How Hard Does Mutation Analysis Have to Be, Anyway?

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

***Abstract*—Mutation analysis is considered the best method for measuring the adequacy of test suites. However, the number of test runs required for a full mutation analysis grows faster than project size, which is not feasible for real-world software projects, which often have more than a million lines of code. It is for projects of this size, however, that developers most need a method for evaluating the efficacy of a test suite. Various strategies have been proposed to deal with the explosion of mutants. However, these strategies at best reduce the number of mutants required to a fraction of overall mutants, which still grows with program size. Running, e.g., 5% of all mutants of a 2MLOC program usually requires analyzing over 100,000 mutants. Similarly, while various approaches have been proposed to tackle equivalent mutants, none completely eliminate the problem, and the fraction of equivalent mutants remaining is hard to estimate, often requiring manual analysis of equivalence.**

In this paper, we provide both theoretical analysis and empirical evidence that a *small constant sample of mutants* yields statistically similar results to running a full mutation analysis, regardless of the size of the program or similarity between mutants. We show that a similar approach, using a *constant sample of inputs* can estimate the degree of stubbornness in mutants remaining to a high degree of statistical confidence, and provide a mutation analysis framework for Python that incorporates the analysis of stubbornness of mutants.

I. Introduction

Traditional mutation analysis [1], [2] involves exhaustive generation of first order mutants, the detection of which is used as a measure of test suite effectiveness. Studies by Andrews et al. [3], [4], and more recently by Just et al. [5] suggest that mutation analysis is capable of generating faults that resemble real bugs, the ease of detection for mutants is similar to that for real faults, and the effectiveness of a test suite in detecting real faults is reflected in its mutation score.

A barrier to the wider adoption of mutation analysis in practice is its high computational cost. Performing mutation analysis on large programs requires analysis of an even larger number of mutants, requiring multiple runs of a test suite. Modern software often has a million or more lines of code [6], and it is impractical to evaluate all mutants when their number is a multiple of such magnitude.

A major area of research in mutation analysis is therefore the reduction of computational requirements by reducing the number of mutants, called the *do fewer* approach [7]. This approach is generally divided into selective and sampling approaches. Selective approaches attempt to avoid low utility mutants (according to various definitions of utility), and compute the mutation score using only what are deemed to be high utility mutants [8], [9]. On the other hand, sampling approaches seek to randomly select a representative set of mutants, which can then be used to approximate the full mutation score [10], [11]. Wong et al. [12] used operator based stratified sampling and found that using as few as 10% of the total mutants can provide accurate results. Researchers have recently evaluated the relative merits of pure random sampling against stratified random sampling based on operators, program elements, and combinations of operator based and program element based stratified sampling [13], [14]. They found that stratified sampling using a combination of different strata was superior to stratified sampling of strata in isolation, or to pure random sampling. They also found that sampling approaches can approximate full mutation score as well as operator selection methods. A recent promising result was that the fraction of adequate mutants tends to grow at a rate $O(n^{0.05...0.25})$ where n is the size of the program for programs below 16KLOC [15].

While the suggested methods are effective, the number of mutants required is still a function of program size, which is still too large for many modern programs. Our research answers the following question: does a lower bound for the sample size of mutants exist that guarantees a reasonable absolute error[1] for mutation score, if we use at least that many mutants (sampled randomly)? If such an absolute lower bound exists, the cost of mutation analysis can be decoupled from the size of the project, and analysis can be resolved in a fixed number of test runs.

Such a lower bound would have practical impact because it would guarantee a ceiling for the amount of time and effort practitioners must invest in using mutation analysis to evaluate the quality of their test suites. Such a guarantee might help sway testers looking at using such tools for the first time, but not wanting to over-commit before they see a return on their time and effort.

We use Tchebysheff's inequality[2] to show that such a lower bound does exist. We find that, theoretically, a sample size as low as $1,000$ mutants can, with a probability of 95%, provide an approximate mutation score with *absolute error* as low as $\pm 7\%$. To validate that finding, we performed an extensive empirical study on 158 open source Java projects. Surprisingly, we found that a sampling of just $1,000$ mutants results in an absolute error as low as $\pm 2\%$, much lower than predicted.

A secondary concern in mutation analysis is the prevalence of equivalent mutants [17], [18]. These are mutants

[1]Note that absolute error is the difference of actual parameter and the estimated one.

[2]Nearly all values are close to mean [16].

which are semantically identical to the original program, and in a general sense their identification is undecidable [19]. Since determining mutation adequacy [20] is dependent on the number of equivalent mutants — in general, one doesn't know whether the remaining mutants can be killed by adding new test cases or if they are equivalent — the utility of mutation adequacy as a stopping criterion is severely weakened by this problem. The number of equivalent mutants reported in the literature varies widely, and is heavily dependent on the subject program, ranging from 2% [21] up to 50% [22], [23]. Hence assuming a fixed percentage to be equivalent is not justifiable in practice. Human evaluation of equivalence is both exorbitantly expensive [22] and error prone, with fault rates up to 20% [10]. This has even led researchers to abandon all attempts at actually evaluating equivalent mutants in unkilled mutants, and to assume that a given test suite is mutation adequate [13]–[15], [24]. While there have been various attempts [18] at providing heuristics for recognition of some equivalent mutants, none provides a bound on even the minimum stubbornness[3] expected of remaining mutants classified as equivalent, which is important for a practicing tester who wishes to know whether a test suite satisfies the required adequacy score.

We propose a simple extension to mutation analysis to resolve this dilemma: treating the closeness of two functions as a statistical problem. To identify whether a mutated function is semantically close to the original (with a given confidence), generate a fixed number of random inputs from the full input domain and verify that original and mutant functions behave equivalently. We show that the sample size suggested by our statistical framework is also applicable for quantifying mutant stubbornness, and for a *fixed* number of random inputs one can achieve *sufficient confidence* that two functions are indeed semantically similar to a given tolerance level.

The main contributions of this paper are therefore:

- We describe a statistical framework (Section III) to find the *fixed* minimum number of mutant samples required to achieve a certain absolute accuracy irrespective of the total number of mutants.
- We evaluate this bound using a large number of real-world Java programs (Section IV) and show that the lower bound on the sample size needed is not high, and sample sizes as small as $1,000$ can achieve good accuracy.
- We make available a new byte-code mutation framework for Python (Section V) that incorporates statistical determination of stubbornness in surviving mutants, and describe preliminary research into analysis of equivalence using statistical sampling.

II. Related Work

The idea of mutation analysis was first proposed by Lipton [1], and its main concepts were formalized by DeMillo et al. in the "Hints" [25] paper. The first implementation of mutation analysis was provided in the PhD thesis of Budd [26] in 1980.

[3] A stubborn mutant is a hard to kill mutant. Stubbornness is the effort required to kill it.

Previous research in mutation analysis [11], [27], [28] suggests that it subsumes different coverage measures, including *statement*, *branch*, and *all-defs* dataflow coverage [11], [27], [28]. There is also some evidence that the faults produced by mutation analysis are similar to real faults in terms of error trace produced [29] and the ease of detection [3], [4]. Recent research by Just et al. [5] using 357 real bugs suggests that the mutation score increases with test effectiveness for 75% of the cases, which was better than the 46% reported for structural coverage.

The validity of mutation analysis rests upon two fundamental assumptions: "The competent programmer hypothesis" — which states that programmers tend to make simple mistakes, and "The coupling effect" — which states that test cases capable of detecting faults in isolation continue to be effective even when faults appear in combination with other faults [25]. Evidence of the coupling effect comes from theoretical analysis by Wah [30], [31] and empirical studies by Offutt [32], [33].

