Mutation Reduction Strategies Considered Harmful

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

***Abstract*—Mutation analysis is a well-known yet unfortunately costly method for measuring test suite quality. Researchers have proposed numerous mutation reduction strategies in order to reduce the high cost of mutation analysis, while preserving the representativeness of the original set of mutants.**

As mutation reduction is an area of active research, it is important to understand the limits of possible improvements. We theoretically and empirically investigate the limits of improvement in effectiveness from using mutation reduction strategies compared to random sampling. Using real-world open source programs as subjects, we find an absolute limit in improvement of effectiveness over random sampling —13.078%.

Given our findings with respect to absolute limits, one may ask: how effective are the extant mutation reduction strategies? We evaluate the effectiveness of multiple mutation reduction strategies in comparison to random sampling. We find that none of the mutation reduction strategies evaluated —many forms of operator selection, and stratified sampling (on operators or program elements) —produced an effectiveness advantage larger than 5% in comparison with random sampling.

Given the poor performance of mutation selection strategies — they may have a negligible advantage at best, and often perform worse than random sampling – we caution practicing testers against applying mutation reduction strategies without adequate justification.

***Index Terms*—Mutation Analysis, Software Testing**

NOMENCLATURE

Efficiency	The amount of reduction achieved by the selection procedure. Given by $\|M\|/\|M_{selected}\|$
Effectiveness	The ratio between the unique mutants in the selected set of mutants to that of the unique mutants in complete set of mutants.
Utility	The improvement in effectiveness due to a technique when compared to mean random sample of the same efficiency.
Adequacy	A test suite is adequate for a set of mutants if it is able to kill all mutants in that set.
Mutation strata	In sampling terminology, a *stratum* is a non overlapping subgroup that shares some characteristic. For our analysis we consider each set of non-distinguished mutants as separate strata.
Reduction strategy	A strategy that seeks to minimize the number of mutants by identifying representative mutants for each strata by predicting how mutants will perform against different test cases.
Oracular strategy	The theoretical limit for any reduction strategy achieved by considering the actual behavior of mutants against test cases, used for the purpose of evaluating performance of mutation reduction strategies.
Minimal test suite	A test suite is minimal when removing any test case results in a reduction in mutant score.
Minimal mutant set	A minimal mutant set has a bijective correspondence with a *minimal test suite* in terms of kills.
$\text{kill} : \mathbb{T} \times \mathbb{M} \to \mathbb{M}$	The number of *mutants* from M killed by the test suite T.
$\text{cover} : \mathbb{T} \times \mathbb{M} \to \mathbb{T}$	The number of *tests* in T that kill mutants in M.
$M_{strategy}$	The reduced set of mutants due to applying a mutant reduction strategy.
$T_{strategy}$	A test suite adequate for $M_{strategy}$.
$T^{min}_{strategy}$	A minimized test suite corresponding to mutant set $M_{strategy}$. Further, T^{min} corresponds to M.
Distinguished	Two mutants m and m' are distinguished if the tests that kill them are different (also called *unique*). That is, $cover(T, \{m\}) \neq cover(T, \{m'\})$.
M_{uniq}	The set of distinguished mutants from the original set of detected mutants M_{killed} such that $\forall_{m,m' \in M} cover(T, \{m\}) \neq cover(T, \{m'\})$.

I. INTRODUCTION

Mutation analysis is the best known approach for evaluating the quality of test suites. It involves producing a set of *mutants* (programs with small differences from the original program), which is then used to evaluate the effectiveness of test suites at detecting the 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 a 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.

Many strategies have been proposed to deal with the problematic cost of mutation analysis. These have been classified [8] into *do faster*, *do smarter*, and *do fewer* approaches, which correspond respectively to techniques improving the speed of execution of a single mutant, techniques parallelizing the evaluation of mutants, and techniques reducing the number of mutants evaluated.

Many *do fewer* strategies — mutation reduction methods that aim to select choose a smaller, representative, subset of mutants to evaluate — have been investigated in the past. These can be broadly divided into two groups. First, there are operator selection strategies, which seek to identify the smallest subset of mutation operators that generates the most useful mutants [9], [10]. Alternatively, there are strata sampling [11], [12] techniques, which propose to identify groups of mutants that have high mutual similarity to reduce the number of mutants without sacrificing representativeness or diversity [13], [14]. Other actively studied methods [15] include using clustering [16], [17], static analysis [18], [19] and other intelligent techniques [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 approach the value of mutation reduction in two ways. The first way is via an evaluation of the absolute limit, in terms of improvement in effectiveness, that an oracular strategy (an unrealistic strategy with access to the result of mutant kills) can achieve. The second approach is via evaluation of the effectiveness of actual common mutation reduction strategies using multiple methods.

For the first part, we consider a simple theoretical framework that allows us to evaluate the improvement in effectiveness provided by the best mutation reduction possible (under the simplifying assumptions of uniform redundancy of faults in mutants, and sufficiency of tests to distinguish faults uniquely), given oracular knowledge of mutation kills. This provides us with an approximate upper bound for the *mean effectiveness*[1] that can be obtained in this simple theoretical system, and suggests that a similar upper bound for *mean effectiveness* may exist for real world systems. Next, 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 mean effectiveness limits under real-world conditions.

For the second part, we evaluate the current mutation reduction strategies to determine the advantage they proffer with respect to random sampling. First, we use **traditional effectiveness** of each strategy, as given in the first part. This involves using a given strategy to choose a reduced set of mutants from the set of detected mutants in the original population, and choosing a *minimum set of test cases* that can kill all the mutants in the reduced set. The minimum set of test cases is then evaluated against the detected mutants from the original set of mutants to determine the effectiveness of the selected test set. This is taken as the effectiveness of the reduced set mutants. Indeed mutants produced differ in terms of their utility. We know that a large number of mutants are redundant [21], which can skew results[2].

While the test suites of many open source programs are far from adequate,[3] they should satisfy a different requirement: namely, each test was almost always added through considerable manual labor [22], and was at least believed to be useful (the number of test cases correlates with the quality of software [22]). 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[4], and perform better than random selection. This criterion — the cardinality of minimum test suite (which is the same as the cardinality of the corresponding minimum mutant set) — was recently suggested by Ammann et al. [23] as a measure of quality of a test suite. Hence we use the **size of the minimum test suite**, which is the same as the **size of the minimum mutant set** as the second criterion to judge reduction strategies.

All test cases are not created equal. Some check large and complex conditions, while others check only for relatively trivial conditions. Hence using a single mutant killed by that test case to represent each test case (as we do above, using the *minimum mutant set* as the criterion for evaluation) is susceptible to skew. Our understanding of mutation semiotics[5] is far from sufficient to specify the actual semantic impact of a mutant. However, we have a reasonably good proxy. We know that *test case assertions*[6] play a large role in the effectiveness of a test case [24], [25], [26]. So we use **test case assertion counts** as a proxy for the effectiveness of a test case.

There could be other ways to evaluate mutation reduction strategies; for example one could imagine a criterion that encourages hardest to detect — yet not-equivalent — mutants, or one 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.

[1] The *mean effectiveness* here is the average effectiveness that can be expected when considering the set of all valid programs. There can be specific instances where the particular features of a given program may lead to arbitrarily better effectiveness if it produces highly skewed mutants. We also note that the random sample we compare to is the expected sample. It is possible (but unlikely as the sample size increases) for a particular random sample of mutants to be either extremely good or bad in terms of the number of redundant mutants.

[2] Indeed, one of the criticisms leveled against our research was that mutants may be of different strengths in terms of the tests they kill.

[3] Mutation adequate test suites are suites with maximal mutation coverage; usually much less than 100% due to equivalent mutants.

[4] We only approximate a minimum suite with greedy methods. See Algorithm 1 for details.

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

[6] For the remainder of this paper, we use *assertions* to mean exclusively test case assertions.

Does the adequacy of test suites have an impact on our empirical results? While our empirical analysis was carried out with strong, but less than adequate test suites, we believe that using mutation adequate test suites will not change our results significantly.

This paper makes the following contributions:

- We show that under simplifying assumptions of uniform redundancy of faults in mutants, and sufficiency of test sets to distinguish faults uniquely, no mutation reduction strategy can have a *mean effectiveness* improvement of more than 58.2% when compared to random sampling, where the *mean effectiveness* is the expected effectiveness when considering the set of all valid programs.
- We show an empirical upper limit for *mean effectiveness*, through the evaluation of a large number of open source projects, with 13.1% mean effectiveness; for 95% of projects the maximum utility is between 12.2% and 14.3% (*one sample u-test* $p < 0.001$)[7].
- 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: traditional, size of minimum set of mutants, and effectiveness of selected minimum test suites, using assertions to evaluate the different reduction strategies. Our evaluation results are applicable to both real-world non-adequate test suites, and traditional mutation adequate test suites.
- We find that extant mutation reduction strategies seldom perform better and are often harmful to effectiveness when compared to simple random sampling of mutants.

This is an extension of our previous work [28], where we showed that there is an absolute empirical and theoretical limit (13.1% on average) to the improvement in mutation effectiveness that is possible using any mutation reduction strategy possible, under oracular knowledge. This paper extends that result by evaluating the *actual* improvement achieved by extant mutation reduction strategies. We examine both operator selection methods and strata sampling methods, and our research suggests that even the relatively modest advantage that theory suggests is possible is rarely achieved in practice. Further, we also account for the possible differences in utility of different mutants by incorporating both test utility and assert utility.

Our results can be interpreted as showing that, while any advantage gained over random sampling is indeed an advantage, however small, these benefits may not be worth their costs. Our understanding of mutant semiotics, as noted before, is very limited, and certainly insufficient to infer whether each kind of selection proposed is advantageous. Indeed, our results suggest that one is often led astray in the effort to find a good heuristic, ending up with methods that decrease the effectiveness of the mutant set compared to a simple random sample (which on its own is a perfectly reasonable approach [29]). We note that even elimination of operators based on subsumption is not [30] a foregone conclusion. The effort directed towards mutant selection mechanisms should be carefully weighed against the potential maximum utility, the known utility of existing approaches, and the risk associated with making results less useful due to biased sampling.

