How Hard Does Mutation Analysis Have To Be, Anyway?

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

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

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

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

Alex Groce
Oregon State University
agroce@gmail.com

ABSTRACT

Mutation analysis is considered the best method for measuring the adequacy of test suites. However, the number of test runs required to do a full mutation analysis grows faster than project size, which makes it an unrealistic metric for most real-world software projects, which often have more than a million lines of code. It is in projects of this size, however, that developers most need a method for evaluating the efficacy of a test suite. Another impediment to adoption of mutation analysis is the equivalent mutant problem, which makes mutation scores less informative.

Various strategies have been proposed to deal with the explosion of mutants, including forms of operator selection and x% stratified sampling based on operators and program elements. 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 two million LOC program usually requires analyzing well 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 being analyzed or the similarity between mutants. We show that a similar approach, of using a *constant sample of inputs* can estimate the degree of stubbornness in mutants remaining to a high degree of statistical confidence. We also provide a mutation analysis framework for Python that incorporates the analysis of stubbornness of mutants.

Categories and Subject Descriptors

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

General Terms

Measurement, Verification

Keywords

Test frameworks, theoretical analysis, empirical analysis, mutation operators

1. INTRODUCTION

Traditional mutation analysis [11, 30] 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. [29] 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.

An impediment to the wider adoption of mutation analysis in industry is its high computational cost. Performing mutation analysis on large programs requires analysis of an even larger number of mutants, which requires multiple runs of a test suite. Modern software often has a million or more lines of code [19], and it is impractical to evaluate all mutants when their number is a multiplier of such a large project size.

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 [43]. This approach is generally divided into selective and sampling approaches. Selective approaches attempt to avoid low utility mutants according to various definitions of utility. They compute the mutation score using only what are deemed to be high utility mutants [42, 48]. 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 [1, 8]. Wong et al. [57] utilized 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 combining both operator based and program element based stratified sampling [61, 62]. They found that stratified sampling using a combination of different strata was superior to stratified sampling of each 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 $O(n^{0.05\ldots0.25})$ where n is the size of the program for programs below 16KLOC [60].

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 any 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 can be accomplished in a fixed number of test runs.

We use Tchebysheff's inequality[2] to show that such a lower bound exists. 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 the above finding, we performed an extensive empirical study on 158 open source Java projects. Surprisingly, we find that sampling of just $1,000$ mutants results in an absolute error as low as $\pm 2\%$.

A secondary concern in mutation analysis is the prevalence of equivalent mutants [7,32]. These are mutants which are semantically identical to the original program, and in a general sense, their identification is undecidable [9].

Since determining mutation adequacy [64] 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% [15] up to 50% [40, 59]. Hence assuming a fixed percentage to be equivalent is not justifiable in practice. Human evaluation of equivalence is both exorbitantly expensive [59] and error prone, with fault rates up to 20% [1]. 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 [50, 61, 62]. While there have been various attempts [32] at providing heuristics for recognition of some equivalent mutants, none provides a bound on even the minimum stubbornness expected of remaining mutants which are classified as equivalent, which is important for a practicing tester who wishes to know whether a test suite has passed the required adequacy score.

We propose a simple extension to mutation analysis to resolve this dilemma, by treating the closeness of two functions as a statistical problem. That is, to identify whether a mutated function is semantically close to the original (with a given confidence) generate a fixed number of random inputs that cover their input domain, and verify that the functions produce the same output/return values. We show that the sample size suggested by our statistical framework is applicable for quantifying mutant stubbornness also, 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:

- We describe a statistical framework in Section 3 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 in Section 4 and show that the lower bound on the sample size needed to approximate mutation score 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 described in Section 5 that incorporates statistical determination of stubbornness in the remaining mutants, and describe the preliminary research into analysis of equivalence using statistical sampling.
- We provide easy to follow procedures in Section 7 for practicing testers to both estimate confidence in the mutation score from a given sample, and also incorporate prior test results to improve this prediction.

2. RELATED WORK

The idea of mutation analysis was first proposed by Lipton [30], and its main concepts were formalized by DeMillo et al. in the "Hints" [17] paper. The first implementation of mutation analysis was provided in the PhD thesis of Budd [10] in 1980.