Researchers have suggested several approaches to reducing the cost of mutation analysis, categorized as *do smarter*, *do faster*, and *do fewer* by Offutt et al. [7]. The *do smarter* approaches include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis. The *do faster* approaches include mutant schema generation, code patching, and other methods to make the mutation analysis faster as a whole. Finally, the *do fewer* approaches try to reduce the number of mutants examined, and include selective mutation and mutant sampling.

Various studies have tried to tackle the problem of approximating the full mutation score without running a full mutation analysis. The idea of using only a subset of mutants (*do fewer*) was conceived first by Budd [11] and Acree [10] who showed that using just 10% of the mutants was sufficient to achieve 99% accuracy of prediction for the final mutation score. This idea was further investigated by Mathur [34], Wong et al. [12], [35], and Offutt et al. [8] using the Mothra [36] mutation operators for FORTRAN. Lu Zhang et al. [13] compared operator-based mutant selection techniques to random mutant sampling, and found that random sampling performs as well as the operator selection methods. Lingming Zhang et al. [14] compared various forms of sampling such as stratified random sampling based on operator strata, stratified random sampling based on program element strata, and a combination of the two. They found that stratified random sampling when strata were used in conjunction performed best in predicting the final mutation score, and as few as 5% of mutants were sufficient sample for a 99% correlation with the actual mutation score. Hsu et al. [37], [38] used "Binomial Sequential Probability Ratio Test" as a stopping rule for i.i.d. (independent, identically distributed) random variables. They found that only 299 mutants needed to be sampled for 1% tolerance within 99% confidence interval.

A number of studies also measured the redundancy among mutants. Ammann et al. [39] compared the behavior of each mutant under all tests and found a large number of redundant mutants. More recently, Papadakis et al. [40] used the compiled representation of programs to identify equivalent mutants. They found that on average 7% of mutants are equivalent while 20% are redundant.

This paper deviates from the above studies in three major ways. First, while other studies suggest using a *fraction* of mutants for testing, in this paper, we study the usefulness of selecting a *fixed number* of mutants to approximate the mutation score and the accuracy of the approximation. Second, most of the above *do-fewer* approaches (e.g. Lu Zhang et al. [13] and Lingming Zhang et al. [14]) base their validity on empirical studies. We offer a statistical foundation for our technique in addition to empirical validation. Third, our proposed statistical framework does not require an assumption of independence. This is important especially since other studies (e.g. Papadakis et al. [40] and Ammann et al. [39]) indicate that mutants are rarely independent, and largely similar with each other.

Researchers have also focused on identifying equivalent mutants, with this work generally divided into the prevention and detection camps [40]. The prevention camp is concerned with reducing the incidence of equivalent mutants by identifying operators that produce a low number of equivalent mutants, and favoring them [22]. The detection camp tries to detect equivalent mutants by various static and dynamic properties of the mutant. These include efforts to identify them using compiler equivalence [17], [40], [41] dynamic analysis of constraint violations [42], [43], and coverage [44]. Recent research by Cai et al. [45] provides a way to represent simple changes to input values, which can also be used to represent changes in functions (i.e. mutants), which could also be used to evaluate semantic impact.

The main difference between these studies and the procedure of quantifying stubbornness in mutants suggested by us is that, while previous researchers have suggested many ways to mitigate the equivalent mutant problem, none have given procedures to analyze confidence in their classification, neither for the equivalent mutants identified (except for definite identification), nor for the remaining live mutants in the code. Using the procedure we outline, we provide estimates on our confidence in the stubbornness of remaining mutants.

III. Theoretical Analysis

In this section, we explain our statistical framework for finding a lower bound, n, for mutant sample sizes. The goal is to approximate the mutation score, m, while ensuring that absolute error does not exceed ϵ, with a confidence interval of $1-\delta$. That is, the probability of the approximated mutation score being out of the accepted error range is δ.

We also show that a similar analysis can provide a lower bound for the number of inputs to be examined for providing a confidence level in classifying equivalent mutants approximately.

A. How many mutants should we sample?

Running a test suite on a mutant can have at most two possible outcomes; either it will be detected, or it won't be. Thus the mutation score can be modeled as the mean of a number of trials of random variables. Since mutation analysis primarily involves modifying a single token at a time, we assume that some of the mutants are strongly correlated with detection of other mutants. In fact, a number of studies [39], [46] have found that there exist redundant mutants which are semantic copies of each other, and equivalent mutants, which are semantic copies of the original version. Similarly, we assume that few mutants are *negatively correlated* – that is, detection of one mutant does not imply that another mutant will *not* be detected (or at least this kind of relation is much rarer than mutants with a positive correlation). Thus the only assumptions that we need for our analysis are:

Assumptions:

- Mutant detection is more positively than negatively correlated.
- The total number of mutants is large.

Note: Our analysis **does not require independence** between detection of different mutants.

Let the random variable D_n denote the number of detected mutants out of our sample n (to be determined). The estimated mutation score is given by $M_n = \frac{D_n}{n}$. The random variable D_n can be modeled as the sum of all random variables representing mutants $X_{1..n}$. That is, $D_n = \sum_i^n X_i$. The expected value $E(M_n)$ is given by $\frac{1}{n}E(D_n)$. The mean of the sum of random variables does not depend on their independence, hence $E(M_n) = m$. The variance $V(M_n)$ is given by $\frac{1}{n^2}V(D_n)$, which can be written in terms of component random variables $X_{1..n}$ as:

$$\frac{1}{n^2}V(D_n) = \frac{1}{n^2}\sum_i^n V(X_i) + 2\sum_{i<j}^n Cov(X_i, X_j)$$

Using our simplifying assumption that some of the mutants are similar, we can assume that

$$2\sum_{i<j}^n Cov(X_i, X_j) >= 0 \tag{1}$$

That is, the variance of the mutants $V(M_n)$ is greater than or equal to that of a similar distribution of *independent* random variables.

Problem statement The following inequality formalizes the constraints of the problem, which states that the probability of absolute error exceeding ϵ is lower than δ.

$$Pr[|M_n - m| \geq \epsilon] \leq \delta$$

We use Tchebysheff's inequality to draw a lower bound for the mutant sample that satisfies the above formula. Tchebysheff's inequality states that,

$$\forall k : P(|x-\mu| \geq k) \leq \frac{V}{k^2}$$

where μ is the mean, V the variance of the distribution, and $k > 0$. Replacing variables in Tchebysheff's inequality, we have

$$Pr[|M_n - m| \geq \epsilon] \leq \frac{V(M_n)}{\epsilon^2}$$

We want to ensure that $Pr[|M_n - m| \geq \epsilon]$ is lower than δ, so we restrict the bound from Tchebysheff's inequality to be lesser than or equal to δ.

$$\frac{V(M_n)}{\epsilon^2} \leq \delta$$

By replacing M_n with $\frac{D_n}{n}$ we have,

$$\frac{V(D_n)}{n^2\epsilon^2} \leq \delta \tag{2}$$

Remember that we are looking for the minimum number of samples that will give us the required accuracy. That is, overestimating this minimum number (i.e. n) only improves the accuracy of the estimate. Hence we look for a conservative estimate of n that satisfies Equation (2). Notice that n is in the denominator, and hence a lower value of the term $\frac{V(D_n)}{n^2\epsilon^2}$ corresponds to a higher value of n. So consider the solution for n in Equation (2). If $V(D_n)$ was underestimated, then a solution of n from that equation would result in a larger n than that which corresponds to the correct $V(D_n)$.

We have seen previously from Equation (1) that covariance of *mutant detection* is greater than or equal to that of a similar distribution of *independent* random variables. So if we assume independence for D_n, the term $V(D_n)$ would be smaller than actual, and hence the value of n would be larger than actual.

