An Information Theoretic Notion of Software Testability[☆]

Krishna Patel[a], Robert M Hierons[a], David Clark[b]

[a]*Department of Computer Science, The University of Sheffield, Sheffield, S1 4DP, UK*
[b]*Department of Computer Science, University College London, WC1E 6BT, UK*

Abstract

Context: In software testing, Failed Error Propagation (FEP) is the situation in which a faulty program state occurs during the execution of the system under test (SUT) but this does not lead to incorrect output. It is known that FEP can adversely affect software testing and this has resulted in associated information theoretic measures.

Objective: To devise measures that can be used to assess the *testability* of the SUT. By testability, we mean how likely it is that a faulty program state, that occurs during testing, will lead to incorrect output. Previous work has considered a single program point rather than an entire program.

Method: New, more fine-grained, measures were devised. Experiments were used to evaluate these and the previously defined measures (Squeeziness and Normalised Squeeziness). The experiments assessed how well these measures correlated with an estimate of the probability of FEP occurring during testing. Mutants were used to estimate this probability.

Results: A strong rank correlation was found between several of the measures and the probability of FEP. Importantly, this included the Normalised Squeeziness of the whole SUT, which is simpler to compute, or estimate, than most of the other measures considered. Additional experiments found that the measures were relatively insensitive to the choice of mutants and also test suite.

Conclusions: There is scope to use information theoretic measures to estimate how

[☆]This document is a result of the EPSRC funded research project EP/P006116/2 InfoTestSS: Information theory and Test Suite Selection.

Email addresses: `krishna.patel@sheffield.ac.uk` (Krishna Patel), `r.hierons@sheffield.ac.uk` (Robert M Hierons), `david.clark@ucl.ac.uk` (David Clark)

prone an SUT is to FEP. As a result, there is potential to use such measures to prioritise testing or estimate how much testing an SUT might require.

Keywords: software testability, information theory, failed error propagation

1. Introduction

Software testing involves executing the system under test (SUT) on a number of test inputs and determining whether the resultant outputs are correct (consistent with requirements). It has been observed that if the SUT has a fault then this fault will only be found in testing with a given input if three conditions are satisfied: the fault is executed (execution); the execution of the fault leads to an incorrect program state (infection); and the incorrect program state propagates to an incorrect output (propagation). These three conditions are reflected in the PIE framework [1], which was recently generalised to the RIPR framework [2].

It is possible that a fault is executed but this does not lead to an incorrect output, with this situation being called *coincidental correctness*. The results of studies suggest that coincidental correctness can have a major impact on testing [3, 4]. If one considers the PIE framework, it is clear that coincidental correctness can occur through a failure of either infection or propagation. Failed Error Propagation (FEP) represents the second of these: a faulty state that occurs during the execution of an SUT that does not propagate to the output.

The potential for FEP to reduce the effectiveness of testing led to the definition of a measure, Squeeziness (Sq). Squeeziness measures the information (entropy) loss, between two random variables in the states occurring at two program points, that results from the execution of a program [5, 6]. It was found that the probability of FEP, for a fault that infects the state at a program point p of program P, correlates strongly with the Squeeziness of the program formed by starting the execution of P at p [6]. Thus, Sq can be used to reason about the probability of FEP if we are interested in a particular program point and there are scenarios in which this is valuable. For example, there is potential to use Sq in order to enhance notions of coverage in order to take into account how difficult it is to detect a fault at a given program point [6].

More recently, it has been observed that Sq does not provide an appropriate basis for comparing two (sub-)programs that have different input domains [7]. This led to the definition of a new measure, called Normalised Squeeziness (NSq), which measures the *proportion* of entropy lost in program execution. Simulations demonstrated that NSq is more effective than Sq when comparing programs (functions) with different input domains [7] but NSq has not hitherto been evaluated on real programs.

Although the above concepts and results can be used to reason about particular program points, they do not tell us how testable a program P is. Here, by testability we mean the probability of FEP occurring if an incorrect program state occurs. If we can find measures that estimate such a notion of testability then these measures might be used as the basis for estimating either the effectiveness of testing or the expected cost of testing. The aim of the research reported in this paper was to achieve exactly this: to propose and evaluate information theoretic measures that might be used to estimate this form of program testability. For each of Sq and NSq we considered three variants:

1. The Sq or NSq of the entire program P. Note that previous work has looked at the Sq or NSq of part of a program; that following a fault.
2. The average value over all program points. In order to compute this, for every program point p of a program P one forms a sub-program P_p representing the execution of P starting at p and computes the Sq or NSq of P_p. The mean is then taken over all program points of P.
3. A weighted version of the above, where the weighting for an internal state σ, at program point p, is based on the proportion of test inputs that lead to state σ at p.