Previous research in mutation analysis [8, 35, 44] suggests that it subsumes different coverage measures, including *statement*, *branch*, and *all-defs* dataflow coverage [8,35,44]. There is also some evidence that that the faults produced by mutation analysis are similar to real faults in terms of error trace produced [14] and the ease of detection [3, 4]. Recent research by Just et al. [29] 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 [17]. Evidence of the coupling effect comes from theoretical analysis by Wah [55, 56] and empirical studies by Offutt [37, 38].

Researchers have suggested several approaches to reducing the cost of mutation analysis, which were categorized as *do smarter*, *do faster*, and *do fewer* by Offutt et al. [43]. The *do smarter* approaches include space-time trade-offs, weak mutation analysis, and parallelization of mutation analysis, and *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 [8] and Acree [1] who showed that using just 10% of the mutants was sufficient to achieve 99% accuracy of prediction for the final mutation

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

[2]Nearly all values are close to mean [52].

score. This idea was further investigated by Mathur [34], Wong et al. [57,58], and Offutt et al. [42] using the Mothra [16] mutation operators for FORTRAN. Lu Zhang et al. [62] 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. [61] 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 both strata were used in conjunction performed best in predicting the final mutation score, and as few as 5% was sufficient for a correlation of 99% with the full mutation score. Hsu et al. [26] use "Binomial Sequential Probability Ratio Test" as a stopping rule for i.i.d random variables, with 299 mutants to be sampled for 1% tolerance within 99% confidence interval.

A number of studies also measures the redundancy among mutants. Ammann et al. [2] compared the behavior of each mutant under all tests and found a large number of redundant mutants. More recently, Papadakis et al. [45] utilized 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 has three major differences with above studies. First, while other studies suggest using a *fraction* of mutants for testing, in this paper, we study usefulness of selecting a *fixed number* of mutants to approximate the mutation score and the accuracy of the approximation. Second, most of above *do-fewer* approaches [61,62] base their validity on empirical studies. However, we offer a statistical foundation for our technique, in addition to empirical validation. Third, our proposed statistical framework does not require assumption of independence. This is important especially since other studies [2, 45] indicate that mutants are rarely independent, and largely similar with each other.

Researchers have also focused on identifying equivalent mutants, generally divided into prevention, and detection camps [45]. The prevention camp is concerned with reducing the incidence of equivalent mutants by identifying operators that produce low number of equivalent mutants, and using them [59]. The detection camp tries to detect the equivalent mutants by various static and dynamic properties of the mutant. These include efforts to identify them using compiler equivalence [7, 39, 45] dynamic analysis of constraint violations [36, 41], and coverage [49]. Recent research by Cai et al. [12] 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 their 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 abate 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.

3. 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 approximation of 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.

3.1 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 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 mutants are fewer than the mutants with a positive correlation). Thus the only assumptions that we need for our analysis are:
Assumptions:

- Mutants are largely positively correlated with others.
- 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 estimate of 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 of $E(M_n)$ is given by $\frac{1}{n}E(D_n)$. The mean of the sum of random variables does not depend on the 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

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

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

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

and, we want to restrict this to be less than δ. That is,

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

Since we know that assuming independence does not change the inequality because covariance is positive (from 1), we assume from here that $V(M_n)$ is independent. We can replace it by

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

Now consider D_n. If we consider the set of mutants that we are sampling, each of them would 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. (Notice 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 here to note is that, that 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 we have here, where we are trying to estimate the mutation score for a particular test suite – the mutants will be either detected or not by it, and the probability of picking a mutant that will be detected is determined only by the number of detected mutants in the total population, which is m.)

Since 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

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

Inequality (2) 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 follow 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 large enough n, binomial distribution approximates a normal distribution [20]. 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 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 $\delta = 0.05$, $N \geq 9604$, which is lower sample size than that predicted using Tchebysheff's inequality.