So, we will now solve Equation (2) under the assumption that mutants are independent, since this will provide the maximum sample size required. D_n is considered binomially-distributed because of the (conservative) independence assumption.

Now consider D_n. If we consider the set of mutants that we are sampling, each of them could be either detected or not by the given test suite. That is, if k is the set of mutants that were detected from the complete population of N mutants, then you have $\frac{k}{N} = m$ chance of picking a mutant that will be detected by the given test suite. Note that it is easy to fall into the trap of thinking that some mutants are easy to detect, and hence they should have a different probability. The thing to note is that, this intuition is based on assuming a random test suite. That is, different mutants have different probabilities of being detected by a random test suite. However, once the test suite is fixed — as it is when we try to estimate the mutation score for a suite — the mutants will be either detected or not by the fixed suite, and the probability of detection of any single mutant by that test suite is the ratio of detected mutants to the total population, m.

D_n is the binomial distribution; replacing $V(D_n)$ with the variance of the $binomial(n, m)$ distribution we get:

$$V(D_n) = n \times m(1-m) \Rightarrow n \geq \frac{m(1-m)}{\epsilon^2\delta}$$

Notice that the sample size n will be largest when $m = \frac{1}{2}$. So in the worst case, this formula can be rewritten as

$$n \geq \frac{1}{4\epsilon^2\delta} \tag{3}$$

Inequality (3) can be used to find the lower bound. That is, given a certain sample size n, and a confidence interval $1-\delta$, we can compute the tolerance ϵ. For example, for $n = 1000$, and $\delta = 0.05$, we have $\epsilon = 0.07$. Note that this bound is a **pessimistic lower bound**.

However, since we have shown that detection of mutants is *bounded* by the binomial distribution for a given test suite, we can rely on stronger tools than Tchebysheff's inequality, if we are willing to allow approximation of the distribution. For a large enough n, a binomial distribution approximates a normal distribution [47][4]. For a normal distribution, ϵ is given by

$$\epsilon = \frac{\sigma}{\sqrt{N}} \times z_{1-\frac{\delta}{2}}$$

where $z_{1-\frac{\delta}{2}}$ is the normal score, the probability that the mean lies within the constraint δ. That is, given that $\sigma^2 \leq m(1-m) \leq 0.25$

$$N \geq \left(\frac{z_{1-\frac{\delta}{2}}}{\epsilon}\right)^2 \times 0.25$$

For $\epsilon = 0.01$ and $\delta = 0.05$, $N \geq 9,604$, which is smaller sample size than that predicted using Tchebysheff's inequality $(50,000)$. This means that for programs where the number of mutants is larger than $9,604$, we can guarantee that a sample size of $9,604$ will approximate the real mutation score with 99% accuracy for 95% of the samples.

An important aspect of this analysis is that our results hold **even if the mutants are not independent**. In fact, the more similar to each other the mutants are, the smaller the number of samples needed to estimate the true mutation score. Assume that we have 10 mutants which are copies of each other. In this case, choosing a sample of just one mutant is sufficient to tell us whether the entire set of mutants can be detected, when compared to a set of 10 dissimilar mutants. In fact, our empirical analysis suggests that a much smaller sample size than predicted by our theory is sufficient to estimate mutation score to a high degree of accuracy. We validate our lower bound empirically in Section IV.

Summary: Since mutants are similar to each other, the distribution of detected mutants has positive covariance. Thus the number of samples required is strictly smaller than for a binomial distribution. Note that mutants are similar both due to construction, and empirically [39], [46]. Hence the binomial distribution provides an *upper* bound on required sample size (similarity of mutants works in our favor).

B. How many inputs should we sample for confidence in equivalence classification?

For analysis of equivalent mutants, consider a program P with a single input and its mutant Q[5]. Our goal is to evaluate whether the mutant is equivalent to the program.

Let n be the input domain[6]. Consider a function F that takes no input, but compares the functions P and Q for some value given by

$$F = P(i) == Q(i)$$

[4] Berry-Esseen theorem suggests that error in normal approximation is bounded by $\frac{C(p^2+q^2)}{\sqrt{npq}}$, where $C \leq 0.7655$ and $q = 1-p$. It can be bounded by $\frac{0.7655}{\sqrt{nq}}$ for $p > 50\%$. That is, for $1,000$ mutants with a mutation score of 90%, the maximum error of normal approximation is 0.007655

[5] A function with any number of inputs can be transformed to a function that takes a single input by wrapping the input in a tuple.

[6] Practically, the input domain is limited by the underlying language, system capacity etc. even for seemingly infinite types such as integers, lists, and recursive data structures, and our analysis does not rely on the size of n.

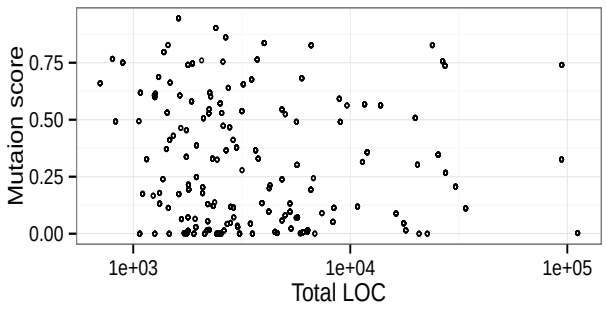

Fig. 1. The distribution of size of project in LOC and mutation score. It shows that we have a representative sample of projects and mutation scores.

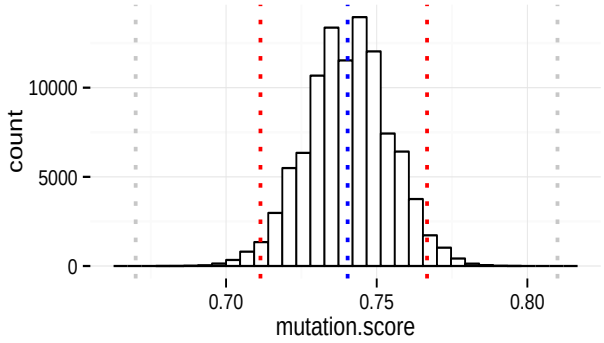

Fig. 2. The distribution of mean mutation score when sample size is $1,000$ for Apache commons-math. The central blue line shows true mean at 74.03, and the red lines mark 95% confidence intervals. Similarly the gray lines show the theoretical bounds for 95% confidence intervals. This shows that the sample variation is well within theoretical bounds.

where $i \in n$. We can now mutate this function to F' by changing the value of i to i'

$$F' = P(i') == Q(i')$$

We can see that the total number of mutants thus produced would be $|n|$. We can use the same statistical approach we outlined earlier to approximate how close P is to Q, in a constant number of mutants of F, as we do not violate either of the requirements of our statistical framework: a large number of mutants, and a positive correlation between *detectability* of mutants. From the analysis in the previous section, $50,000$ input samples is sufficient for a 95% confidence that the mutant and original differ by less than 1% of the possible input values (or $9,604$ using normal approximation). This means that if a function and its mutant differ on at least 1% of input values, a sample of $9,604$ input values will identify it 95% of the time.

The point to note here is that we estimate not the equivalence of the mutants, but the degree of stubbornness at the function level, which is an *upper bound* on probability of detecting that mutant for tests targeting that function. Empirically mutants that are resistant towards detection at the function level tend to be equivalent. It may be objected here that there are mutants that are killed by a single test case, and the procedure given is not applicable to such mutants (our statistical guarantee is that if the ratio of distinguishing inputs to non-distinguishing inputs exceeds a certain threshold, we have a high probability of finding them). However, while there exist mutants killed only by a single input, large input domain fault patterns, with a fixed ratio between distinguishing and non-distinguishing inputs (and hence amenable to our statistical analysis) are likely more common [48].