The first of the above, using Sq or NSq, provides the measures that should be simplest to compute or estimate. Specifically, the other two sets of measures require the Sq or NSq to be computed n times for a program with n program points, increasing the computation cost by a factor of n. However, it was unclear whether the Sq or NSq of a program are sufficiently fine-grained to form the basis of estimates of testability. In particular, these measures are for the program as a whole but FEP is caused by only the part of the program that is after the fault executed. Additionally, fine-grained alternatives can account for the fact that different program points might vary in their contribution to information loss. This observation motivated our interest in the more fine-grained alternatives.

We evaluated these measures through experiments that used 18 C programs as subjects. In order to estimate the probability of FEP for a program P, we generated a set M_P of mutants of P that included at most one mutant for each program point. We then tested these mutants with a randomly generated set of test inputs, computing the proportion of inputs for which FEP occurred. We determined the rank correlation between this estimate of the probability of FEP and the proposed measures. Interestingly, the Normalised Squeeziness of a program strongly correlated with the probability of FEP (rank correlation of just over 0.77). This was only beaten by one measure, Weighted Average Normalised Squeeziness, and the differences were relatively small (a rank correlation of just over 0.78 for Weighted Average Normalised Squeeziness). These results are promising since, not only were strong correlations found, but these were found for a measure (NSq) that is relatively easy to compute or estimate. We also found that approaches based on NSq tended to outperform the corresponding approaches that use Sq, confirming results previously observed in simulations of programs [7].

We used mutants in the experiments because, for each program P and program point p, we wanted a version of P in which there is a fault at p. Although there are repositories that contain faulty versions of programs, we are not aware of any repositories that contain a faulty version for each program point p. We used at most one mutant for each program point to aid scalability. Several choices might have affected the results and so we carried out additional experiments in order to investigate these. First, we performed experiments that explored the effect of using different mutations at a program point p. It was found that the choice of mutant at program point p typically had only a relatively small effect on the Sq and NSq values associated with p, providing some confidence that the choice of mutant is unlikely to have significantly affected the results. For each program, we also generated 30 different test suites and estimated each measure using these, leading to 30 different values for each measure. The results showed only small variations in the values of the measures. This suggests that the results of the experiments are unlikely to have been affected by the choice of test suite. Importantly, the results also suggest that there is scope to use sampling to estimate the values of the measures and this should help associated techniques scale.

This paper makes the following main contributions:

1. Four new information theoretic measures: Average Squeeziness, Average Normalised Squeeziness, Weighted Average Squeeziness, and Weighted Average Normalised Squeeziness.
2. The first evaluation of Normalised Squeeziness on real programs.
3. Empirical evidence that the Normalised Squeeziness of a program strongly correlates with its testability.
4. Empirical evidence that demonstrates that the measures used are relatively insensitive to the choice of test suite and mutants used.

This paper is structured as follows. Section 2 provides background information and defines the measures used in the paper. Section 3 outlines the design of the experiments. Section 4 then presents the results of the experiments, while Section 5 discusses factors that might affect the validity of our results. Section 6 reviews related work. Finally, Section 7 summarises the main conclusions.

2. Background and Measures Used

This section describes the measures used. We start by saying what we mean by Average Failed Error Propagation, which is our estimate of the true testability of an SUT. We then describe six alternative entropy loss measures that are potential predictors of Average Failed Error Propagation. Two of these are measures that have previously been described (Squeeziness and Normalised Squeeziness) but were only evaluated in the context in which we have a known fault location and we consider the sub-program formed by starting the execution at this program point. We then define what we mean by Average Squeeziness and Average Normalised Squeeziness. Finally, we introduce weighted versions of Average Squeeziness and Average Normalised Squeeziness.

Note that all of these measures are defined in terms of a probability distribution on the input to the program, or sub-program. In principle, the most useful distribution is the usage distribution: the one collected through use of the program after deployment.