An important aspect of the result is that our results hold *even if the mutants are not independent.* In fact, the more similar to each other they are, the smaller the number of samples needed to estimate the true mutation score. We can use an intuition to illustrate this fact. 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 investigation suggests that a much smaller sample size than what is predicted by the theory is sufficient to estimate mutation score to a high degree of accuracy.

We validate our lower bound empirically in Section 4.

3.2 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 [3]. Our goal is to evaluate whether the mutant is equivalent to the program.

Let n be the input domain [4]. Consider a function F that takes no input, but compares the functions P and Q for some value given by

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

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. Notice that 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.

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. Note that empirically mutants that are extremely resistant towards detection at the function level tend to be equivalent mutants

4. EVALUATION OF MUTATION SAMPLING

[3] 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

[4] 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

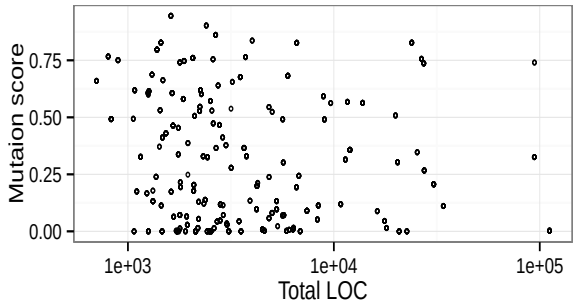

Figure 1: The distribution of size of project in LOC and mutation score. It shows that we have a representative sample of projects and mutation scores.

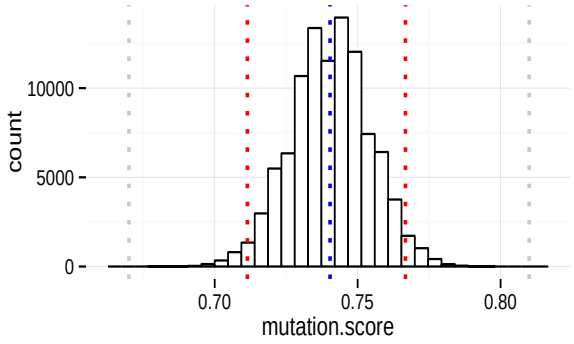

Figure 2: The distribution of mean mutation score when sample size is 1000 for Apache commons-math. The central blue line shows true mean at 74.03, and the red lines mark 95% confidence intervals. Similarly the gray lines show the theoretical bounds for 95% confidence intervals. This shows that the sample variation is well within theoretical bounds.

4.1 Methodology

For our empirical experiments, 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 [5] of Java projects from Github [21] and similarly from the Apache Software Foundation [5]. All projects selected used the popular maven [6] build system. This gave us $1,800$ projects. From these, the projects that did not compile (for reasons such as unavailable dependencies, or compilation errors due to syntax or bad configurations) were removed first. Next, the projects that did not pass their test suites were eliminated since mutation analysis requires a passing test suite. Finally, we only chose projects that were non-trivial, had a test suite (some of these test suites actually had 0 mutation score – we opted to keep these for representativeness of the samle, but the results remain same even if they are removed), and had at least $1,000$ mutants. This follows the methodology we used in our previous work [24], 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.

We used PIT [13]for mutation analysis (used in multiple studies [24, 27]), extended to provide the complete matrix of test failures between mutants and test cases.

4.2 Analysis

Our empirical analysis was done in two parts. First we looked in detail a moderately large project with high coverage (90%), to see if our predictions hold 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 0.7404. We also estimated the 2.5% and 97.5% quantiles (95% of values lie between these quantiles). These were found to be 0.713, and 0.767 respectively, plotted as the red lines in the figure. As we expected, this is well within our theoretical prediction of 0.67 and 0.81 for $1,000$ samples, which 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*[6] 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 can get an accuracy of at least 7.2% with 95% probability. 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 while calculating the bound, and secondly, the theoretical bound was predicted by assuming that the mutation score would result in the maximum variation at $m = 0.5$.

In order to visualize how the tolerances change when the sample size is increased, we plotted the summary of variation of 100 repetitions for each of the 158 projects, with sample sizes corresponding to $(2^3, 2^4 ... 2^{16})$. This is given in Figure 4. In the boxplots the boxes represent 25% and 75% quantiles, while the ends of lines represent 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 mu-

[5]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 testsuites, and hence should not confound our results.