IV. Evaluation of Mutation Sampling

A. Methodology

For our empirical evaluation, we tried to ensure that the programs chosen offered a reasonably unbiased representation of modern software. We also attempted to reduce the number of variables that can contribute to random noise during evaluation. Keeping these goals in mind, we chose a sample[7] of Java projects from Github [49] and similarly from the Apache Software Foundation [50]. All projects selected used the popular maven [51] build system. This gave us $1,800$ projects. From these, we eliminated aggregate projects that were difficult to analyze, resulting in $1,321$ projects, of which only 796 had test suites. Out of these, 326 remained after eliminating projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations). Next, the projects that did not pass their test suites were eliminated as mutation analysis requires a passing test suite. Finally, we only chose projects that were non-trivial; i.e. had at least $1,000$ mutants[8]. This follows the methodology we used in our previous work [52], and resulted in 158 projects selected. The project size and mutation score distribution is given in Figure 1, which shows that we have a reasonably non-biased distribution in terms of detection.

Note that we have a much larger set of large sized projects (158 projects with mean 7061 LOC) than previous studies such as Namin et al. [24] (7 projects with mean 312 LOC), Zhang et al. [13] (7 projects with mean 312 LOC), Zhang et al. [14] (7 projects with mean 15083 LOC), and Zhang et al. [15] (12 projects with mean 6209 LOC). Similarly, our test subjects have large test suites (441.766 sd 920.385) in comparison to previous studies — Namin et al. [24] (mean 3115.286, sd 1572.038), Zhang et al. [13] (mean 3115.286, sd 1572.038), Zhang et al. [14] (mean 3115.286, sd 1572.038), and Zhang et al. [15] (mean 81, sd 29.061).

Our subjects had 90 test suites with more than 100 tests (for comparison, other studies for e.g. Zhang et al. [15] have only three test suites with more than 100 test cases)

Similarly, the mutation scores of our subjects (mean 0.311 sd 0.275), and *Apache commons-math* in the second part with mutation score 0.74, are also comparable to previous studies such as Namin et al. [24] (mean 0.322, sd 0.318), Zhang et al. [13] (mean 0.831, sd 0.055), Zhang et al. [14] (mean 0.831, sd 0.055), and Zhang et al. [15] (mean 0.529, sd 0.256).

[7] Github allows us to access only a subset of projects using their search API. We believe that the results returned by Github search would not be dependent on their test suites, and hence should not confound our results.

[8] Some of the projects had 0 mutation score, which we opted to keep for the representativeness of the sample, but the results remain same even if they are removed. Without considering the zero mutation score projects, the mean was 34.103% (standard deviation 26.967).

In summary, our set of projects is fairly large, has large test suites, and has comparable mutation scores to suites in previous studies that handled similar sized projects. Other studies of similar nature had either smaller test suites, smaller sized subjects, and/or a much smaller number of subjects. Further, our study includes both low and high mutation scores as required to demonstrate the effectiveness of our technique[9].

We used PIT [54] for mutation analysis (used in multiple studies [52], [55]–[57]), extended to provide the full matrix of test failures over mutants and test cases.

B. Analysis

Our empirical analysis was done in two parts. First we looked in detail at a moderately large project with 90% statement coverage, to see if our predictions held true for a large number of repeated samplings. In the second part, we looked at the validity of our predictions across a diverse set of projects.

For the first part, we chose *Apache commons-math*, which is a medium large open source project, to evaluate the bounds. This is a 95KLOC project, with 122,484 mutants and a true mutation score of 74.03% detection.

For this project, we sampled $1,000$ mutants for each run, and computed the sample's mutation score. This was repeated $100,000$ times, and the resulting mean distribution is plotted in Figure 2. The central blue line indicates the true mean of the project at 74.03%. For our experiment, the mean was found to be 74.04%. We also estimated the 2.5% and 97.5% quantiles (95% of values lie between these quantiles). These were found to be 71.3%, and 76.7% respectively, plotted as the red lines in the figure. As we expected from our theory, and expected dependence between mutants, this is well within our theoretical prediction ($\pm 3.1\%$ using normal approximation, and $\pm 7\%$ using no approximation) for $1,000$ samples. The approximation lines for $74.03 \pm 7\%$ at 67% and 81% are plotted as gray lines in the figure.

For the second part we compared the mutation scores reported by sampling 158 projects with both 100 randomly sampled mutants, and $1,000$ sampled mutants. The results are shown in Figure 3. We found the mean *absolute difference*[10] from the true mutation scores when we used a sample size of 100 was 2.56%, and when the sample size was increased to $1,000$, it was reduced to 0.62%.

Next, we considered the quantiles at 2.5% and 97.5%. For the sample size of 100, they were found to be -7.216% and 7.206% respectively. That is, even if we sample just 100 mutants, we observe an error in approximation of at most 7.2% with 95% probability (we expected less than 9.8% from normal approximation and 22.3% with no approximation). The same quantiles on sample size of $1,000$ yielded -2.146% and 1.454% respectively. We note that this is a smaller range than what is predicted by our statistical analysis (which suggests that a sample of $1,000$ mutants results in only $\pm 7\%$ tolerance). This smaller bound is due to the fact that we assumed no dependence when calculating the bound, and secondly, the theoretical bound was predicted by assuming that the mutation score would result in the maximum variation at $m = 0.5$.

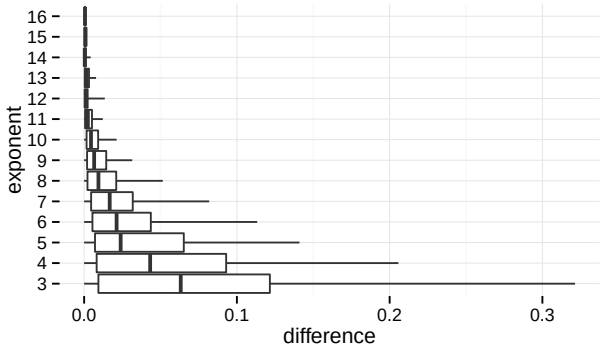

Fig. 4. This graph shows the progressive improvement of accuracy (x) as sample sizes increase as powers of 2 (2^y) for 158 projects. The boxes in boxplots represent 0.25 and 0.75 quantiles, and the ends of black lines represent quantiles at 0.025 and 0.975 respectively. This figure shows how precision improves as sample size increases.

In order to visualize how the tolerances change when sample size is increased, we plotted the summary of variation of 100 repetitions for each of the 158 projects, with mutant sample sizes corresponding to $(2^3, 2^4 ... 2^{16})$ (Figure 4). In the boxplots, the boxes represent the 25% and 75% quantiles, while the ends of the lines represent the 2.5% and 97.5% quantiles respectively. This figure suggests that as the sample sizes grow, accuracy also improves as expected.

To understand the impact of stratification and x% sampling in different strata, we plotted the full number of mutants vs an x% mutant sample where the x% was given by decreasing fractions of 2^{-i} — i.e. $(\frac{1}{2}, \frac{1}{4} \cdots \frac{1}{64})$ — for each of the stratification strategies, including operator stratification, element based stratification (for line, method and class), and combined stratification for line and operator. This was done for all 158 projects, and each measurement was repeated 100 times, with the mean plotted. Finally, we looked at the difference between the original mutation score and the sample mutation score, and marked all the observations that were more than 1% off as bad (red). This is following the limit of 99% frequently used by researchers [14] as sufficient to consider a subset to be representative of the full set of mutants. Results are shown in Figure 5. For clarity, we have chosen to restrict the figure to less than $12,000$ mutants, and we have only plotted $1,000$ points from the complete set. The six distinct lines in the graph represent fractions of powers of two in decreasing power $(\frac{1}{2} .. \frac{1}{64})$.