[7]We use the non-parametric Mann-Whitney *u-test* as it is more robust to normality assumptions and outliers [27]. The more common *t-test* yields similar results.

There could be other valid reasons for choosing a selective mutation strategy such as reduction of execution cost of specific mutants, avoidance of equivalent mutants, or selection of specific bug types. Our results only concern mutation reduction for a specific purpose: **reduction of redundant mutants.**

Our research supports the need for further research into new mutation operators. That is, we have shown that even the best selective strategy can at best have a limited improvement in effectiveness, while a bad selective strategy can lead to unlimited decrease in effectiveness (for example, by choosing mutants representing only a single fault).

We have a much more positive scenario if we implement new operators instead, and sample from the larger population. Here, a plausible scenario is that each new operator actually manages to introduce new faults. In such a case, we can improve effectiveness arbitrarily (under the assumption that the sample size is larger than the number of unique faults). That is, the improvement in effectiveness is unlimited. On the other hand, say the new operators did not add any new faults, and the mutants introducd were very similar to existing operators. In this scenario, a random sample of mutants from the larger population would have same number of unique faults as that of a random sample of mutants from the original mutant population on average. That is, in the worst case we we can expect no or limited disadvantage. To summarize, if addition of new operators goes well, there is possibility of unlimited improvement in effectiveness, while in the worst case, there is limited or no decrease in effectiveness.

This asymmetry between removing and adding operators results from difference in the populations from which the random comparison sample is drawn. For operator selection, the optimally chosen set is always a subset of the original population. Because the random sample is drawn from the original population, it can potentially contain a mutant from each strata in the perfect set, limiting gain in effectiveness. For operator addition, the optimal set is a superset of the original population, with as many new strata as there are new mutants (and there is no bound on the number of new strata). Since the random sample is constructed from the original population, it cannot ever contain the added strata.

A higher payoff might be obtained by **finding newer categories** than by removal of extant mutation operators.

Organization. Section II describes previous research in mutation reduction strategies. Section III discusses the theoretical framework for estimating the limits of mutation reduction strategies. Section IV discusses the sampling and operator selection strategies we study in detail. The results of experiments are detailed in Section V. A detailed discussion is provided in

Section VI. Threats to validity are explained in Section VII. We summarize our findings and conclusions in Section VIII.

II. Related Work

Mathur credits [31] the idea of mutation analysis to a term paper by Richard Lipton in 1970. The foundational assumptions and theory were first proposed by DeMillo et al. [32], and were first implemented by Budd et al. [33] in 1980. Mutation analysis relies on two fundamental assumptions — *the competent programmer hypothesis*, and *the coupling effect*. The competent programmer hypothesis suggests that programmers make simple mistakes. The coupling hypothesis suggests that test cases capable of detecting a simple fault in isolation will be able to detect it even when the fault appears in conjunction with other faults. Evidence of the coupling effect comes from both theoretical analysis [34], [35], [36], as well as empirical studies [37], [38], [39], [36]. For the competent programmer hypothesis, the size of a mean syntactic neighborhood for simple mistakes was quantified in our previous work [40].

According to Budd [12], mutation analysis is a stronger criteria than other coverage measures. The subsumption of multiple coverage measures by mutation analysis, including all the basic coverage measures [41] was shown by Offutt [42]. The subsumption of dataflow criteria was shown by Mathur [43]. Daran et al. [44] found that mutation analysis produces faults that are similar to actual faults in terms of the error traces produced. Andrews et al. [3], [4] found that ease of detection of mutants was similar to that of real faults when compared to manually generated faults (in that manually generated faults were harder to find). Recent research by Just et al. [5] using 357 real faults suggests that in 75% of the cases, mutation score and test case effectiveness improved together, which is a strong relationship compared to the same coupling for coverage (46%).

As mutation analysis requires a large number of potentially complete test executions, the cost of execution is often [45] considered to be a chief barrier to widespread adoption of the technique. Numerous approaches exist, that seek to reduce the cost of mutation analysis. Offutt and Untch [8] categorize these, in an orthogonal classification, as: do *fewer*, do *smarter*, and do *faster* approaches. Operator selection, mutant sampling, and mutant clustering fall under *do fewer* — approaches that seek to reduce the number of mutants evaluated. The *do smarter* approaches seek to reduce the time taken for the entire mutation analysis by intelligently managing the various phases. These include weak mutation, parallelization of mutation analysis, and space/time trade-offs. Similarly, *do faster* approaches seek to reduce the time taken for evaluation of a single mutant, and include mutant schema generation, code patching, and other methods.

The *do fewer* approaches, especially simple random sampling, debuted with the initial research in mutation analysis [12], [11], where it was noticed that even a 10% random sample of mutants can on average be almost as effective (99%) as the complete set of mutants.. Sampling was further investigated by Mathur [46], Wong et al. [47], [48], and Offutt et al. [9].

Determining relative merits of selective mutation strategies such as operator selection and random sampling has been an active field of research. Wong et al. [48] found similar effectiveness and efficiency (80%), when comparing operator selection (two selected operators) with x% sampling of operators. Mresa et al. [49] found that one can reduce the cost of mutation by directly targeting the cost of mutation operators. Excluding the operators with the highest cost still resulted in a set of mutants with good effectiveness. They found that operator selection works well compared to x% selective mutation when the targeted effectiveness is low. However, using only cost effective operators failed to generate sufficiently diverse mutants when targeted effectiveness is high.

Previous research by Zhang et al. [13] suggests that random sampling of mutants provides comparable effectiveness to that found for operator-based techniques. Counter-intuitively, on comparing strata sampling[8] with random sampling, they found that simple random sampling had a higher effectiveness for larger programs, while strata sampling was more effective for smaller programs. Zhang et al. [14] evaluated the effectiveness of sampling strategies on top of operator based selection provided by Javalanche. Their research suggests that strata sampling based on program elements performed best, with just 5% of mutants sufficient for high correlation (99%) with full mutation score, and that method level strata performed better than other strata such as statement or class. They suggest method level strata perform better against statement level strata due to the small number of mutants at statement level, and hence the difficulty in producing representativeness samples for each statement at smaller fractions.

Skew in fault representativeness among mutants was initially noticed by Budd et al. [12] who found that particular types of mutants are representative for particular kinds of faults. Constrained mutation was pioneered by Mathur [46], [50] and was further investigated by Wong et al. [51]. An extension of this approach called n-selection was suggested by Offutt et al. [9] where the most numerous mutation operators were removed one at a time. Taking into account the advances in mutation operator selection, a set of guidelines for operator selection was identified, and evaluated by Barbosa et al.[52]. Namin et al. [53], [54] formulated the concept of *sufficient mutation operators*, that reduce cost of mutation but maintain high statistical correlation with the full mutation score. This work showed that mutation reduction could be seen as a variable reduction problem where individual mutation operators were treated as independent variables, and principal variables that contributed the largest effect were found through statistical analysis. Their conclusion was that using just 28 out of 100 operators in Proteum was probably sufficient for an effective mutation analysis.

Untch [55] was the first to suggest statement deletion — a form of higher order mutation as an alternative to complete mutation analysis. Untch found a high correlation ($R^2 = 0.97$) between the statement deletion mutation score

[8]The two step random sampling is in effect strata sampling on operators, with equal priority.

and the traditional full mutation score; deletion only generated a smaller number of mutants than other operator selection methods. Deng et al. [56] extended the deletion operator for diverse language elements, and obtained an effectiveness of 92% while reducing the number of mutants by 80%.

The subsumption of individual mutants and mutation operators is an active area of research [57], [58], [59]. A mutant is subsumed by another when any test case that kills the later is guaranteed to kill the former. Extended to subsumption in operators, it means that particular operators could be completely avoided. Research from Kurtz et al. [60], [19] suggests that subsumption alone can lead to 96% reduction in mutants. We note that this result is based on an investigation of a single program, `cal`.

Higher order mutants (HOM) are another approach for improving the quality of mutants by combining simpler mutants into more complex mutants. Jia et al. [61], [62], found that the number of mutants can be reduced by 50% by making use of subsumption of simpler mutants by higher order mutants.

Mutation clustering[15], [20], [63] is another *do-fewer* approach where similar mutants are identified based on various properties, and a representative set is identified.

Our work is an extension of previous work on comparison of mutation reduction strategies [13], [14]. We note that the study by Zhang et al. [13] used 7 small (mean 313 LOC, maximum 513 LOC) C programs (5 programs if excluding different versions of the same program) that are called the Siemens test suite [64]. The test suites for these were created by researchers studying the impact of various techniques in fault detection, and hence may not be representative of real world test suites. Finally, they did not consider the impact of various mutation stratification techniques (suggested by Zhang et al. [14]) that can have a large impact. The later study by Zhang et al. [14], while using real world programs and test suites, does not actually investigate the relative merits of random sampling and operator selection. Rather, the study starts with a selected subset of operators (Javalanche only implements a selected subset of operators), on top of which other strategies are implemented. Hence their study does not actually evaluate the comparative benefits of operator selection and pure random sampling. Our study is the first exhaustive study for all well-known *do-fewer* techniques except mutation clustering. We consider a wider range of mutation approaches, and a larger set of large real-world projects, than any previous comparable study, which makes our work more generalizable and usable by practicing testers. Finally, Zhang et al. [29] investigated the scalability of selective mutation by considering how well a randomly sampled set of mutants represent the original population. They found that the number of mutants to be sampled for an adequate representation of mutants is dependent on the original number of non-equivalent mutants. Further, they find that a small number of randomly sampled mutants can be representative of even much larger set of mutants. Note that our work is not concerned with, and does not recommend a specific sample size for random sampling. Indeed, for software engineers who wishes to ascertain the sample size needed, Zhang et al. [29] may serve as a reasonable starting point.

In our previous work [28] we showed that there is an upper bound (13.1% on average) on the improvement in *mean effectiveness* that is possible using even an ideal mutation reduction strategy using post-hoc oracular knowledge of mutant kills. This paper extends that result by evaluating the actual improvement achieved by extant mutation reduction strategies, when they do not unrealistically have access to the mutant kills achieved. We examine multiple operator selection methods and strata sampling strategies, and our research suggests that even the modest advantage that theory suggests is possible is only rarely achieved. A critique of our previous research was that it failed to account for possible utility difference between different mutants. This present paper hence accounts for the possible differences in utility of different mutants by incorporating both test utility and assert utility.