[6]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

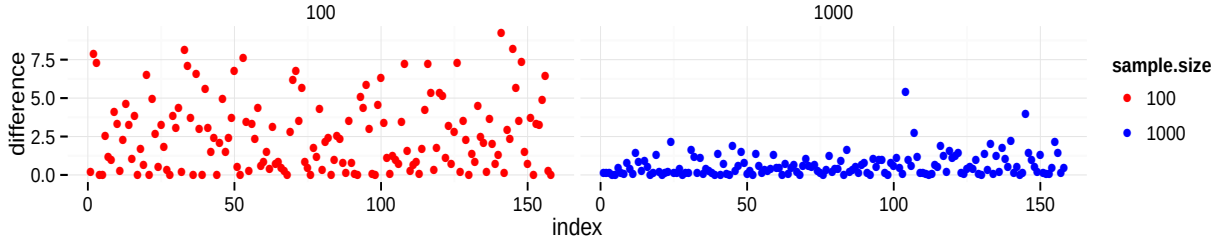

Figure 3: The difference from true mutation score when sample size is 100, and when sample size is 1000 in %. The projects are ordered by the total number of mutants, which shows that accuracy has no relation to project size.

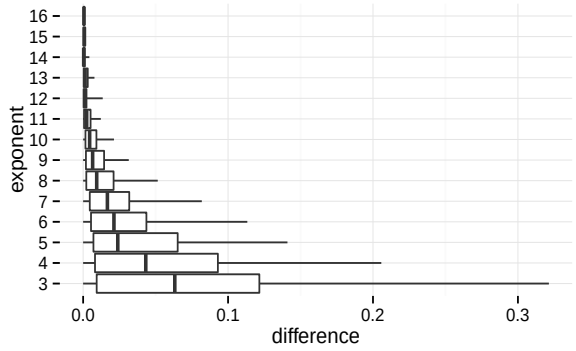

Figure 4: This graph shows the progressive improvement of accuracy (x) as sample sizes increase as powers of 2 (2^y) for 158 projects. The boxes in boxplots represent 0.25 and 0.75 quantiles, and the ends of black lines represent quantiles at 0.025 and 0.975 respectively. This figure shows how precision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

tants vs an x% mutant sample where the x% was given by decreasing fractions of 2^{-i} – i.e $(\frac{1}{2}, \frac{1}{4} \ldots \frac{1}{64})$ — 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 reported between the original mutation score and the mutation score reported by the sample, and marked all the observations that fell below 1% as bad (red). This is following the limit of 99% frequently used by researchers [61] 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.*

5. EVALUATION OF STUBBORN MUTANTS

For our evaluation of the sample size required for 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 developing our own mutation analysis tool and framework for evaluating stubbornness in live mutants.

5.1 Mutation Framework

Python [46] is one of the common languages 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. Secondly, recent results suggest a large number of source modifications result in expressions which can result in identical compiled code. While there exist mutation analysis programs for Python [18], they were either source or AST based, were either for Python 3, which excluded well tested applications in Python 2, or they were abandoned prototypes [51, 53].

To avoid the equivalence of compiled expressions, and also to avoid issues with invalid mutants, we started by extending mutant [51] the byte-code mutation analysis tool. We implemented the traditional operators [47, 48] such as *modify numerical constants*, *negate jumps*, *replace arithmetic* and *binary operators*. We also make use of optimization techniques such as parallel execution of mutants, and filtering of coverage data to ensure that only relevant mutants are generated.

For the evaluation of stubborn mutants, we require that the 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 [7]. 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 does 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 are 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 [54] which allows us to ensure that we need to keep only the sample size number of input items in memory.