The figure suggests that above a certain constant threshold number of mutants, the accuracy is always better than 1%. Moreover, *the accuracy does not depend on the total number of mutants*.

V. Evaluation of Stubborn Mutants

As we show in Section III-B, our statistical framework is applicable to the problem of stubborn and equivalent mutants also. For our evaluation of the mutant sample size required for

[9] Please see [53] for detailed information on the projects and their test suites.

[10] By absolute difference, we mean the sign of difference was ignored. If we include the sign of difference, the mean differences were -0.08 and -0.03 respectively for size 100 and 1,000 samples

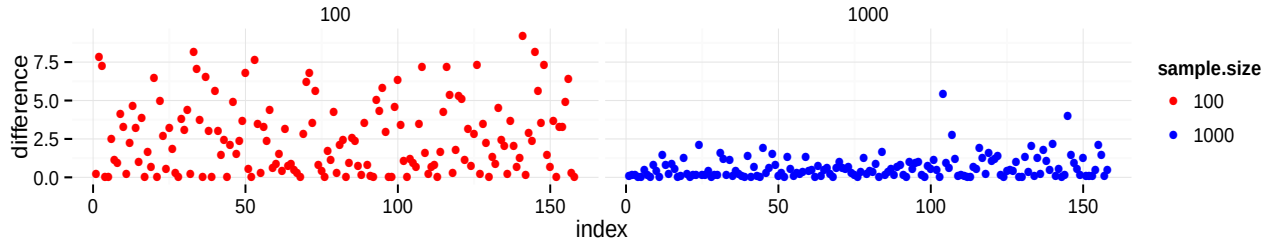

Fig. 3. The difference from true mutation score when sample size is 100, and when sample size is 1,000 in %. The projects are ordered by the total number of mutants, which shows that accuracy has no relation to project size.

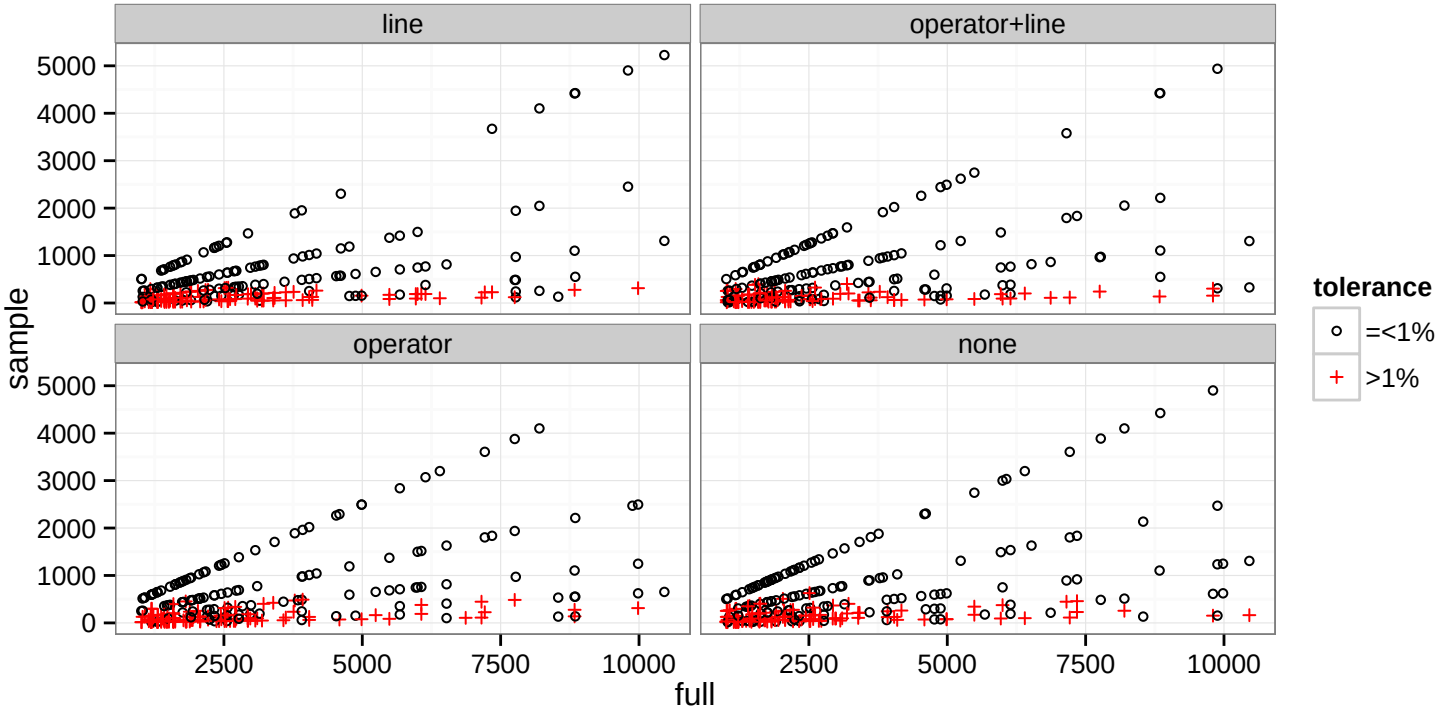

Fig. 5. This graph plots the full mutation score against the sample sizes suggested by different stratified sampling strategies (We show only the combinations of line and operator strata. Combinations of method and class with operator are similar). The red color (solid) indicates places where measured mean mutation score (from 100 repetitions) was different from the true mutation score by at least 1%. Notice how *all* red points are below 1000 irrespective of the strategy or sampling fractions. The distinct linear patterns that can be observed are due to the decreasing fractions of powers of two that we sampled. The plot is a random sample of 1,000 points out of 566,400 for clarity. This graph suggests that the sample size required for a reasonable accuracy is a constant.

stubborn mutants, we could not find a large enough sample of programs with known equivalent mutants. Further, none of the mutation analysis tools that we evaluated had any interface for incorporating automatic equivalent mutant analysis. This led us to develop our own mutation analysis tool and framework for evaluating stubbornness in live mutants.

A. Mutation Framework

Python [58] is a language frequently used by developers, which we chose as the platform for our experiment. Since Python is not statically typed, source modification of Python programs can lead to invalid programs. Recent results suggest a large number of source modifications result in expressions which can result in identically compiled code. While there exist mutation analysis programs for Python [59], they were either source or AST based, for Python 3 (which excluded well tested applications in Python 2), or they were abandoned prototypes [60], [61].

To avoid equivalence of compiled expressions and issues with invalid mutants, we started by extending *mutant* [61], a byte-code mutation analysis tool. We implemented traditional operators [9], [62] such as *modify numerical constants*, *negate jumps*, *replace arithmetic* and *binary operators*. We also made use of optimization techniques such as parallel execution of mutants and filtering by coverage to ensure that only relevant mutants were generated.

For the evaluation of stubborn mutants, we require that a function be annotated with the domain of its input parameters. Given the domain of the parameters, the framework can

generate random samples for any of the Python primitive types, non-recursive user defined types, and homogeneous containers.

However, generating random samples of the extremely large domains of containers such as lists is hard[11]. Further, the confidence we have in the equivalency result is only as good as the input we sampled it with. That is, if we consider binary search, where one expects a sorted list as input, the assurance that 99% of the input values do not result in a different output has very different meaning based on whether we are specifying the inputs to be sorted lists or just all possible lists. Since the validity of inputs is impossible to guess at, we allow functions to specify their own generators of all possible inputs.

Another issue is the problem of random sampling. We require access to all possible inputs in order to randomly select from them. However, it is infeasible to keep all possible input values in memory. To overcome this, we make use of the *reservoir sampling* algorithm [63] which allows us to ensure that we need to keep only the sample size number of input items in memory.