III. Theoretical Analysis

Note that this section is a summary of the theoretical evaluation presented in our previous research [28]. Some of the detailed comments are elided for brevity. The aim of a mutation reduction strategy is to identify the *minimum* set of mutants that incorporates all[9] the faults in the original set. This may be done by identifying and collecting mutants into groups that represent particular faults. A single mutant from such a group is sufficient to represent all other mutants. Such a strategy depends on two characteristics of the mutant population if it is to be better than random sampling. The first of these characteristics is the amount of redundancy in each group. A *uniform redundancy* of faults is the best distribution for random sampling. For any other distribution (where the number of mutants in each faults is dissimilar), the effectiveness of random sampling is reduced. However, the mutant distribution is generated by the syntax of program being evaluated, and hence dependent on the particular program. As we seek to find the mean improvement for a *perfect strategy* for any arbitrary program, we use equal number of mutants per fault as a conservative distribution choice.

Our second emphasized characteristic is the number of minimum mutants necessary. Two mutants are *distinguishable* from each other in terms of the faults they represent if the tests that detect them are different. The set of distinguishable mutants is a set of mutants such that any pair of mutants are distinguishable. We note that, in the real world, the set of distinguishable mutants is often larger than the set of minimum mutants required to select a *minimum test suite*.[10] However, we note that this is due to the characteristics of the test suite, which is not connected to the mutant population. The difference exists only because the test suite is not diverse

[9] A general solution to this is not possible. Hence the practical aim of mutation reduction strategies is to aim for including as many different faults as practically possible in the orignial set, which is limited by the total number of unique faults.

[10] With respect to a mutant set, the smallest test suite that can kill all mutants in the set is called the *minimum test suite*. A *minimal test suite*, on the other hand, is a test suite such that removal of any test case from that test suite will cause mutation score to decrease.

We use a *greedy* algorithm to approximate the *minimum* test suite. The *greedy* algorithm has approximation bound of $k \cdot ln(n)$ (n number of elements, k the true minimum). Since the algorithm is robust, with a strong approximation bound, we assume the *minimal* set thus computed is approximately the *minimum* set.

enough. Hence, for theoretical purposes, we assume that any distinguishable mutants could be uniquely identified by tests. Further, inadequate test suites favor random sampling because a random sample can miss any such mutant that is actually distinguishable, but not in the minimal set (when comparing random sampling to the perfect strategy that selects minimal mutants).

We make the following simiplifications to make theoretical analysis tractable: that mutants are uniformly redundant with respect to faults, and that there exists a test case that can detect any given fault uniquely.

Next, we analyse the limits of reduction possible using this system using an ideal strategy with oracular knowledge of kills.

Impact of parameter deviation:

Skew: Any skew can reduce random sampling effectiveness, which can mean an increase in utility for the perfect strategy compared to random sampling.

Distinguishability: If a distinguishable mutant is skipped due to inadequate test suite, it can mean a decrease in utility for the perfect strategy compared to random sampling.

Analysis: The *utility* ($U_{strategy}$) of a strategy is defined as the improvement in effectiveness from using that strategy when compared to random sampling of the same efficiency[11] of mutants). That is,

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

This is the traditional evaluation of effectiveness [28], but extended for non-adequate test suites. The utility is essentially a measurement of how much advantage one gains by using this strategy over random sampling of the same efficiency in terms of test adequacy criteria. A *perfect* strategy can select the set of minimal mutants. We denote its utility by $U_{perfect}$[12]

Next, we compute the maximum $U_{perfect}$ for an idealized system, given uniform redundancy of faults, that is, equal number of redundant mutants for each distinguished mutant.

In sampling terminology, a *stratum* is a non overlapping subgroup that shares some characteristic. For our analysis we consider each set of non-distinguished mutants as separate strata. For a set of detected mutants, a reduction strategy should result in a mutant set where a test suite adequate for the reduced set should be adequate for the original set. That is:

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

Where $T_{perfect}$ is the set of test cases adequate for *perfect* strategy. The test suite quality thus chosen is dependent on the unique mutants in the sample. For x elements per non-distinguished stratum, and total $k \times x = n$ mutants (where k represents the number of independent strata), we have a sample size of $k \times p$ where p is the number of samples in each non-distinguished stratum. For perfect representation, $p = 1$ is sufficient, and ensures maximum improvement in effectiveness over random sampling (as shown below).

So, how many strata are expected in a random sample of size s?

Let X_i be a random variable such that:

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

Let X be the number of strata present in the sample. That is, $X = \sum_{i=1}^{k} X_i$, The expected value of X is then:

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

Now, the probability that the mutant 1 was selected is given by:

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

Expectation of X_i:

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

That is, the number of strata in a random sample is given by:

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

Because we know the sampling is perfect, the number of strata appearing in any sample is k, and the utility is computed as the ratio of difference to the baseline – random sample.

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

This converges to

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

and the maximum value is reached at $p = 1$.

$$U_{max} = \frac{1}{e - 1} \approx 58.2\% \tag{3}$$

The U_{max} thus computed is the *mean* advantage that an ideal strategy (with oracular knowledge) will have with a uniform distribution of mutants. While individual samples could still be arbitrarily advantageous, this is the *expected* improvement over random samples.

In other words, given an arbitrary program for which one has a robust set of test cases able to identify distinguishable mutants, and given a *perfect* strategy with oracular knowledge of mutant kills, one can expect it to have at least a mean effectiveness advantage of 58.2% over random sampling of the same number of mutants. If the mutant distribution is skewed, then the effectiveness of a strategy with oracular knowledge increases. On the other hand, if the test suite is not robust enough to identify distinguishable mutants uniquely, one may expect the effectiveness of the strategy with oracular knowledge to decrease. For real world strategies without prior knowledge of kills, the advantage held by the perfect strategy is hard (indeed, almost impossible) to achieve. Hence, we expect a much smaller utility for real world strategies.

[11] For our analysis, we only compare strategies holding efficiency constant. That is, $|M_{strategy}| = |M_{random}|$.

[12] We use the subscript p to stand for *perfect*, and *r* for *random*.

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 (extension)
DC *	Replace boolean conditions with false (folded into RC in mainline)

TABLE I: PIT Mutation Operators (We use abbreviations instead of operator names.). The starred (*) operators were added to account for inadequacies identified in original PIT operators.

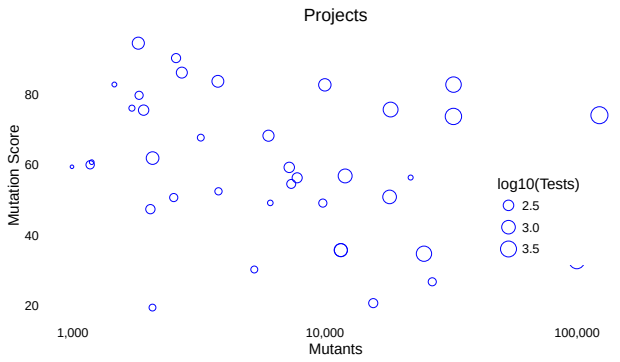

Fig. 1: The distribution of mutants and kills from test suites. The graph suggests a reasonably non-biased sample. We have both large, high mutation score programs, and also small projects with low mutation scores.

IV. Methodology

While our theoretical analysis is useful for evaluating the maximum utility a perfect strategy can produce under simplifying assumptions, it is necessary to verify the actual utility obtained using real world mutants. This is important because we do not know how close the real world distributions for mutants and test suites are to our simplifying assumptions. The first question we tackle is this: what is the maximum advantage one can expect to gain on real world systems? To find the maximum amount of advantage, one again considers the advantage of a hypothetical perfect strategy on a large set of open source programs and their test suites.

While selecting the sample programs, we had a few overriding concerns to ensure the generality and applicability of our findings [65]. First, we sought to ensure that our results were as applicable as possible to the practicing tester. That is, our results had to be applicable on as wide variety of systems as possible. Second, a statistically significant result is important to ensure that our results are not led astray by noise. To ensure the statistical validity of our results, we tried to reduce the number of uncontrolled variables present.

We started with $1,800$ Java projects from GitHub [66], and Apache project, which used the Maven [67] build system. In the case of Github, these were obtained through their search API, and in the case of Apache, we manually examined each project under the Apache umbrella to see whether it was a Java project using Maven as the build system. From this, we filtered out aggregate projects and projects without a test suite, leaving 796 projects. Not all of these compiled, with failure reasons ranging from unavailable dependencies and compilation errors due to syntax to bad configurations. This left us with 326 projects. However, not all of these projects could actually pass their own test suite. Effective mutation analysis requires a completely passing test suite, which required filtering these out as well. We eliminated any hung tests. We also eliminated any tests that did not detect any mutant, since they were redundant to our analysis. As a final step, we removed all projects with trivial test suites with tests as the cut-off. Any projects with unstable/flaky [68] test cases (that switched from fail to pass and back each time non deterministically) were removed as well. The final tally was 38 projects, given in Table II, where *Project* is the project name, $|M|$ is the size of the mutant set, M_{killed} is the size of the detected mutants, M_{uniq} is the number of distinguished mutants within detected mutants, $|T|$ is the size of test set, and T_{min} the size of the minimal mutant set.

Our mutation framework was PIT [69], which was extended to provide operators that it was lacking [70] (now accepted into mainline). The PIT operators are given in Table I. The details of each operator may be obtained from PIT documentation [71]. To mitigate random noise, we averaged results of each criterion over ten runs. Figure 1 provides the distribution of mutation scores and test suites.

For our experiment, we first evaluated an *oracular perfect strategy* against random sampling. Next, we compared the performance of various stratified sampling strategies and operator selection strategies against random sampling.