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

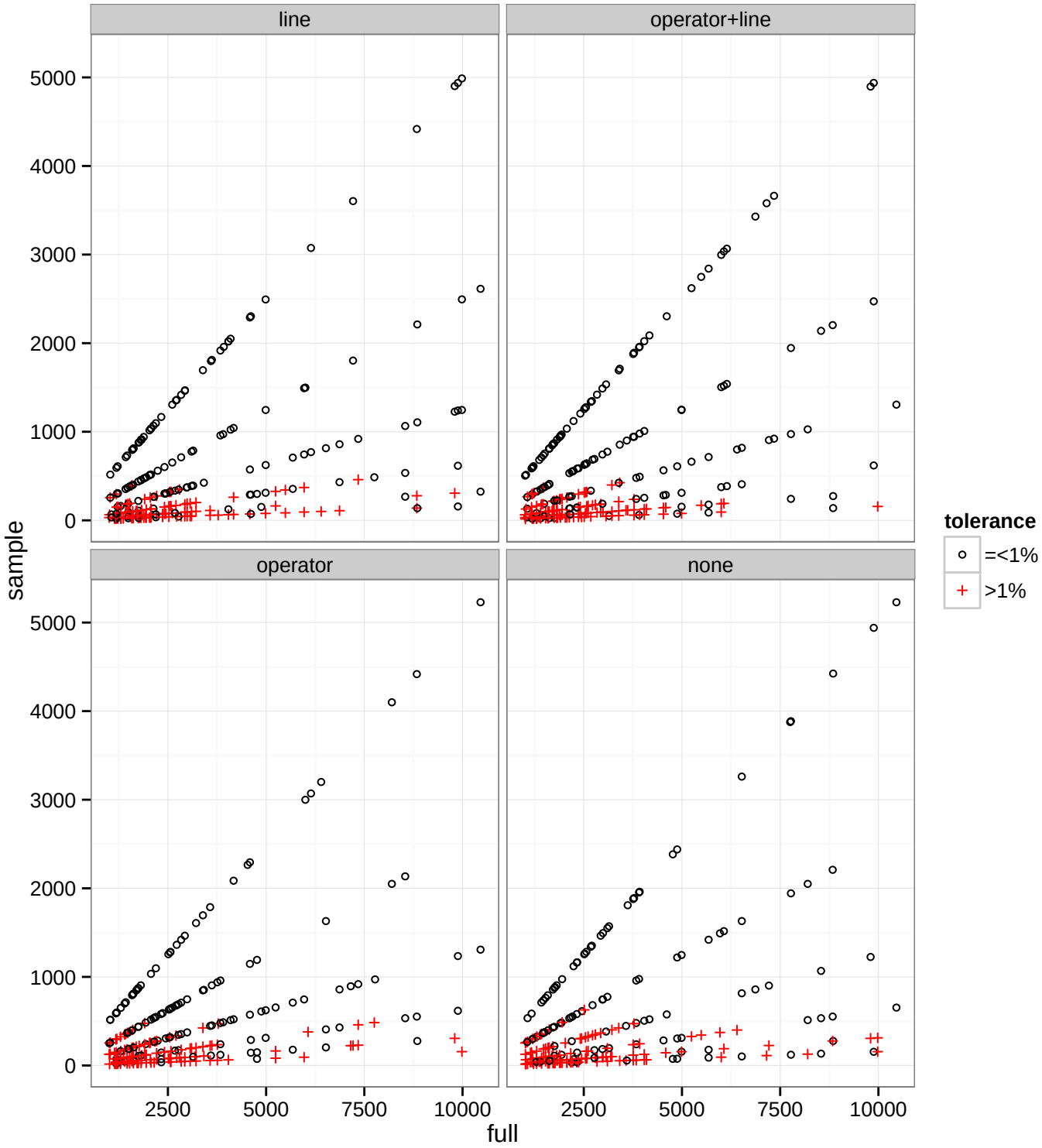

Figure 5: This graph plots the full mutation score against the sample sizes suggested by different stratified sampling strategies (We show only the combinations of line and operator strata. Combinations of method and class with operator are similar). The red color (solid) indicates places where measured mean mutation score (from 100 repetitions) was different from the true mutation score by at least 1%. The distinct lines 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.

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 function at hand.