Failing that, we observe the Maximum Entropy Principle used in statistics: when the distribution is unknown, use what you do know to synthesise the distribution with maximal entropy consistent with your knowledge. When nothing is known, this will be a uniform distribution. Synthesising a distribution with maximal entropy has the advantage of being maximally explorative. The measures defined below thus assume that the input domain of the program is uniformly distributed, but make no assumptions about the distribution of the output domain. In what follows the term support of a random variable is the set of events or outcomes that have a non-zero probability while the notion of a random variable's entropy being conditioned on another describes the conditional entropy as defined in Definition 3.

2.1. Average Probability of Failed Error Propagation

Recall that Failed Error Propagation occurs when a corrupted program state manifests in the program (infection occurs), but does not propagate to an output. The proportion of faulty versions of a program in which corrupted states manifest after the execution of the fault, but produce the correct outputs, is a natural measure of the probability of encountering Failed Error Propagation in the program. The first measure we present is the implementation of such a measure, and is called Average Failed Error Propagation. The remainder of this section describes Average Failed Error Propagation.

The actual prevalence of FEP will depend on the specification (or a correct version of the program), the nature of any faults, and also the test inputs used. We therefore assume that there is a program P, a set of test inputs $T = \{tc_1, tc_2, ..., tc_n\}$ for P, and also that $\mathcal{M} = \{M_1, M_2, ..., M_m\}$ is a set of variants of P that represent possible faults. If there is a fault model, that describes likely faults, then this could be used as the basis for $\mathcal{M}$ but otherwise one could use standard mutation operators.

The execution of a program leads to an execution trace, which is the sequence of internal states that occur. Weak mutation compares the execution traces produced when a program P and a mutant M_j are executed using a test input tc_i. Let $P(tc_i)$ denote the execution trace produced when P is executed with tc_i and, for $M_j \in \mathcal{M}$, let $M_j(tc_i)$ be the execution trace produced when M_j is executed with tc_i. Then, M_j

is said to have been *killed by T under weak mutation* if there is some tc_i in T such that $P(tc_i) \neq M_j(tc_i)$ [6].

Strong mutation testing differs from weak mutation testing through only considering the program output instead of the entire execution trace. Given a test input tc_i, let $P(tc_i)|_O$ denote the output produced when P is executed with tc_i and, for $M_j \in \mathcal{M}$, let $M_j(tc_i)|_O$ be the output produced when M_j is executed with tc_i. Then, M_j is said to have been *killed by T under strong mutation* if there is some tc_i in T such that $P(tc_i)|_O \neq M_j(tc_i)|_O$. Clearly, if a mutant is killed under strong mutation then it is also killed under weak mutation.

Failed Error Propagation (FEP) occurs when there is an incorrect/unexpected program state during testing but this does not propagate to the output. Clark and Hierons [5] defined a collision as a situation in which two different states are mapped on to the same state by a function, and observed that collisions are necessary for FEP. Androutsopoulos et al. [6] recognised that scenarios in which mutants are killed by Weak Mutation Testing, but not Strong Mutation Testing, occur because of collisions, and represent all instances of FEP. They proposed a measure called Average Failed Error Propagation, which measures the frequency of these scenarios, and by implication, FEP.

Let M_j^s and M_j^w denote the set of test inputs from T that kill mutant M_j under strong mutation testing and weak mutation testing respectively and assume that M_j^w is non-zero[1]. Recall that $\mathcal{M}$ contains m mutants.

Definition 1. *Average Probability of Failed Error Propagation.*

$$AVGFEP = \frac{\sum_{j=1}^{m} \frac{|M_j^w| - |M_j^s|}{|M_j^w|}}{m}$$

Even though Average Failed Error Propagation is a natural measure of Failed Error Propagation, its computation can be extremely resource intensive. To that end, it is desirable to have alternative measures that are less resource intensive. The efficacy

[1]If a mutant is never killed under weak mutation then the execution of this mutant tells us nothing about FEP.

of alternative measures can be demonstrated by the extent to which they are correlated with Average Failed Error Propagation. Sections 2.2 to 2.3 describe alternative measures evaluated in this paper.