A. Oracular Strategy

Our task is to find the $U_{perfect}$ for each project. For a perfect strategy we only need complete representativeness,

$$kill(T_p, M) = kill(T, M)$$

and non-redundancy in selected mutants.

$$\forall_{m \in M_p} kill(T_p, M_p \setminus \{m\}) \subset kill(T_p, M_p)$$

The minimum mutant set [23] is representative and non-redundant. Hence it satisfies our requirements.

While finding the minimum test suite is NP-Complete[13] one can approximate it using the greedy algorithm due to Chvatal [73], given in Algorithm 1. In the worst case, if the number of mutants is n, and the smallest test suite that can cover it is k, this algorithm will achieve a $k \cdot ln(n)$ approximation. As we see in Figure 2, the algorithm is robust in practice, and finds results close to the actual minimum. So

[13]This is the *Set Covering Problem* [23] which is NP-Complete [72].

TABLE II: The projects, size of mutant set and test suites

Project	$\|M\|$	M_{killed}	M_{uniq}	$\|T\|$	$\|T_{min}\|$
annotation-cli	992	589	110	109	38.97
asterisk-java	15,530	3,206	451	214	196.32
beanutils	12,017	6,823	1,570	1,143	556.67
beanutils2	2,071	1,281	465	670	181.00
betwixt	7,213	4,271	1,198	305	206.35
clazz	5,242	1,583	151	140	64.00
cli	2,705	2,330	788	365	186.24
cli2	3,759	3,145	1,066	494	303.86
collections	24,681	8,561	2,091	2,241	938.32
commons-codec	9,983	8,252	1,393	605	444.69
commons-lang3	32,323	26,741	4,479	2,456	1,998.11
commons-math1-l10n	6,067	2,980	219	119	109.02
commons-math1	122,484	90,681	17,424	5,881	4,009.98
config-magic	1,188	721	204	112	74.55
configuration	18,198	13,766	4,522	1,772	1,058.36
csv	1,831	1,459	411	173	117.97
dbutils	2,030	961	207	224	141.53
events	1,171	702	59	180	33.87
faunus	9,801	4,809	553	173	146.11
fongo	1,461	1,209	175	113	70.73
hank	26,622	7,109	546	171	162.88
java-api-wrapper	1,715	1,304	308	125	107.04
java-classmate	2,566	2,316	551	215	196.57
jdbi	7,754	4,362	903	277	175.57
jfreechart	99,657	32,456	4,686	2,167	1,696.86
joda-money	2,512	1,272	236	173	128.48
jodatime	32,293	23,796	6,920	3,973	2,333.49
jopt-simple	1,818	1,718	589	538	158.37
mercurial-plugin	2,069	401	102	138	61.77
mirror	1,908	1,440	532	301	201.21
mp3agic	7,344	4,003	730	206	146.79
ognl	21,852	12,308	2,990	114	85.43
pipes	3,216	2,176	338	138	120.00
primitives	11,553	4,125	1,365	803	486.71
sandbox-primitives	11,553	4,125	1,365	803	488.56
validator	5,967	4,070	759	383	264.35
webbit	3,780	1,981	325	147	116.93
xstream	18,030	9,163	1,960	1,010	488.25

long as $NP \neq P$[14] this is the best approximation one can have for the actual minimum [75]. To ensure that our algorithm returned the correct results, we verified that the *minimum frequency* of kills of the set of mutants by the minimal test suite was 1 (A larger *minimum frequency* indicates a redundancy in that many tests – a rare but well-known problem with the *greedy* algorithm).

We also average the estimated minimal[15], test suite size over 100 runs (see Figure 2, which provides the variability of the runs ordered by the size of minimal test suite). The variability is indeed very little, and decreases as the size of test suite increases.

We next randomly sampled $|M^{min}_{perfect}|$ mutants from

[14] Ammann et al. [23] provides another algorithm that we call *reverse greedy algorithm*. It has two deficiencies. Say k is the actual minimum. the approximation ratio of the *greedy* algorithm is at most $k \cdot ln(n)$. However, that of *reverse greedy* would be much larger [74] (if an approximation ratio exists). The *reverse greedy* also requires a much larger number of steps to complete than the *greedy* when the size of minimal set is very small compared to the full set.

[15] We use the minimal approximation to minimum from here on. Hence, we do not distinguish between minimal and minimum further.

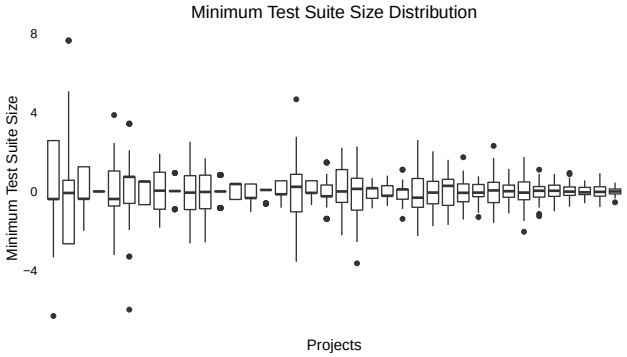

Fig. 2: Variation of minimal test suite size ordered by mean minimal test suite size. The figure suggests that there is very little variation, and the variation decreases with the increase in the number of test cases.

M. The minimal test suite T^{min}_{random} was calculated, and was applied to M to find the mutants that are killed: $kill(T^{min}_{random}, M)$. This result is used to compute the utility of perfect strategy with respect to that particular random sample. We repeated the experiments 100 times for each project. The results were averaged to compute $U_{perfect}$ for each project.

B. Sampling Strategies

We used several sampling criteria suggested in the literature. For each sampling criterion 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) Stratified random sampling over mutation operators: First suggested by Wong et al. [48], 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.

2) Stratified random sampling over program elements: Following the suggestion of Zhang et al. [14], 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. [14] for sampling fractional values.

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

C. Operator Selection

For selective methods, we evaluated the mutation operators suggested by Wong et al. [48], Offutt et al. [76], [56], and Namin et al. [54]. Since Javalanche [77] 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 (see below).

1) Constrained Mutation: Wong et al. [48] 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. [76]. 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 [76]. 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.

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 [77], [10] adapts for Java bytecode the E-Selective operator set suggested by Offutt et al. [76] 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 [77] 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 [10].[16] These map directly to PIT operators IC, M, NC, and VMC.

4) Variable Reduction: This method was proposed by Namin et al. [54], who 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 the *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

$$\mu M_s = 0 + 0.55nmc + 0.2rc + 0.1dc + 0.2rv + 0.1cc + 0.7emv + 0.02m + 0.02ri$$

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. [9] 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. [56]. 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 would 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 minimum 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[17].

D. Evaluation of Reduction Strategies

For the purpose of comparing different mutation reduction strategies, we use three different measures. The traditional *strategy effectiveness* which compares the effectiveness of selected mutants in representing the full set of mutants, the *test*

[16] 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.

[17] 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.

utility (also called *minimum mutant set utility*) which compares the size of minimum test suite (or the size of minimum set of mutants — which is the same), and the *assert utility* which incorporates the effectiveness of the selected test cases by using the number of asserts in each. Note that there is a difference between *minimal* test suite and *minimum* test suite. A *minimal* test suite is test suite such that removal of any test case from that test suite will cause the mutation score to decrease. However, there can be other test suites that have a smaller number of test cases. A *minimum* test suite is the smallest *minimal* test suite.

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 minimum, non-redundant test suite that detects all of these mutants. The observation is repeated multiple times to account for any noise.

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:

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

Positive values show that the strategy requires *more* tests than the 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 selection). Values close to zero mean 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 correlation with fault detection and mutation kill rate [26], [24], 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 *minimum* test suite (using Algorithm 1) that kill 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:

$$E_r = \frac{|kill(T^{min}_{random}, M)|}{|kill(T, M)|}$$

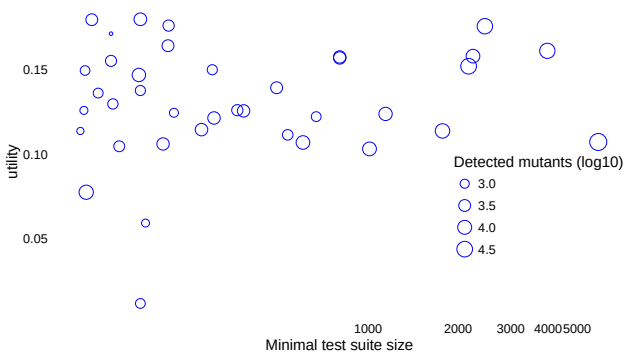

Fig. 3: The figure plots utility (y-axis) against the average minimal test suite size (log10). Bubble size represents the magnitude of detected mutants (log10). The figure suggests that there is no correlation between utility and average minimal test suite size.

The traditional mutation reduction criterion *strategy effectiveness*[18] (E_s), is computed by collecting the minimum set of 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:

$$E_s = \frac{|kill(T^{min}_{strategy}, M)|}{|kill(T, M)|} - E_r$$

The utility of the strategy is computed as:

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

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 result for all 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 multiple times to avoid random noise in the results.