Very preliminary research using our framework, XMutant [22] – available as an open source project, along with 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 1000-sampling, which were also found to be equivalent by human analysis. The 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 is in order: If we are using mutation analysis to evaluate adequate unit tests, then mutants that are trivially detected during sampling process show serious inadequacy in tests. Similarly, for evaluating adequacy of unit tests, while mutants that escape detection during sampling can not 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 a 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, or simply show major problems with the suite. In that case, very stubborn mutants again may be thrown out with some statistical guarantee that they are at least hard to kill, which previous work in the field did not do.

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

Notice that, we are not claiming that random sampling is the best way to identify these equivalents. For example, any patterns in input such as one would expect from the "Small Scope Hypothesis" [33] can aid detection tremendously. 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 [63].

6. DISCUSSION

In this section, we discuss the results of empirical validation of the statistical framework presented in Section 3. 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.

6.1 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%.

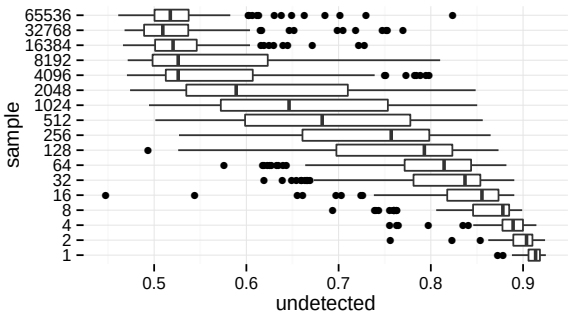

Figure 6: The reduction in undetected mutants for *timsort* as sample size increases. Y axis represents the sample size and X axis the ratio of undetected mutants found. Experiments were repeated 100 times for each sample size.

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

6.2 1,000-sampling at large

Comparing the results of the statistical lower bound of 1,000-sampling and empirical evaluation, while our statistical framework suggests a pessimistic expected error rate of ±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 implication of Observation 2 is twofold. First, it suggests that the number of required mutants to accurately approximate 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 low as 1,000 mutants) is sufficient for estimating 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-based mutation sampling, suggested by [61], 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 score with absolute error smaller than 1% and red symbols show the absolute error larger than 1%. This figure shows that all element based mutation samples of sizes 1,000 could achieve an accuracy of 1%. But for sample sizes smaller than 1,000, in many instances the error is higher. This suggests a limitation of the applicability of element-based mutation sampling. 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.

7. HELP FOR THE PRACTICING 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*, which has a true mutant 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, null probability 0.5
X-squared = 28.09, df = 1, p-value = 1.158e-07
alternative hypothesis: true 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, null probability c(0.8)
X-squared = 10.7641, df = 1, p-value = 0.001035
alternative hypothesis: true 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.

7.0.1 Bayesian tools

Bayesian approaches are an alternative to the steps we have outlined previously. 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 those detected. 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. So to get a 95% credible interval[8], we note that only 5% of values lie outside this interval, of which half are values 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

[8]Bayesian statistics use "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 credible interval is the extent of uncertainty we have about the mean value (which according to Bayesian statistics, is a random variable).

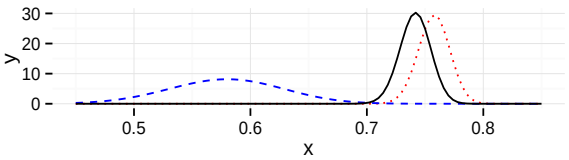

Figure 7: Using prior knowledge to improve prediction. Blue (dashed) graph shows the prior knowledge, while the red (dotted) shows the result of current sampling, resulting in our prediction (black). Our uncertainty in prior is represented by its larger variance, hence lesser effect.

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(.025, .5, .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.

8. 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.

8.1 Threats to theoretical analysis

Our analysis relies on two assumptions, that the number of mutants involved are sufficiently large, necessitating a reduction by sampling, and secondly that either the detection of mutants are 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 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.

8.2 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 Github repository using the `Maven` build system. We used all projects that we could retrive 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 could skew the projects returned by Github, either due to any skew of Github projects themselves, or due to any kind of prioritization by Github in returning the results.