However, reservoir sampling trades space requirements for runtime, and this makes the runtime for sampling containers such as lists exorbitant if we attempt pure random sampling over the entire domain. One way to deal with this is to provide an estimate for a reduced domain – that is choose a subset of practical relevance or where we expect most bugs to hide based on an analysis of the function at hand[12].

Very preliminary research using our framework, XMutant [65] (which is available as an open source project, along with our evaluated programs) suggests that even under these constraints, random sampling is able to evaluate and distinguish trivial and stubborn mutants. Applying XMutant to binary search resulted in three mutants tagged as stubborn by $1,000$-sampling, which were also found to be equivalent by human analysis. Figure 6 shows the reduction of estimated probable equivalent mutants as the sample size increases for *timsort* using lists of size up to 7.

A few notes on our strategy are in order: if we are using mutation analysis to evaluate adequate unit tests, then mutants that are trivially detected during the sampling process show serious inadequacy in tests. Similarly, for evaluating adequacy of unit tests, while mutants that escape detection during sampling cannot be thrown out completely, they may be excluded from computing the mutation score with given confidence, as they are genuinely hard to kill and not just the sign of a very weak test suite.

If tests target the whole program, mutants we show are easy to kill at *function level* deserve considerable attention. They are either equivalent for subtle inter-procedural reasons (such as a function that will never get called with distinguishing inputs irrespective of program inputs), or show test suite deficiencies. In that case, very stubborn mutants may be excluded with some statistical guarantee that they are hard to kill.

[11]Here we assume that the domain of such functions is bounded by what can be supported by the underlying system. For example, Python supports less than *sys.maxint* items in a single array.

[12]Note that this can be optimized much further; for instance, using incremental computation [64], it is possible to reuse the computation done for one sample on others with little effort, and hence extend the range much further.

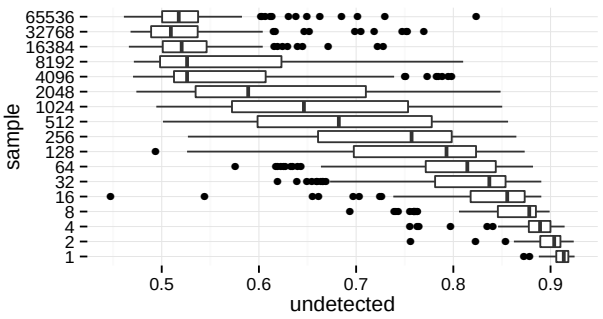

Fig. 6. The reduction in undetected mutants for *timsort* as sample size increases. Y axis represents the sample size and X axis the ratio of undetected mutants found. Experiments were repeated 100 times for each sample size.

What our strategy in essence achieves for whole program tests is to prioritize mutants for examination. Those mutants that look very easy to kill by sampling but are alive should be given priority in the order of sample size.

We also note that our algorithm for sampling of stubborn mutants can be improved further. For example, incremental computation [64] fits naturally into our scheme, and enables a higher number of samples to be drawn.

Notice that we are not claiming that random sampling is the best way to identify equivalence. For example, any patterns in input such as one would expect from the "Small Scope Hypothesis" [66] or geometric patterns in failure causing inputs [67] can aid detection. Rather, we are asking researchers to quantify the efforts made to eliminate equivalent mutants in the language of statistics so that it may be replicated. Adopting this recommendation also provides a sanity check against test suites that are coverage adequate, but are inadequate or have incorrect assertions, as was found in 65% of unit tests [68].

VI. Discussion

In this section, we discuss the results of the empirical validation of the statistical framework presented in Section III. Our statistical framework suggested that the absolute error for 1,000 mutant samples is less than 7%. We describe the actual absolute error that we observed in our programs and their test suites.

A. Case Study: Apache Commons Math

Figure 2, again, shows the histogram of mutation score for $100,000$ repetitions of 1,000-sampling for Apache commons-math, a nearly 100KLOC program with over $120,000$ mutants (100-sampling is also given for comparison). It shows that the largest absolute error for estimated mutation score is slightly more than 5% while absolute error in 95% of instances is lower than 2.7%.

Observation 1: 95% samples of size $1,000$ provide an absolute error less than 2.7%.

B. 1,000-sampling at large

While our statistical framework suggests a pessimistic expected error rate of $\pm 7\%$, empirical data, shown in Figure 3,

suggest that in practice, the absolute error is much lower (0.62% on average). Figure 3 also shows that absolute error of 1,000-sampling in very few projects exceeds 2.5%. This observation can be summarized as the following observation.

Observation 2: 1,000-sampling approximates mutation score with high accuracy, 0.62% on average.

The implications of Observation 2 are twofold. First, it suggests that the number of required mutants to accurately approximate the mutation score for a test suite is not a function of the total number of mutants (and therefore not tied to program size). Second, it suggests that a relatively small sample (as small as 1,000 mutants) is sufficient for estimating the mutation score with considerable accuracy.

In studies related to $x\%$ sampling, the actual sample size of mutants is often overlooked. Thus, in another part of our empirical evaluation, we performed $x\%$ *element-strata* based mutation sampling, suggested by [14], where $x = \frac{1}{2^k}$ for $1 \leq k \leq 6$. Figure 5 summarizes the result of evaluations of 158 projects. Black symbols in this scatter plot denote estimated mutation scores with an absolute error smaller than 1% and the red symbols show the absolute error larger than 1%. This figure shows that all *element strata* based samples of sizes $1,000$ could achieve an accuracy of 1%. But for sample sizes smaller than $1,000$, the error is higher in many instances. This suggests a limitation to the applicability of *element strata* based sampling of mutations. That is, the accuracy of sampling drops if the number of mutants is below a certain threshold. This can be demonstrated by considering 10% sampling on a trivial 10 mutant population with 1 detected mutant. In such a population, 10% sampling is not sufficient, since there is only $\frac{1}{10}$ probability of getting it right.

C. Practical implications

With mutation sampling, it becomes difficult to identify the components in larger systems that are in need of particular additional testing, especially when the larger systems (such as the ones that sampling seeks to address) are often composed of more than 1,000 interacting components (plainly, 1,000 sampling will not be of much help in identifying such components).

However, all is not lost. We have shown in our previous research that structural coverage techniques such as statement coverage [52], branch coverage [69], and path coverage [70] can provide sufficient information to isolate under-tested portions of code. We especially recommend simple statement coverage for this purpose, as it was shown to predict [52] mutation score with 94% accuracy (98% accuracy at branch coverage levels above 80%) for manual test suites.

We recommend the following procedure to identify under tested portions of code when $1,000$-sampling is used. Use at least $1,000$-sampling (we do not propose an upper limit) to evaluate whether the test suite of the complete system is sufficiently robust. For those mutants that are left alive from a $1,000+$-sample, use our stubborn mutant evaluation from Section III-B to identify mutants that are possibly equivalent with desired confidence. The mutation score is computed as the ratio between killed mutants in the sample, and the total number of mutants except those that were identified as probably equivalent ($\frac{M_{killed}}{M_{total} - M_{prob.equiv}}$). This should be used in conjunction with statement coverage to ensure that there are no obvious blind spots due to random sampling and to identify any components with unusually low mutations scores or coverage. Once the test suites are enhanced further to address such problems, a *new random sample* should be used to judge whether the test suite as a whole has desired quality.

VII. Threats to Validity

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

A. Threats to Theoretical Analysis

Our analysis relies on two assumptions, that the number of mutants involved is sufficiently large, necessitating a reduction by sampling, and secondly that either the detection of mutants is non-correlated, or they are largely positively correlated. While these assumptions seem to be in line with recent empirical results including the finding that there exist a number of equivalent mutants, and a number of redundant mutants (detection of which are positively correlated with each other), and the fact that the nature of mutation analysis is to make small changes that lends itself to mutants with similar behavior, there exists a possibility that this assumption may not be warranted even though current research strongly suggests that the assumption is true.