[18] also called operator mutation score by Mresa et al. [49]

TABLE III: The maximum utility achievable by a perfect strategy for each project

$Project$	$\|kill(T,M)\|$	$\|kill(T_r^m,M)\|$	U_{perf}
annotation-cli	589	529.51	0.11
asterisk-java	3,206	2,754.69	0.16
beanutils	6,823	6,071.53	0.12
beanutils2	1,281	1,141.73	0.12
betwixt	4,271	3,809.19	0.12
clazz	1,583	1,402.39	0.13
cli	2,330	2,069.84	0.13
cli2	3,145	2,760.66	0.14
collections	8,561	7,392.63	0.16
commons-codec	8,252	7,455.50	0.11
commons-lang3	26,741	22,742.46	0.18
commons-math1-l10n	2,980	2,527.66	0.18
commons-math1	90,681	81,898.25	0.11
config-magic	721	640.91	0.13
configuration	13,766	12,359.89	0.11
csv	1,459	1,282.93	0.14
dbutils	961	854.83	0.12
events	702	662.97	0.06
faunus	4,809	4,078.22	0.18
fongo	1,209	1,052.99	0.15
hank	7,109	6,200.08	0.15
java-api-wrapper	1,304	1,148.52	0.14
java-classmate	2,316	1,969.76	0.18
jdbi	4,362	3,914.73	0.11
jfreechart	32,456	28,171.19	0.15
joda-money	1,272	1,257.55	0.01
jodatime	23,796	20,491.96	0.16
jopt-simple	1,718	1,546.21	0.11
mercurial-plugin	401	342.91	0.17
mirror	1,440	1,252.50	0.15
mp3agic	4,003	3,620.41	0.11
ognl	12,308	11,426.09	0.08
pipes	2,176	1,884.73	0.16
primitives	4,125	3,565.83	0.16
sandbox-primitives	4,125	3,563.85	0.16
validator	4,070	3,616.71	0.13
webbit	1,981	1,793.96	0.10
xstream	9,163	8,307.12	0.10

TABLE IV: The maximum utility achievable by a perfect strategy for each project using distinguishable mutants $\hat{M}$

$Project$	$\|kill(T,\hat{M})\|$	$\|kill(T_r^m,\hat{M})\|$	U_{perf}
annotation-cli	110	93.68	0.18
asterisk-java	451	372.25	0.21
beanutils	1,570	1,341.04	0.17
beanutils2	465	392.30	0.19
betwixt	1,198	1,055.30	0.14
clazz	151	129.24	0.17
cli	788	688.05	0.15
cli2	1,066	903.30	0.18
collections	2,091	1,750.05	0.19
commons-codec	1,393	1,192.29	0.17
commons-lang3	4,479	3,663.98	0.22
commons-math1-l10n	219	177.86	0.23
commons-math1	17,424	15,139.90	0.15
config-magic	204	171.60	0.19
configuration	4,522	3,934.21	0.15
csv	411	349.30	0.18
dbutils	207	170.60	0.21
events	59	49.15	0.20
faunus	553	467.03	0.18
fongo	175	145.13	0.21
hank	546	465.52	0.17
java-api-wrapper	308	259.87	0.19
java-classmate	551	450.46	0.22
jdbi	903	783.99	0.15
jfreechart	4,686	3,910.15	0.20
joda-money	236	230.76	0.02
jodatime	6,920	5,801.10	0.19
jopt-simple	589	514.36	0.15
mercurial-plugin	102	80.95	0.26
mirror	532	444.17	0.20
mp3agic	730	639.01	0.14
ognl	2,990	2,835.77	0.05
pipes	338	288.41	0.17
primitives	1,365	1,155.09	0.18
sandbox-primitives	1,365	1,155.01	0.18
validator	759	647.36	0.17
webbit	325	280.89	0.16
xstream	1,960	1,691.84	0.16

The utility U_{perf} is computed as $U_{perf} = 1 - \frac{kill(T,M)}{kill(T_r^m,M)}$ where $kill(T,M)$ is the number of detected mutants in M, and $kill(T_r^m,M)$ is the number of mutants detected by a minimal test corresponding to a random sample of mutants of the same size as the minimal set of mutants.

A. Comparison of Oracular Strategy to Random Sample

1) All Mutants: Our results for the comparison of oracular strategy with random sample are given in Table III, where *Project* is the project name, $|kill(T,M)|$ the number of mutants detected in M by the test suite T, $kill(T_r^m,M)$ the number of mutants detected in M by the test suite T_{random}^{min}, and U_{perf} is the utility of the perfect strategy. The largest utility achieved by the perfect strategy was 18%, while the lowest utility was 1.15%. The mean utility of the perfect strategy was 13.1%. One sample *u-test* shows that 95% of projects have maximum utility between 12.2% and 14.3% with $p < 0.001$. Figure 5 shows distribution of utility for each project. The projects are sorted by their average minimal test suite size.

Does the situation improve with larger test suite size or project size? Not really. The Figure 3 plots utility U_p against the average minimal test suite size (log). The figure shows that there is little correlation between the two, and test suite size is not a factor in improving utility. Similarly, the Figure 4 plots utility U_p against the number of detected mutants. Indeed, none of the factors including minimal test suite size, total mutants, killed mutants, and mutation score show even mod-

TABLE V: The correlation of utility for all mutants, killed mutants, mutation score, and minimal 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	-0.02
M_{kill}/M	-0.02	-0.01
T^{min}	-0.02	-0.02

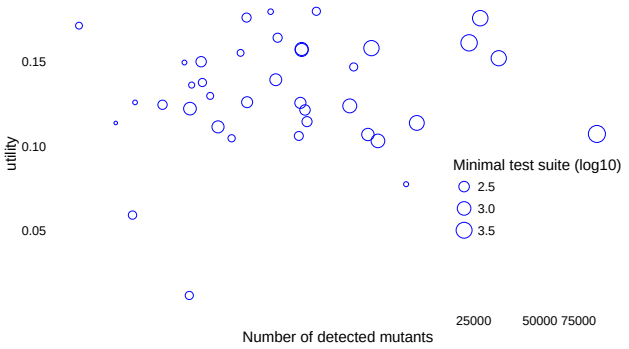

Fig. 4: The figure plots utility (y-axis) against the number of detected mutants. Bubble size represents the magnitude of average minimal test suite size (log10). The figure suggests that there is no correlation between utility and number of detected mutants.

erate correlation with utility U_p. The correlation factors are given in Table V. The low correlation suggests that population characteristics such as mutation score, or size of minimal test suite does not have an impact on our results.

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

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

The variable *project* has a correlation of 0.689 with $U_{perfect}$, and the combined terms have a correlation with $U_{perfect}$ of 0.9995.

2) Distinguishable Mutants: Results are given in Table IV, where *Project* is the project name, $|kill(T, \hat{M})|$ the number of mutants detected in M_{uniq} by the test suite T, $kill(T_r^m, \hat{M})$ the number of mutants detected in M_{uniq} by the test suite T_{random}^{min}, and U_{perf} is the utility of the perfect strategy. The largest utility achieved by the perfect strategy was 26.2%, while the lowest utility was 2.28%.

The mean utility of the perfect strategy was 17.5%. One sample *u-test* showed that 95% of projects have a maximum utility between 16.8% and 18.8% ($p < 0.001$).

Figure 6 shows utility distribution for each project, again sorted by average minimal test suite size. This situation does not change with either test suite or project size.

Utility U_p has low correlation with total mutants, detected mutants, mutation score, and minimal test suite size. Correlation factors are given in Table V.

Analysis of variance (ANOVA) on $U_{perfect}$ found that the variability due to project is again significant at $p < 0.001$ and strongly interacts with $kill(T_{random}, M)$.

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

The variable *project* has a correlation of 0.742 with the $U_{perfect}$, and the combined terms have a correlation with $U_{perfect}$ of 0.9994.

B. Comparison of Selection Strategies

1) Operator Selection: Considering the operator selection results (Table VI, standard deviation in Table XI. the strategy with the maximum advantage in *utility* was *Constrained* (0.18% compared to random sampling). The strategy with the maximum advantage in *test utility* was *S-Selective* (3.02% compared to random sampling). Similarly, the strategy with the maximum advantage in *assert utility* was again *S-Selective* (1.44% compared to random sampling).

Considering the N-selection results (Table VII, standard deviation in Table XII. In terms of utility, the best strategy was *RS* (4.75% compared to random sampling). The strategy with the maximum advantage in *test utility* was removal of *NMC* (2.37% compared to random sampling). Similarly, the strategy with the maximum advantage in *assert utility* was *NMC* (2.02% compared to random sampling).

2) Stratified sampling over operators: Considering stratified sampling over operators (Table VIII, standard deviation in Table XIII. In terms of utility, the best strategy was *1/64* (0.69% compared to random sampling). The strategy with the maximum advantage in *test utility* was *1/32* (0.78% compared to random sampling). Similarly, the strategy with the maximum advantage in *assert utility* was *1/64* (2.06% compared to random sampling).

3) Stratified sampling over program elements: Considering stratified sampling over program elements (Table IX, standard deviation in Table XIV. in terms of utility, the best strategy was *1/64 sampling of class* (2.81% compared to random sampling). The strategy with the maximum advantage in *test utility* was *1/16 sampling of method* (6.68% compared to random sampling). Similarly, the strategy with the maximum advantage in *assert utility* was *1/32 sampling of class* (7.82% compared to random sampling).

VI. Discussion

One of the biggest questions for a practicing software tester is whether the test suite is good enough. While there exist numerous techniques to evaluate test suites, mutation analysis is often considered to be the golden standard. Unfortunately, mutation analysis is hobbled by the amount of computation required for a full run. A reduction in the computational requirements of mutation analysis, while maintaining its effectiveness, is actively sought after.

In this context, it is crucial to understand the limits of such reduction strategies, especially the comparative performance of each strategy against simple random sampling which serves as a baseline. This can help us evaluate benefits of further research.

A. Comparison With Oracular Strategy

Theoretical analysis of a simple idealized system finds a *mean effectiveness* improvement of 58.2% over random sampling for a perfect mutation reduction strategy with oracular knowledge of mutation kills, assuming a uniform redundancy of mutants, and robust test cases able to disignuish unique faults.

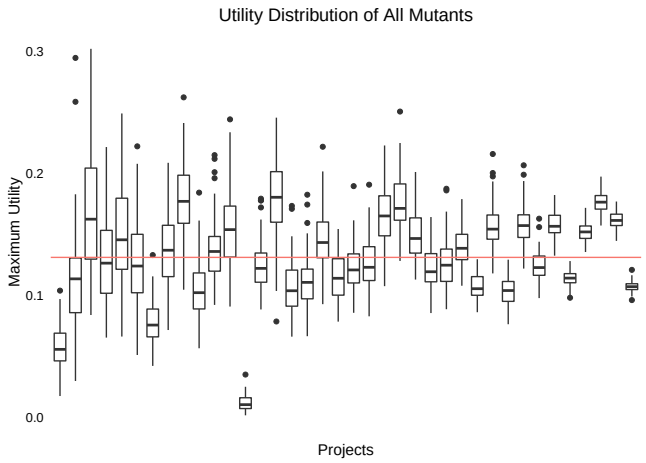

Fig. 5: Using all mutants.