Projects with 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 the real world projects. Unfortunately given that a large majority of these projects are in the process of development, with many small personal projects, some of them have zero mutation score (even with a test suite), and in general, low 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*, a large project of 94KLOC 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 say (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 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 to validity 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 empirical analysis of stubborn mutants are 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 reporte mutation score, along with the statistical significance used for evaluation – is not affected.

9. CONCLUSION

This paper used Tchebysheff's inequality to find a theoretical lower bound for the number of randomly sampled mutants needed to achieve a certain 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 compute the mutation score with a relatively high accuracy with acceptable computation costs (depending to test suite size).

Our findings have a few consequences worth exploring further. One of the promising avenues of research in recent years have been the higher order mutants [25, 28], where multiple mutations are combined together for a single mutant. Higher order mutation brings with it many benefits such as increased subtlety of faults, and reduced number of equivalent mutants [28, 31]. However, as Jia [28] explains, it has not been popular due to the combinatorial explosion making even the second order mutants our of reach of traditional mutation analysis. However, with our finding, the problem of combinatorial explosion vanishes. Irrespective of the population size, sampling about $50,000$ 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 $50,000$ 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 until the second order [37, 38] empirically.

Mutation analysis derives its legitimacy partly from being able to capture the fault patterns. By using mechanisms such as selective mutation, one loses out on capturing fault patterns similar to those by excluded mutants. Our advice to the tool implementers is to be generous in the operators you implement, and use statistics to your advantage.

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 1000 for sample size is sufficient for a reliable estimate of mutation score, within a decimal point. For developers

looking for a fast turnaround, use the Bayesian techniques to reduce the sampling requirements even further.

Our full data set is available for replication [23].

10. REFERENCES

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

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

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

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

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

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

[7] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical report, DTIC Document, 1979.

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

[9] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

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

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

[12] 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*, pages 145–155, New York, NY, USA, 2014. ACM.

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

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

[15] 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). Technical report, Georgia Institute of Technology, 1978.

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

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

[18] A. Derezinska and K. Halas. Experimental evaluation of mutation testing approaches to python programs. In *International Conference on Software Testing, Verification and Validation Workshops*, pages 156–164, 2014.

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

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

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

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

[23] R. Gopinath. Replication data for: How Hard Does Mutation Analysis Have To Be, Anyway? http://eecs.osuosl.org/rahul/fse2015.

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

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

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

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

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

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

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

[31] 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*, 40(1):23–42, Jan 2014.

[32] 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*, 40(1):23–42, January 2014.

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

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

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

[36] S. Nica and F. Wotawa. Using constraints for equivalent mutant detection. In *Workshop on Formal Methods in the Development of Software, WS-FMDS*, pages 1–8, 2012.

[37] A. J. Offutt. The Coupling Effect : Fact or Fiction? *ACM SIGSOFT Software Engineering Notes*, 14(8):131–140, Nov. 1989.

[38] A. J. Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[39] A. J. Offutt and W. M. Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[40] A. J. Offutt and W. M. Craft. Using Compiler Optimization Techniques to Detect Equivalent Mutants. *Software Testing, Verification and Reliability*, 4(3):1–26, 1996.

[41] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 1997.

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

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

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

[45] 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.

[46] Python Software Foundation. Python programming language. https://www.python.org/.

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

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

[49] D. Schuler and A. Zeller. Covering and uncovering equivalent mutants. *Software Testing, Verification and Reliability*, 23(5):353–374, 2013.

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

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

[52] P. Tchebichef. Des valeurs moyennes. *Journal de mathÃl'matiques pures et appliquÃl'es*, 2(12):177–184, 1867.

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

[54] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[55] K. S. H. T. Wah. A theoretical study of fault coupling. *Software Testing, Verification and Reliability*, 10(1):3–45, 2000.

[56] K. S. H. T. Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2):119–161, 2003.

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

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

[59] 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*, pages 919–930, 2014.

[60] 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.

[61] 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.

[62] 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*, pages 435–444, New York, NY, USA, 2010. ACM.

[63] 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*, pages 382–391. IEEE, 2013.

[64] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, Dec. 1997.