B. Threats to Empirical Analysis

Threats due to sampling bias: To ensure that our results were representative of real world programs, we opted to sample Java projects from the Github repository using the `Maven` build system. We used all projects that we could retrieve given the Github API, and constraints of building and testing. This however, implies that our sample of programs could be biased by any factor that skews the projects returned by Github.

Projects of small size and low coverage: Since we used real world projects with real test suites from Github, the size, coverage, and hence the mutation scores are representative of real world projects. Unfortunately given that a large majority of these projects are in the process of development, with many small (in LOC) personal projects, some of them have zero mutation score (even with a test suite), and in general, low statement coverage and mutation score, with very few adequate test sets. However, the accuracy of estimation remains within the predicted region even when we consider only the subset of projects which have high mutation coverage, which we show by the analysis of the *Apache-commons math* project, a large project of 94 KLOC with 90% statement coverage and 73.20% mutation score.

Bias due to tool used: Finally, we had to rely on the PIT mutation testing tool (since the other bytecode mutation tool, Javalanche was hard to get working for all programs in our repository, we opted for the tool that gave us the ability to analyze the largest number of programs), and had to extend its capabilities to some extent for our purposes. While PIT is a popular mutation analysis tool, it does have some drawbacks such as an incomplete repertoire of mutation operators and a smaller set of mutants produced per token. However, since

our research does not depend on any property of mutants produced per se (the evaluation could have been conducted on any random subset of mutants from a more traditional mutation system), we believe that our results are not affected by this decision. However, software bugs are a fact of life. While every care has been taken to avoid them, there is still some possibility of some bugs having escaped us. We also relied extensively on the R statistical platform for our analysis. Any bugs in the implementation of statistical tools that we used in R can have an impact on the accuracy of our results.

While these threats may cause our estimates to be inaccurate, our central message — to use a constant sized sample to approximate the full mutation score rather than an x% sample — is backed by statistical theory, and remains valid even if the threats we outlined have an impact on our estimates minimal sample size.

Finally, while our empirical analysis of stubborn mutants is very preliminary, we believe that the statistical analysis is sound, and the conclusion — that we should attempt to distinguish stubborn (and hence possibly equivalent) mutants from trivial ones before assuming blanket equivalence, and reporting mutation score, along with the statistical significance used for evaluation — is not affected.

VIII. Conclusion

This paper used Tchebysheff's inequality to find a theoretical lower bound for the number of randomly sampled mutants needed to achieve a given accuracy in predicting the full mutation score. The paper also shows that the same framework can be used to provide a theoretical lower bound for the number of inputs to be sampled for predicting the equivalence of a mutant with a certain accuracy. Using this framework, we observe that mutation score can be approximated with high accuracy ($\pm 7\%$) for sample sizes as low as $1,000$ mutants. Empirical evaluations on a set of 158 Java projects with different sizes validate the result of our statistical analysis.

The relatively small sample sizes suggested by our statistical framework can assure practitioners that they can effectively approximate mutation score with only a *constant amount of computation* (for fixed test suite size) for even extremely large projects, a fundamental improvement over the monotonically increasing number of mutants needed by previous approaches.

Our findings have a few consequences worth exploring further. One promising avenue of recent research has been analysis of *higher order mutants* [71], [72], where multiple mutations are combined into a single mutant. Higher order mutation brings with it many benefits such as increasingly subtle faults, and reduced number of equivalent mutants [71], [73]. However, as Jia [71] explains, it has not been popular due to the combinatorial explosion making even second order mutants out of reach due to combinatorial explosion. Our results show that this combinatorial explosion can be resolved through constant sampling. Irrespective of the population size resulting from higher order mutation, sampling about $9,604$ mutants can provide a theoretical guarantee of 1% accuracy (in fact much better than 1% in practice).

Our results also suggest a simple way to evaluate nth order mutation. Assume that our mutation tool provides p first order one-to-one operators for mutation, and we would like to evaluate nth order mutation on a program of size N. This can produce $T = N \times \binom{p}{n}$ ordered n-tuples of mutants. We simply generate $9,604$ random numbers in the range $(1 \ldots T)$, and pick the corresponding n-tuples as the nth order mutants to be evaluated, which can provide an accurate value of the mutation score for evaluating all of T. This also opens up opportunities for further validations of the coupling effect, which has only been investigated up to the second order [32], [33] empirically.

It may be pointed out that as the programs grow large, the test suites grow large. Since the test suites need to be run as many times as there are mutants, there is still some growth in the runtime requirements of mutation testing. However, we note that the entire test suite does not need to be rerun to evaluate a single mutant. Rather we only need to pay attention to those tests that cover the mutant in question. For unit tests, the incremental costs of mutation analysis in adding new tests can be very minimal, providing some up-front investment in determining code coverage. In most cases (for large projects), we can limit the cost of mutation analysis to far lower than that of a full test suite run as only the relevant tests need to be run.

Mutation analysis aims to capture real fault patterns. By using mechanisms such as selective mutation, one loses out on capturing fault patterns similar to those in excluded mutants. Our advice to mutation tool implementers is to be generous in the operators implemented, and use statistics to your advantage. If systematic defects in a test suite (e.g. low coverage of a module) contribute strongly to a low mutation score, they are statistically highly likely to be reproduced in a random sample as well, because (as we have shown) even a constant sized random sample predicts overall score well.

Our message to researchers working in this field is to use the sample size indicated by theory to evaluate techniques using mutation rather than the full set of mutants (unless there is sufficient evidence that a smaller sample size suffices for the particular set of mutants), and to provide the confidence intervals on both mutants sampled, and also on the probable equivalence of mutants remaining, so that other researchers can estimate how much effort was put into eliminating equivalent mutants.

For practicing testers, we suggest that the empirical bound of $1,000$ for sample size is sufficient for a reliable estimate of mutation score, within a percentage. For developers looking for even faster turnaround and practical advice on sampling, in an extended online version of this paper (available on our web sites and as a technical report) we discuss statistical techniques, including Bayesian methods, for reducing sampling requirements further.

Our full data set is available for replication [53].

References

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

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

[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. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635929

[6] P. Doughty-White and M. Quick, "Information is beautiful : Codebases - millions of lines of code," http://www.informationisbeautiful.net/visualizations/million-lines-of-code/.

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

[8] 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.

[9] 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.

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

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

[12] W. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *Journal of Systems and Software*, vol. 31, no. 3, pp. 185 – 196, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0164121294000980

[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. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806863

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

[16] P. Tchebichef, "Des valeurs moyennes," *Journal de Mathématiques Pures et Appliquées*, vol. 2, no. 12, pp. 177–184, 1867.

[17] D. Baldwin and F. Sayward, "Heuristics for determining equivalence of program mutations." DTIC Document, Tech. Rep., 1979.

[18] L. Madeyski, W. Orzeszyna, R. Torkar, and M. JÃşzala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, January 2014.

[19] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982. [Online]. Available: http://dx.doi.org/10.1007/BF00625279

[20] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1997. [Online]. Available: http://doi.acm.org/10.1145/267580.267590

[21] R. A. Demillo, A. T. Acree, T. A. Budd, R. J. Lipton, and F. G. Sayward, "Metainduction and Progam Mutation: Realistic Software Validation (Mutation Analysis)," Georgia Institute of Technology, Tech. Rep., 1978.