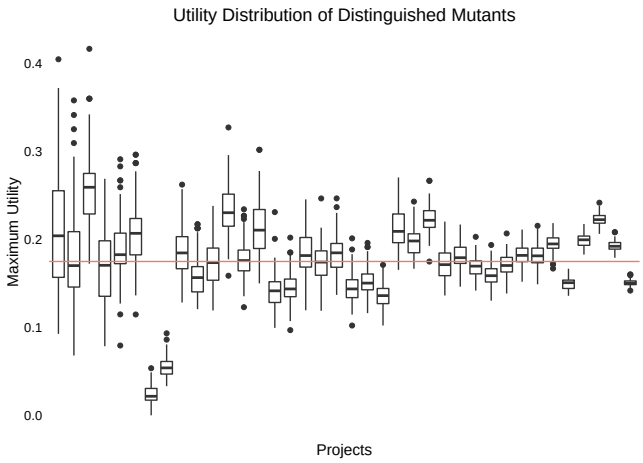

Fig. 6: Using distinguished mutants.

Distribution of maximum utility using distinguished mutants across projects. The projects are ordered by the cardinality of mean minimal test suite. The red line indicates the mean of all observations.

The mean operator selection results for all projects.

Strategy	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
Constrained	**0.18**	-12.94	-9.24	14.13	0.05	98.32
E-Selective	-1.05	-7.76	-7.60	36.42	-1.04	99.64
S-Selective	0.08	**3.02**	**1.44**	49.35	**0.08**	99.86
R-Selective	-1.39	-9.89	-9.03	14.22	-1.35	98.74
ES-Selective	0.01	0.63	0.38	85.78	0.01	99.99
RS-Selective	-0.57	-6.10	-5.77	50.65	-0.56	99.82
RE-Selective	0.00	1.87	1.37	63.58	0.00	99.94
Javalance	-0.10	-5.49	-3.88	59.73	-0.10	99.91
VarReduction	0.03	1.73	1.29	71.09	0.03	99.97
SDL	-0.01	1.58	0.30	63.02	-0.01	99.94

TABLE VI: The operator selection strategy

Removed	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
rm.nmc	0.04	**2.37**	**2.02**	73.00	0.04	99.96
rm.rv	0.04	-1.46	-2.21	63.09	0.04	99.92
rm.ic	0.02	-1.38	-2.29	53.45	0.02	99.87
rm.dc	0.08	-2.77	-3.43	44.45	0.08	99.81
rm.nc	0.05	0.22	-2.00	32.02	0.05	99.64
rm.rc	-0.28	-1.99	-7.14	20.92	-0.27	99.31
rm.vmc	-0.68	-5.06	-11.62	17.32	-0.67	99.14
rm.cc	-1.03	-18.25	-18.17	11.70	-1.19	97.55
rm.emv	-5.94	-29.54	-28.13	7.03	-6.02	94.48
rm.m	-7.59	-29.93	-26.79	4.56	-6.39	92.66
rm.cb	-12.89	-36.36	-36.96	2.89	-9.72	88.49
rm.i	-21.00	-40.58	-40.81	1.73	-15.18	82.11
rm.ri	-33.38	-39.81	-31.14	0.59	-13.29	47.87
rm.rs	**4.75**	-12.11	-25.68	0.15	**0.86**	33.69
rm.es	-11.08	-14.25	-16.28	0.04	-1.81	10.37

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

Fraction	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2	0.01	0.23	0.26	50.00	0.01	99.87
1/4	0.04	-0.33	-0.24	25.00	0.03	99.50
1/8	0.11	0.60	0.83	12.51	0.10	98.63
1/16	0.23	-0.35	-0.59	6.25	0.20	96.57
1/32	-0.02	**0.78**	1.89	3.13	-0.05	92.83
1/64	**0.69**	0.74	**2.06**	1.56	**0.42**	86.08

TABLE VIII: The operator-based x% sample strategy

Empirical analysis of a large number of open source projects reveals that the practical limit is much lower than this theoretical advantage: it is on average only 13.1% for mutants produced by PIT. Discounting the effects of skew (by using only distinguished mutants) the potential improvement is still just 17.5% on average.

The theoretical limit in analysis 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. How different is the distribution of faults

Fraction Elt	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2 line	0.05	2.24	1.74	49.98	0.05	99.87
1/4 line	0.12	3.11	2.21	24.96	0.12	99.49
1/8 line	0.35	3.92	3.16	12.47	0.34	98.62
1/16 line	0.67	3.03	2.14	6.24	0.59	96.57
1/32 line	0.54	1.50	0.64	3.15	0.40	92.81
1/64 line	0.39	0.05	1.20	1.57	0.32	86.04
1/2 method	0.04	2.16	1.73	50.00	0.04	99.87
1/4 method	0.11	3.25	2.34	25.00	0.11	99.51
1/8 method	0.39	4.48	3.62	12.51	0.38	98.68
1/16 method	0.98	**6.68**	7.15	6.24	0.91	96.69
1/32 method	1.94	5.30	5.92	3.13	1.66	92.69
1/64 method	1.91	6.37	6.24	1.57	1.12	86.35
1/2 class	0.01	0.90	0.80	49.99	0.01	99.88
1/4 class	0.05	0.84	0.31	25.01	0.05	99.53
1/8 class	0.21	2.45	1.91	12.50	0.20	98.68
1/16 class	0.59	3.29	3.39	6.24	0.55	96.85
1/32 class	1.79	5.17	**7.82**	3.12	1.42	92.77
1/64 class	**2.81**	4.12	5.00	1.55	**2.08**	86.09

TABLE IX: 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.

in mutants in real world compared to our simple model? and how far is our assumption of test cases that distinguishes faults uniquely? The *TVD* column in Table X captures the *total variation distance*[19] between a uniform distribution of faults and the actual distribution. The *Mean* column is computed in the following manner: We computed the total number of mutants detected by each test in a given project, and divided that by the total number of mutants. We then computed the *mean* and *variance* of this set of values for each project. That is, a low value for *Mean* and *Var* provides an indication of whether that test case was able to detect a mutant uniquely. Note that violation of these two assumptions have opposite effects on the *mean effectiveness*. So what do these values mean? Indeed, the real world distribution of faults in mutants seem to be far from the simplified model we considered in the theoretical analysis. The value of the theoretical model is in showing that there exist a limit, even for such simple systems, and the possible improvement in reduction is not unlimited as is often supposed. Further, as we have seen in the empirical analysis, the effect of sharing in the mutant kills between test cases is larger than the effect due skew in redundant mutants leading to a much lower upper bound in *mean effectiveness*.

Finally, we found that effects of skew were small (the difference between mean effectiveness improvement of the set of distinguished mutants and the mean effectiveness improvement of all mutants is only 4.39%).

The empirical upper bounds on gain in utility are quite low, and call into question the effort invested into devising, evaluating, and improving mutation reduction strategies. Of course, random sampling is subject to the vagaries of chance: one can obtain 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-math* is just 0.397%. The chance for a really bad sample is very low in the case of any project large enough to require mutant reduction, and drops quickly as the number of test cases increases. It is possible 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 ($U_{perfect}$ =14%). In general, using accepted standard practices for statistical sampling to produce reasonably-sized random mutant samples should be effective in practice for avoiding unusually bad results. Random sampling is also easy to implement and incurs negligible overhead.

[19] The *total variation distance* is the largest difference in probabilities that can be assigned to an event when it is considered under the different probability distributions under consideration [78].

B. Comparison with Selection Strategies

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 *minimum 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 *minimum test suite*. Hence, the *average size* of a minimum 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. Consider the results from our empirical evaluation:

TABLE X: Comparison of actual distribution to the hypothetical distribution of mutants

$Project$	TVD	$Mean$	Var
annotation-cli	0.610	132.519	619.194
asterisk-java	0.824	54.634	2,233.218
beanutils	0.528	114.084	11,379.741
beanutils2	0.797	22.927	989.231
betwixt	0.370	375.375	85,282.405
clazz	0.761	45.633	1,831.671
cli	0.649	103.560	4,305.466
cli2	0.578	77.097	6,895.613
collections	0.828	23.263	608.567
commons-codec	0.694	66.869	8,155.081
commons-lang3	0.658	38.425	2,385.877
commons-math1-l10n	0.882	167.735	45,788.994
commons-math1	0.874	46.839	9,733.084
config-magic	0.556	77.784	1,279.737
configuration	0.596	311.203	190,370.838
csv	0.417	66.973	3,135.170
dbutils	0.832	15.247	277.450
events	0.785	34.732	1,638.498
faunus	0.777	162.535	17,236.249
fongo	0.665	87.982	1,152.285
hank	0.851	102.035	10,957.940
java-api-wrapper	0.700	59.073	4,382.035
java-classmate	0.540	56.341	6,825.645
jdbi	0.512	369.899	70,890.693
jfreechart	0.890	122.665	39,898.268
joda-money	0.949	15.907	2,745.724
jodatime	0.787	139.046	17,765.564
jopt-simple	0.576	74.075	6,906.461
mercurial-plugin	0.911	10.688	60.632
mirror	0.696	35.847	1,182.496
mp3agic	0.545	142.746	58,857.711
ognl	0.516	523.544	161,933.423
pipes	0.770	45.934	1,372.295
primitives	0.676	21.181	347.946
sandbox-primitives	0.674	21.253	348.259
validator	0.555	107.037	13,150.119
webbit	0.694	143.712	27,401.945
xstream	0.621	362.312	166,391.685

The TVD is the *total variation distance* of the actual distribution of mutant kills from a *Uniform* distribution. The $Mean$ and Var are respectively the *mean* and *variance* of the mean number of mutants killed per test case.

1) Operator Selection: For Operator selection, (Table VI), *Constrained* performs best in utility, while *S-Selective* performs best in test utility and assert utility. For N-selection, (Table VII) the best test utility and assert utility was removal of operators until NMC. The best strategy effectiveness for all projects was the removal of operators until RS. Note that the advantage gained for most strategies compared to random sampling is very small, and are often negative.