2.2. Squeeziness and Normalised Squeeziness

Information Theory is a branch of mathematics that is concerned with quantifying and reasoning about information. Entropy [8] is the most fundamental building block in information theory; it is a measure of the amount of uncertainty in a discrete random variable, and is often equated to the information content of a variable. Let us suppose that X is a random variable and for $x \in X$ we use $p(x)$ to denote the probability associated with x. Then the entropy of X is defined as follows.

Definition 2. *Entropy of a random variable.*

$$\mathcal{H}(X) = -\sum_{x \in X} p(x) log_2 p(x)$$

Conditional entropy is defined in terms of two random variables, say A and B, and can measure, for example, how much entropy is left in A once you know the entropy of B. This is closely related to the concept of conditional probability. One way of thinking about it is to measure the entropy of the joint random variable, $\langle A, B \rangle$, and subtract the entropy in B, so

Definition 3. *Conditional Entropy of a pair of random variables.*

$$\mathcal{H}(A \mid B) = \mathcal{H}(A, B) - \mathcal{H}(B)$$

Shannon defined conditional entropy from first principles but in our definition above we instead exploited the Chain rule [8].

We are interested in the entropy lost when a program executes and how this can be used to predict the failure of error propagation in the program. Let P denote a program. P can accept an input, x, such that $x \in ID$, where ID is the input domain of P, and produce an output, y, in response to x, such that $y \in OD$, where OD is the output domain of P. ID and OD can be modelled as discrete random variables, I and O respectively. Assuming the program is deterministic, all the entropy in the program during execution comes from the probability distribution on the inputs. Being deterministic, the outputs are then a function of the inputs and the entropy of I conditioned on O measures the Squeeziness of the program, i.e. the entropy lost through execution across all inputs.

Squeeziness is a measure of information loss [5, 6] and can be calculated as follows:

Definition 4. *Squeeziness.*

$$Sq(I, O) = \mathcal{H}(I) - \mathcal{H}(O)$$

Let $Dice(D1, D2)$ be a function that accepts two six-sided dice rolls as input, and produces the product of $D1$ and $D2$ as output. This function will be used as an example to illustrate the computation of Squeeziness.

The Input Domain of this function can be modelled as a discrete random variable, I. In particular, I is the set of unique inputs to the function i.e. $\{(x, y)|1 \leq x, y \leq 6\}$ and a function that maps each unique input to its probability of occurrence. Assume each input has an equal probability of being selected and there are 36 inputs in total, each input is associated with $1/36$.

The Entropy of I can be calculated using using the $\mathcal{H}(X)$ equation that was defined above. More specifically, $1/36 \times log_2(1/36)$ would be computed 36 times, the results of these 36 computations would then be summed, and the result of this summation would be negated. The Entropy of I is $\mathcal{H}(I) = 5.17$.

The Output Domain of this function can also be modelled as a discrete random variable, O. More specifically, O is the set of unique outputs that can be produced by the function i.e. $\{1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 30, 36\}$ and a function that maps each unique output to its probability of occurrence. The probability of a particular output being selected is based on the probability of an input that can produce this output being selected. For example, there are only $3/36$ inputs ($(1, 4)$, $(2, 2)$, $(4, 1)$) that can lead to output 4, and so output 4 would be associated with $3/36$. The outputs are associated with the following probabilities: $1 = 1/36$, $2 = 2/36$, $3 = 2/36$, $4 = 3/36$, $5 = 2/36$, $6 = 4/36$, $8 = 2/36$, $9 = 1/36$, $10 = 2/36$, $12 = 4/36$, $15 = 2/36$, $16 = 1/36$, $18 = 2/36$, $20 = 2/36$, $24 = 2/36$, $25 = 1/36$, $30 = 2/36$, and $36 = 1/36$.

The Entropy of O can be calculated using the $\mathcal{H}(X)$ equation that was defined above. More specifically, $1/36 \times log_2(1/36)$ would be computed 5 times, $2/36 \times log_2(2/36)$ would be computed 10 times, $3/36 \times log_2(3/36)$ would be computed once, and $4/36 \times log_2(4/36)$ would be computed twice. The results of these 18 computations would then be summed, and the results of this summation would be negated. The Entropy of O is $\mathcal{H}(O) = 4.04$.

As the program is deterministic, Squeeziness can be calculated by using the $Sq(I, O)$ equation that was defined above. More specifically, $Sq(I, O) = 5.17 - 4.04$, or in other words, $Sq(I, O) = 1.13$.