[22] 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. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568225.2568265

[23] A. J. Offutt and W. M. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 1–26, 1996.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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. [Online]. Available: http://doi.acm.org/10.1145/229000.226313

[30] K. S. H. T. Wah, "A theoretical study of fault coupling," *Software Testing, Verification and Reliability*, vol. 10, no. 1, pp. 3–45, 2000. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1099-1689(200003)10:1<3::AID-STVR196>3.0.CO;2-P

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

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

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

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

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

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

[37] M. Sahinoglu and E. H. Spafford, "A bayes sequential statistical procedure for approving products in mutation-based software testing," in *IFIP Conference for Approving Software Products*, 1990, pp. 43–56.

[38] W. Hsu, M. Sahinoglu, and E. H. Spafford, "An experimental approach to statistical mutation-based testing," Software Engineering Research Center, Purdue University, Tech. Rep. SERC-TR-63-P, 1992.

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

[40] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *International Conference on Software Engineering*, 2015.

[41] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," *Software Testing, Verification and Reliability*, vol. 4, no. 3, pp. 131–154, 1994.

[42] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[43] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," in *Workshop on Formal Methods in the Development of Software, WS-FMDS*, 2012, pp. 1–8. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.86.1

[44] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[45] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann, "A theory of changes for higher-order languages: Incrementalizing λ-calculi by static differentiation," in *ACM SIGPLAN Conference*

on Programming Language Design and Implementation. New York, NY, USA: ACM, 2014, pp. 145–155. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594304

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

[47] J. L. Fleiss, B. Levin, and M. C. Paik, *Statistical methods for rates and proportions*. John Wiley & Sons, 2013.

[48] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *The Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

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

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

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

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

[53] R. Gopinath, "Replication data for: How Hard Does Mutation Analysis Have To Be, Anyway?" http://eecs.osuosl.org/rahul/issre15.

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

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

[56] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 246–256. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635921

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

[58] Python Software Foundation, "Python programming language," https://www.python.org/.

[59] A. Derezinska and K. Halas, "Experimental evaluation of mutation testing approaches to python programs," in *International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 156–164. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2014.24

[60] M. Teo, "Python mutant tester (pymutester)," https://pypi.python.org/pypi/pymutester/0.1.0.

[61] M. Stephens, "Mutation testing for python," https://pypi.python.org/pypi/mutant/0.1.

[62] 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.

[63] J. S. Vitter, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[64] Y. Cai, P. G. Giarrusso, T. Rendel, and K. Ostermann, "A theory of changes for higher-order languages: Incrementalizing u039b-calculi by static differentiation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, 2014, pp. 145–155.

[65] R. Gopinath, "Bytecode based mutation analysis for python," https://bitbucket.org/rgopinath/xmutant/.

[66] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard, "An evaluation of exhaustive testing for data structures," MIT Computer Science and Artificial Intelligence Laboratory Report MIT -LCS-TR-921, Tech. Rep., 2003.

[67] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775–782, 1996.

[68] J. Zhi and V. Garousi, "On adequacy of assertions in automated test suites: An empirical investigation," in *International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2013, pp. 382–391.

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

[70] A. Groce, "(quickly) testing the tester via path coverage," in *International Workshop on Dynamic Analysis*. ACM, 2009, pp. 22–28.

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

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

[73] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, Jan 2014.

IX. Appendix

A. Help for the tester

In this section, we outline the steps to be followed for determining test suite quality using mutants. We use the R statistical environment for our explanations.

For a practicing tester, the first question to be answered about any new test case is whether it adds value to the existing set of test cases. A secondary related question is whether the number of test cases is sufficient. Using the outlined procedures, answering these questions become a simple matter of using the statistical test for proportions.

We use the *Apache commons-math* code base, which has a true mutation score of 74.03%, to demonstrate the steps involved. The distribution of mutation scores obtained using a sample size of $1,000$ for this project is visualized in Figure 2. Say that we would like to know the approximate mutation score of the project quickly. To do that, we sample 100 mutants randomly out of the complete set of mutants. Let us assume that we had a result of 77 detected mutants. We use that in R to get the confidence intervals:

```
> prop.test(77, 100)
1-sample Proportions Test with continuity correction
data: 77 out of 100, nil probability 0.5
X-squared = 28.09, df = 1, p-value = 1.158e-07
alternative hypothesis: actual p is not equal to 0.5
95 percent confidence interval: 0.673059 0.845785
sample estimates: p = 0.77
```

Notice the 95% confidence interval, and the estimate, which is close to the true mutation score. Suppose we would like to verify whether we have crossed our target adequacy level of an 80% mutation score a little more accurately. To do that, we sample $1,000$ mutants, detecting 758 mutants.

```
> prop.test(758,1000,p=c(0.80))
1-sample Proportions Test with continuity correction
data: 758 out of 1000, nil probability c(0.8)
X-squared = 10.7641, df = 1, p-value = 0.001035
alternative hypothesis: actual p is not equal to 0.8
95 percent confidence interval: 0.7299831 0.7840052
sample estimates: p = 0.758
```

Notice the tighter confidence intervals, which do not include the 80% boundary, and also the significant p-value which suggests that we have not crossed the boundary yet.

1) Bayesian tools: Bayesian approaches are an alternative to the frequentist test of proportion above. They can use existing information regarding the mutation score (such as a previous run), and hence provide more accurate results.

The mutation score is the result of two random variables, the total number of mutants, and detected mutants. This can be modeled as a β distribution $\beta(s+1, f+1)$ with the parameter s representing number of successes, and f representing the number of failures. For a 95% credible interval[13], we note that only 5% of values lie outside this interval, of which half are less than $0.05/2$, and the other half are values greater than $1 - 0.05/2$. We pass in these quantiles, and also the mean at 0.5 to the $qbeta$ function, which returns the corresponding values.

```
> qbeta(c(0.025, 0.5, 0.975), 77 + 1, 23 + 1)
[1] 0.6781545 0.7664411 0.8414316
```

This suggests that our 95% credible interval for mutation score is between 67.81% and 84.14%, with predicted mutation score of 76.64%. As in the previous experiment, we would like to verify whether we have crossed the required score of 80% for a $1,000$-sample with 758 detection. We use the $pbeta$ function for that purpose.

```
> pbeta(0.8, 758 + 1, 242 + 1)
[1] 0.9994616
```

The relative frequency of detection is less than 0.8 with probability 0.9995. Let us say we have already tested our program before new changes were checked in, and we found that 580 mutants were detected for a sample size of $1,000$. Since we have not modified the program and tests radically, we have some confidence that our new score will be "close enough" to the old score to serve as a prior. In order to use this prior knowledge, we translate this score to a smaller sample that captures our intuition for how much weight we should give to the older result. Here, we translate our older score to a sample size of 100 with 58 mutants detected. We simply use this number in our formula and increase the samples evaluated.

```
> qbeta(c(0.025,0.5,0.975),58 + 758 +1, 42 + 242 +1)
[1] 0.7151307 0.7415254 0.7667981
```

By incorporating prior knowledge, we have improved our 95% credible interval to between 71.51% and 76.67%. This usage of prior knowledge is visualized in Figure 7.

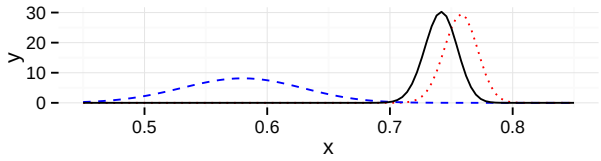

Fig. 7. Using prior knowledge to improve prediction. Blue (dashed) graph shows the prior knowledge, while the red (dotted) shows the result of current sampling, resulting in our prediction (black). Our uncertainty in prior is represented by its larger variance, hence lesser effect.

[13]Bayesian statistics uses "credible intervals" which are slightly different from "confidence intervals". A confidence interval is the likelihood that our interval contains the correct value (which is a constant), while a credible interval is the extent of uncertainty we have about the mean value (which according to Bayesian statistics, is a random variable).