2) Stratified sampling over operators: For stratified sampling over operators (Table VIII), the best test utility appears to be at $1/32$ and for assert utility, and strategy effectiveness, the best is $1/64$. Note that the advantage gained in each case is very small.

3) Stratified sampling over program elements: For stratified sampling over program elements (Table IX), there appears to be a small but consistent advantage for most sample fractions. The best test utility was achieved by $1/16$ method-based sampling. Similarly, the best assert utility was achieved by $1/32$ class based sampling, and the best strategy effectiveness was for $1/64$ class based sampling.

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. Second, operator selection strategies provide little benefit (or even decrease performance) over random selection for even strategy effectiveness (the traditional criteria).

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.

The strata-based random selection strategies fare a little better. While they are mostly advantageous, the advantage is always rather small — below 5%. 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 [79], [14] when this assumption is met. Our results indicate that while there is a small advantage, this advantage is almost always less than 1% compared to the baseline for strategy effectiveness. One factor that one may wish to consider is that strata based techniques can reduce the variance of computed results compared to random sampling, and hence may be of use. However, strata-based sampling 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. Hence simple strata based sampling may be advantageous to consider when the considered strata are large enough to provide representative samples. However, in this respect, one may consider using one of the *sub-random* sampling systems such as Poisson disc sampling [80] which avoids the regularity of systematic sampling, but also reduces variability of results.

We caution that cost-reduction is not the only reason for using selective mutation. Operator selection techniques [81] have been used to reduce the incidence of equivalent mutants, and some of the higher order mutation operators such as statement deletion [56], [82] have markedly fewer equivalent mutants than other operators. Further, one of the stated aims of Javalanche [83] is to avoid equivalent mutants as much as possible. However, one has to be very careful about the impact in effectiveness if one uses operator selection for these purposes.

Our results are applicable not only to selective mutation, but also to mutation implementors looking to add new mutation operators. Imagine a mutation system implementor has achieved perfect set of mutation operators: their current set of mutants does not have any redundant mutants (this is highly unlikely given our understanding of mutant semiotics, and the

complexity of real programs). When we consider the addition of a new set of random mutants that *do not* improve the mutant set, in that they are all redundant with respect to the original set (probably unlikely in practice, given that we are introducing new mutants), the maximum disadvantage thus caused is bounded by our limit (18.8% upper limit for 95% of projects). However, at least a few of the new mutants are in reality likely to improve the representativeness of a mutation set compared to possible faults. Since there is no upper bound on the number of new distinguishable mutants that could be introduced, there is no upper bound for the maximum advantage gained by adding new mutation operators. A bounded disadvantage and unbounded advantage is clearly a desirable situation. Adding new operators is especially attractive in the light of recent results showing classes of real faults that are not effectively captured by any of the mutation operators in common use [5].

VII. Threats to Validity

Our theoretical study may be subject to the following threats to validity. We showed that there exist a limit to the mean effectiveness under two simplifying assumptions: uniform redundancy of faults in mutants, and sufficient test cases to uniquely identify faults. However, it is possible that the real world distribution of faults may be much more complex, and our conclusions from this simple model may not be applicable to the real world faults and test cases.

Our empirical study may be subject to the following threats to validity.

Construct validity: We use the minimum set of mutants as the measure of diversity of mutants. It is possible that the minimum set of mutants is not representative of the actual diversity of mutants. However, we note that the minimum set of mutants is the best method suggested in the literature to measure actual diversity of mutants. We have further used the number of asserts as a secondary measure to further protect against unforeseen biases.

Internal validity: Our measurement of the minimal set of mutants is only an approximation. While the algorithm used guarantees an $H(|M|)$ approximation, it is not clear how much actual variation this could have caused in our measurement. We note that we take an average of 100 runs for each observation to protect against such errors.

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 ensured a fair repertoire of mutation operators in PIT, 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.

External validity: Our results depend on the representativeness of our samples which were obtained from the Github repository. We have used Java `Maven` projects for ease of automation of experiment and measurement. While we do not foresee any confounding biases in our selection procedure, the possibility exists. Hence, the generalizability of our findings depends on the representativeness of these projects.

Finally, software bugs are a part of life. While we have tried to ensure that our tools, and analysis are free of errors, the same can not be guaranteed. Hence, replication of these results by a different group using different tools is of utmost importance.

VIII. Conclusion

This paper shows that blind random sampling of mutants is surprisingly close in effectiveness to the best achievable bound for mutation reduction strategies that unrealistically use perfect knowledge of mutation analysis results. There is surprisingly little room for improvement over random sampling to be gained by smarter reduction strategies. Previous researchers showed that there is very little advantage to *current* operator selection strategies compared to random sampling [13], [14]. However, these experiments lacked direct comparison with random sampling of the *same* number of mutants. It was also demonstrated that the *current* strategies fare poorly [23] when compared to the actual minimum mutant set, again without a comparison to random sampling. Our primary contribution in this paper is to combine an analysis of the absolute limits to the improvement over random sampling that any reduction strategy can have (*irrespective* of the intelligence of the strategy) with a thorough, direct, empirical comparison of the effectiveness of most current strategies' effectiveness compared to random sampling.

The theoretical analysis suggests a mean effectiveness advantage of 58.2% over random sampling for a perfect mutation reduction strategy with (unrealistic) oracular knowledge of kills for an arbitrary program, under the assumption of uniform redundancy in mutants, and test sets robust enough to distinguish unique faults. Empirically, we find that actual projects yield a much lower advantage 13.1% even for a perfect reduction strategy with oracular knowledge. Eliminating the effects of skew in redundant mutant populations by considering only distinguished mutants, we find that the advantage of a perfect mutation reduction strategy is still only 17.5% over random sampling. The low impact of skew on results (4.39%) suggests that our simplifying assumptions for theoretical analysis are reasonable. No actual reduction strategies examined come close to the empirical limit, with the best performing no better than 5%, and most others performing worse (sometimes much worse) than random sampling. The disparity between the theoretical prediction and the empirical results is due to the inadequacies of real world test suites, which have a much smaller minimum mutant set than the distinguishable mutant set. While work on mutation reduction strategies routinely claims a high reduction factor, and one might expect a similar magnitude of utility over random sampling, this to materialize either in theory or practice, for a large set of real world open source programs and suites.

A researcher or an implementor of mutation testing tools should carefully consider the value of devising or imple-

menting a mutation reduction strategy, given this theoretical and empirical performance for both hypothetical and real methods. Because 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 they need to test (such consideration could be based on historical data for the project, or on projects that are in some established sense similar). Random sampling of mutants is not extremely far from an empirical upper bound on an ideal mutation reduction strategy, and has the considerable advantage of having little room for an unanticipated bias due to adapting a selection method that turns out to be ill-advised for a particular program.

The most important takeaway from our research may be that it is perhaps most effective to improve mutation analysis via further research into newer mutation operators (or new categories of mutation operators such as domain specific operators for concurrency or resource allocation). We show that there is limited or no reduction in utility due to addition of newer operators even in the worst case, while there is no upper bound for the possible improvement.

Our advice to mutation tool implementors is twofold: try to provide as many sources of variation as possible, and avoid questionable reduction strategies that reduce overall variation. You can always reduce the number of mutants to execute using simple 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 strategies such as operator selection, or clustering unless there are other requirements such as avoidance of equivalent mutants, reduction of mutation cost, or selection of specific bug types. (Note that even subsumption of specific operators is not a forgone conclusion [30]). Use strata-based sampling only when you can be sure that all strata elements can produce representative samples for a given sampling fraction.

Indeed, the most important insight to be derived from our research can be succinctly described by what we call *Hamlet's principle*, formulated in the context of random testing [84]: *in the absence of a rational basis for systematic methods, random methods are best at avoiding bias*.

References

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

[2] P. Ammann and A. 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] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] ——, "A quality estimation of mutation clustering in c# programs," in *New Results in Dependability and Computer Systems*. Springer, 2013, pp. 119–129.

[17] Y.-S. Ma and S.-W. Kim, "Mutation testing cost reduction by clustering overlapped mutants," *Journal of Systems and Software*, vol. 115, pp. 18–30, 2016.

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

[19] B. Kurtz, P. Ammann, and A. J. Offutt, "Static analysis of mutant subsumption," in *International Conference on Software Testing, Verification and Validation Workshops*, April 2015, pp. 1–10.

[20] J. Strug and B. Strug, "Machine learning approach in mutation testing," in *Testing Software and Systems*. Springer, 2012, pp. 200–214.

[21] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Do redundant mutants affect the effectiveness and efficiency of mutation analysis?" in *International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 720–725.

[22] 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.

[23] P. Ammann, M. E. Delamaro, and A. 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.

[24] 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.

[25] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *IEEE 4th ICST*, ser. International Conference on Software Testing, Verification and Validation. Washington, DC, USA: IEEE Computer Society, 2011, pp. 90–99.

[26] Y. Zhang and A. Mesbah, "Assertions are strongly correlated with test suite effectiveness," in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ACM, 2015.

[27] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[28] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "On the limits of mutation reduction strategies," in *International Conference on Software Engineering*. IEEE, 2016.

[29] 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*, 2014, pp. 277–287.

[30] B. Lindstrm and A. Mrki, "On strong mutation and subsuming mutants. ieee workshop on mutation analysis (mutation 2016)." in *Workshop on Mutation Analysis*, 2016.

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

[32] R. A. DeMillo and R. J. L. 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] K. S. H. T. Wah, "A theoretical study of fault coupling," *Software Testing, Verification and Reliability*, vol. 10, no. 1, pp. 3–45, 2000.

[35] ——, "An analysis of the coupling effect i: single test data," *Science of Computer Programming*, vol. 48, no. 2, pp. 119–161, 2003.

[36] R. Gopinath, , C. Jensen, and A. Groce, "The theory of composite faults," in *Software Testing, Verification and Validation (ICST), 2017 IEEE Eighth International Conference on*. IEEE, 2017.

[37] A. J. Offutt, "The Coupling Effect : Fact or Fiction?" *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, Nov. 1989.

[38] ——, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, 1992.

[39] W. B. Langdon, M. Harman, and Y. Jia, "Efficient multi-objective higher order mutation testing with genetic programming," *Journal of systems and Software*, vol. 83, no. 12, pp. 2416–2430, 2010.

