants (a theoretical maximum of $9,604$, and 1,000 in practice for 1% accuracy) is sufficient for computing mutation score with high accuracy irrespective of the total number of mutants. This suggests that sampling will lead to neither loss of effectiveness nor loss of accuracy, and hence addition of new mutation operators (and sampling the required number of mutants) is potentially a very fruitful endeavour.

7. 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.

Threats due to approximation: We use the greedy algorithm due to Chvatal [11] for approximating the minimum test suite size. While this is guaranteed to be $H(|M|)$ approximate, there is still some scope for error. We guard against this error by taking the average of 100 runs for each observation. Secondly, we used random samples to evaluate the effectiveness of random sampling. While we have used 100 trials each for each observation, the possibility of bias does exist.

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.

Bias due to tool used: For our study, 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. We have done our best to extend PIT to provide a reasonably sufficient set of mutation operators, ensuring also that the mutation operators were non-redundant. Further, we have tried to minimize the impact of redundancy by considering the effect of distinguished mutants. There is still a possibility that the kind of mutants produced may be skewed, which may impact our analysis. Hence, this study needs to be repeated with mutants from diverse tools and projects in future.

To ensure that our study can be replicated easily with different data sources, we have published our empirical observations from PIT as an *R* package, and the computations producing the paper as a *Knitr* [53] document. The complete research, along with instructions to build it, is available [22].

8. CONCLUSION

Our research suggests that blind random sampling of mutants is highly effective compared to the best achievable bound for mutation reduction strategies, using perfect knowledge of mutation analysis results, and there is surprisingly little room for improvement.

Our theoretical investigation suggests a mean advantage of 58.2% for a perfect mutation reduction strategy with oracular knowledge of kills over random sampling given an arbitrary program, under the assumption of no skew in redundant mutants. Empirically, we find a much lower advantage 13.078% for a perfect reduction strategy with oracular knowledge. Even if we eliminate the effects of skew in redundant mutant population by considering only distinguished mutants, we find that the advantage of a perfect mutation reduction strategy is only 17.545% in comparison to random sampling. The low impact of skew (4.467%) suggests that our simplifying assumptions for theoretical analysis were not very far off the mark. The disparity between the theoretical prediction and empirical results is due to the inadequacies of real world test suites, resulting in a much smaller minimum mutant set than the distinguishable mutant set. We note that mutation reduction strategies routinely claim high reduction factors, and one might expect a similar magnitude of utility over random sampling, which fails to materialize either in theory or practice.

The second takeaway from our research is that a researcher or an implementer of mutation testing tools should consider the value of implementing a mutation reduction strategy carefully given the limited utility we observe. In fact, our research [23] suggests that popular operator selection strategies we examined have reduced utility compared to random sampling, and even strata sampling techniques based on program elements seldom exceed a 10% improvement. Given

that the variability due to projects is significant, a testing practitioner would also do well to consider whether the mutation reduction strategy being used is suited for the particular system under test (perhaps based on historical data). Random sampling of mutants is not extremely far from an empirical upper bound on an ideal mutation reduction strategy, and has the advantage of having little room for an unanticipated bias due to a "clever" selection method that might not work well for a given project. The limit reported here is based on using full knowledge of the mutation kill matrix, which is difficult to attain in practice.

Perhaps the most important takeaway from our research is that it is possible to improve the effectiveness of mutation analysis, not by removing mutation operators, but rather by further research into newer mutation operators (or new categories of mutation operators), and also by research into higher order mutation [26]. Our research suggests that the maximum reduction in utility is just 23.268%, while there is no limit to the achievable improvement.

9. REFERENCES

[1] A. T. Acree, Jr. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1980.

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

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

[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*, 32(8), 2006.

[5] Apache Software Foundation. Apache commons. http://commons.apache.org/.

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

[7] 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.

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

[9] 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*, pages 220–233. ACM, 1980.

[10] T. A. Budd, R. J. Lipton, R. A. DeMillo, and F. G. Sayward. *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[11] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

[12] H. Coles. Pit mutation testing. http://pitest.org/.

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

[14] 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*, pages 158–171. ACM, 1996.

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

[16] 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*, pages 142–151. IEEE, 1988.

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

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

[19] A. Derezińska. A quality estimation of mutation clustering in c# programs. In *New Results in Dependability and Computer Systems*, pages 119–129. Springer, 2013.

[20] A. Derezińska. Toward generalization of mutant clustering results in mutation testing. In *Soft Computing in Computer and Information Science*, pages 395–407. Springer, 2015.

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

[22] R. Gopinath. Replication: Limits of mutation reduction strategies. http://eecs.osuosl.org/rahul/icse2016/.

[23] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. Do mutation reduction strategies matter? Technical report, Oregon State University, Aug 2015. http://hdl.handle.net/1957/56917.

[24] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce. How hard does mutation analysis have to be, anyway? In *International Symposium on Software Reliability Engineering*. IEEE, 2015.

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

[26] M. Harman, Y. Jia, and W. B. Langdon. A manifesto for higher order mutation testing. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 80–89. IEEE, 2010.

[27] S. Hussain. Mutation clustering. Master's thesis, King's College London, Strand, London, UK, 2008.

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

[29] C. Ji, Z. Chen, B. Xu, and Z. Zhao. A novel method of mutation clustering based on domain analysis. In *SEKE*, pages 422–425, 2009.

[30] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 249–258. IEEE, 2008.

[31] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10):1379–1393, Oct. 2009.

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

[33] 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*, pages 654–665, Hong Kong, China, 2014. ACM.

[34] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 1972.

[35] B. Kurtz, P. Ammann, and J. Offutt. Static analysis of mutant subsumption. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10, April 2015.

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

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

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

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

[40] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[41] 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.

[42] A. S. Namin and J. H. Andrews. Finding sufficient mutation operators via variable reduction. In *Workshop on Mutation Analysis*, page 5, 2006.

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

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

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

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

[47] 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*, FSE 2014, pages 246–256, New York, NY, USA, 2014. ACM.

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

[49] J. Strug and B. Strug. Machine learning approach in mutation testing. In *Testing Software and Systems*, pages 200–214. Springer, 2012.

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

[51] 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.

[52] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Purdue University, West Lafayette, IN, USA, 1993. UMI Order No. GAX94-20921.

[53] Y. Xie. *Dynamic Documents with R and knitr*, volume 29. CRC Press, 2013.

[54] 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.

[55] 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.

[56] 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*, NY, USA, 2010. ACM.