[40] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *International Symposium on Software Reliability Engineering*, Nov 2014, pp. 189–200.

[41] G. J. Myers, "The art of software testing," *A Willy-Interscience Pub*, 1979.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

[48] W. E. 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.

[49] 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.

[50] A. P. Mathur and W. E. Wong, "Evaluation of the cost of alternate mutation strategies," in *Brazilian Symposium on Software Engineering (SBES)*. Brazilian Computer Society, 1993, pp. 320–335.

[51] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Constrained mutation in c programs," in *Brazilian Symposium on Software Engineering (SBES)*. Brazilian Computer Society, 1994, pp. 439–452.

[52] 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.

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

[54] A. S. 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.

[55] 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.

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

[57] R. Gopinath, M. A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "Measuring effectiveness of mutant sets," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016.

[58] D. Shin and D.-H. Bae, "A theoretical framework for understanding mutation-based testing methods," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2016.

[59] B. Lindström and A. Márki, "On redundant mutants and strong mutation," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2015.

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

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

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

[63] S. Hussain, "Mutation clustering," Master's thesis, Kings College London, Strand, London, 2008.

[64] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 179–188.

[65] J. Siegmund, N. Siegmund, and S. Apel, "Views on internal and external validity in empirical software engineering," in *International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2015, pp. 9–19.

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

[67] Apache Software Foundation, "Apache maven project," http://maven.apache.org.

[68] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

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

[70] P. Ammann, "Transforming mutation testing from the technology of the future into the technology of the present," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2015.

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

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

[73] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[74] A. (http://cstheory.stackexchange.com/users/37275/anonymous), "What is the reverse of greedy algorithm for setcover?" Theoretical Computer Science Stack Exchange, url:http://cstheory.stackexchange.com/q/33685 (version: 2016-01-29). [Online]. Available: http://cstheory.stackexchange.com/q/33685

[75] U. Feige, "A threshold of ln n for approximating set cover," *J. ACM*, pp. 634–652, 1998.

[76] 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.

[77] 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.

[78] A. L. Gibbs and F. E. Su, "On choosing and bounding probability metrics," *International statistical review*, vol. 70, no. 3, pp. 419–435, 2002.

[79] R. Gopinath, M. 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

[80] J. Kopf, D. Cohen-Or, O. Deussen, and D. Lischinski, "Recursive wang tiles for real-time blue noise," *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2006)*, vol. 25, no. 3, pp. 509–518, 2006.

[81] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," *International Conference on Software Engineering*, pp. 919–930, 2014.

[82] M. E. Delamaro, A. J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 11–20.

[83] B. J. Grun, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2009, pp. 192–199.

[84] 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.

[85] I. Dinur and D. Steurer, "Analytical approach to parallel repetition," in *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, ser. STOC '14, 2014.

APPENDIX

Computing Approximate Minimum Testsuite

The computation of the *minimum* test suite is an instance of the problem of *set cover*. It is one of the well known *NP-Complete* problems, and hence the *minimum test suite* can only be approximated.

There are two main algorithms for computing the approximate minimum test suite. The *greedy* algorithm and the *reverse greedy* algorithm.

Greedy Algorithm: In the *greedy* algorithm given in Algorithm 1, the approximate minimum set is built step by step by finding the next best test cases to incorporate into the minimum set.

Algorithm 1 Finding the approximate minimum test suite

function GREEDYMINTEST($Tests, Mutants$)
  $T \leftarrow Tests$
  $M \leftarrow kill(T, Mutants)$
  $T_{min} \leftarrow \emptyset$
  while $T \neq \emptyset \vee M \neq \emptyset$ **do**
    $t \leftarrow random(\max_t |kill(\{t\}, M)|)$
    $T \leftarrow T \setminus \{t\}$
    $M \leftarrow kill(T, Mutants)$
    $T_{min} \leftarrow T_{min} \cup \{t\}$
  end while
  return T_{min}
end function

This algorithm achieves an approximation bound of $k(1 + ln(\frac{n}{k}))$ sets where $n = |U|$ (the total number of elements) and k is the size of the minimal set. Further, it has been shown [85] that if there exists a better approximation, such that it can approximate set cover in $C\,ln(n)$, where $C \leq 1$, then $P = NP$.

Reverse Greedy Algorithm: In the reverse greedy algorithm (Algorithm 2), for each iteration, we remove the least effective test that is not required for maintaining the mutation score.

Algorithm 2 Finding the approximate minimum test suite

function REVERSEGREEDYMINTEST($Mutants, Tests$)
  $T \leftarrow Tests$
  $M \leftarrow kill(T, Mutants)$
  $T_r \leftarrow \{t : kill(T \setminus \{t\}, M) = kill(T, M)\}$
  while $T_r \neq \emptyset$ **do**
    $t \leftarrow \min_{t \in T_r} |kill(\{t\}, M)|$
    $T \leftarrow T \setminus \{t\}$
    $T_r \leftarrow \{t : t \in T_r \wedge kill(T \setminus \{t\}, M) = kill(T, M)\}$
  end while
  return T
end function

We demonstrate that this algorithm will have a worse guarantee [74] than that of the greedy algorithm. Say we have a universal set of mutants $U = \{1 \ldots 2n\}$. For every $i = 1, .., n$, define a set with $n + 1$ elements by $S_i = \{i, n + 1, \ldots, 2n\}$, and a set $A = \{n + 1, \ldots, 2n\}$. Say we want to cover U with the sets $C = \{S_1, \ldots, S_n, A\}$. Now, the least effective set is A because it covers only n elements when compared to S_i which covers $n + 1$ elements. Hence it is discarded first, and the algorithm will return a cover with $\{S_1, \ldots, S_n\}$, with size n. However, the minimum cover is just $\{S_1, A\}$, with size 2. That is, the guarantee of approximation can not be better than that of greedy but could be worse.

The operator selection results for all projects (standard deviation).

Strategy	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
Constrained	4.32	7.92	12.17	**2.72**	3.56	3.65
E-Selective	2.26	9.48	9.76	6.64	2.24	0.66
S-Selective	0.18	6.44	6.55	8.25	0.18	0.19
R-Selective	3.95	14.13	20.78	4.29	3.81	1.48
ES-Selective	**0.02**	3.54	4.30	4.29	**0.02**	**0.02**
RS-Selective	2.07	8.70	10.20	8.25	2.06	0.35
RE-Selective	0.12	**3.46**	3.37	6.64	0.12	0.09
Javalance	0.31	7.12	5.52	4.50	0.31	0.16
VarReduction	0.05	4.75	4.17	5.90	0.05	0.05
SDL	0.23	3.85	**2.44**	7.49	0.23	0.09

TABLE XI: The operator selection strategy - standard deviation

Removed	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
rm.nmc	**0.06**	**4.60**	**5.23**	6.86	**0.06**	**0.06**
rm.rv	0.13	9.34	8.74	7.51	0.13	0.14
rm.ic	0.31	10.33	8.67	5.39	0.30	0.20
rm.dc	0.30	9.65	8.39	5.90	0.30	0.28
rm.nc	0.53	10.81	8.29	5.59	0.52	0.48
rm.rc	1.44	12.43	10.32	5.17	1.41	0.85
rm.vmc	2.70	10.50	12.41	4.80	2.65	0.97
rm.cc	5.72	12.41	16.37	5.03	4.82	5.77
rm.emv	8.41	11.49	17.63	5.17	6.72	12.50
rm.m	9.05	12.33	23.15	2.55	7.15	12.56
rm.cb	17.85	13.14	25.74	1.79	12.14	15.18
rm.i	22.03	22.86	28.05	1.31	13.87	19.51
rm.ri	28.24	19.77	35.09	1.09	16.01	38.52
rm.rs	41.41	16.20	36.77	0.21	12.63	33.22
rm.es	54.96	15.33	57.23	**0.12**	10.48	21.52

TABLE XII: The N-selective strategy. Each row removes the named operator from the preceding row - standard deviation

Fraction	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2	**0.05**	**1.42**	**1.64**	**0.03**	**0.05**	**0.21**
1/4	0.12	1.90	2.10	**0.03**	0.12	0.62
1/8	0.58	3.04	4.41	**0.03**	0.55	1.59
1/16	1.39	2.97	4.49	**0.03**	1.18	3.60
1/32	1.54	4.84	6.32	0.04	1.25	7.11
1/64	3.89	4.89	12.43	**0.03**	2.55	12.20

TABLE XIII: The operator-based x% sample strategy - standard deviation

Fraction Elt	Utility	Test Utility	Assert Utility	Mutation Share	Strategy Effectiveness	Baseline Effectiveness
1/2 line	0.09	1.63	2.16	0.16	0.09	**0.18**
1/4 line	0.16	1.77	2.52	0.15	0.16	0.63
1/8 line	0.70	2.81	2.98	0.16	0.67	1.48
1/16 line	2.01	3.45	4.72	0.19	1.63	4.00
1/32 line	2.16	5.85	7.88	0.15	1.53	7.03
1/64 line	2.15	5.59	7.93	0.10	1.70	12.31
1/2 method	0.12	2.01	2.09	0.10	0.12	0.20
1/4 method	0.18	2.48	2.35	0.08	0.17	0.62
1/8 method	0.69	3.32	3.92	0.09	0.66	1.38
1/16 method	1.37	5.79	6.63	0.09	1.20	3.48
1/32 method	2.22	4.55	7.72	0.07	1.67	7.33
1/64 method	6.20	10.23	11.25	0.08	2.55	12.97
1/2 class	**0.07**	**1.18**	**1.86**	**0.04**	**0.07**	**0.18**
1/4 class	0.14	1.78	2.35	0.05	0.14	0.61
1/8 class	0.36	2.36	3.09	0.05	0.35	1.39
1/16 class	0.77	4.55	5.02	**0.04**	0.70	3.20
1/32 class	4.22	3.81	7.11	0.06	2.71	7.45
1/64 class	4.03	5.95	10.69	0.05	2.18	12.18

TABLE XIV: The element-based x% sample strategy - standard deviation