Since Squeeziness depends on both the function (program) and the entropy of the inputs it is not so useful for comparing programs with different initial entropies. By normalising the entropy loss as relative to the initial entropy, we can enjoy a normalised scale of interval $[0, 1]$ for comparing programs with different input domain sizes [7]. Note, however, that if I contains only one value or no values then $\mathcal{H}(I) = 0$, but in the limit as $\mathcal{H}(I) \rightarrow 0$, $(\mathcal{H}(I) - \mathcal{H}(O))/\mathcal{H}(I) \rightarrow 0$.

This leads to the following new definition of Normalised Squeeziness used in this paper, which extends the previous definition to work when $\mathcal{H}(I) = 0$. Compare with earlier definition [7].

Definition 5. *Normalised Squeeziness.*

$$NSq(I, O) = \begin{cases} 0 & \textit{if } \mathcal{H}(I) = 0; \\ \dfrac{Sq(I, O)}{\mathcal{H}(I)} & \textit{otherwise}. \end{cases}$$

We now introduce some new measures based on Squeeziness and Normalised Squeeziness.

2.3. Average Squeeziness and Average Normalised Squeeziness

A program is a sequence of n program statements. In this paper, when we refer to a program, P, we are referring to such a sequence i.e. $P = \langle S_1^P, S_2^P, \ldots, S_n^P \rangle$.

A mutant, M_a, is another sequence of program statements, $M_a = \langle S_1^{M_a}, S_2^{M_a}, \ldots, S_n^{M_a} \rangle$, such that the difference between M_a and P can be isolated to a single program statement [9]; this program statement is referred to as the *mutation point*.

As before, let $\mathcal{M} = \{M_1, M_2, ..., M_m\}$ be a set of m mutants of P. In the rest of this section, we use $\mathcal{M}_{\mathcal{P}}(M_j)$ to denote the mutation point of mutant $M_j \in \mathcal{M}$. Further, we use $\mathcal{PP}(M_j)$ to denote the (post) program formed by starting the execution of M_j at the mutation point $\mathcal{M}_{\mathcal{P}}(M_j)$.

Algorithm 1: Source Code

```
1 // Entry Point
2 $X = 0$
3 for $I = 1$ to $10$ do
4     if $I \geq 5$ then
5         $X = X + 1$ //Mutation Point
6     $X = X + 1$
```

Algorithm 2: Post Program

```
1 $X = 4$
2 for $I = 5$ to $10$ do
3     if $I \geq 5$ then
4         // Entry Point
5         $X = X + 1$ // Mutation Point
6     $X = X + 1$
```

Algorithms 1 and 2 show two versions of the same snippet of source code from a program. These algorithms will be used to exemplify the notion of the post program.

The entry point of the program in Algorithm 1 is on Line 1. Line 2 assigns 0 to X. Lines 3 to 6 define a loop that iterates 10 times. During each of the first four iterations, variable X is incremented. The mutation point, which is on Line 5, is reached on the fifth iteration.

Algorithm 2 is a version of Algorithm 1, which reflects the state of the program during the execution of Line 4 on the fifth iteration of the loop. Note that $I = 5$ on Line 2 reflects the fact that the program is on the fifth iteration of the loop, the comment on Line 4 communicates the point in the program that the execution has reached, and $X = 4$ on Line 1 because X would have been incremented four times by the last line of the loop by this point in the execution. The execution trace of the program, from this point onwards is said to belong to the post program. Thus, Algorithm 2 can be conceptualised as the post program, such that the entry point of the post program is Line 4.

In this section we introduce measures based on taking the average of Squeeziness or Normalised Squeeziness, where we are averaging over possible post-programs. In particular, Section 2.3.1 introduces one definition of each of these measures, and Section 2.3.2 presents an alternative definition of these measures in which we weight the contributions made by different program states at a given program point.

2.3.1. Average Squeeziness (ASq) and Average Normalised Squeeziness (ANSq)

Given mutant M_j, if FEP occurs then this is a result of a collision in the post-program $\mathcal{PP}(M_j)$ [5, 6].

Let I_j be a discrete random variable that represents the space of reachable states[2] immediately after the mutation point of M_j, where all possible values are given the same probability (i.e. we use a uniform distribution). Also, let O_j be a discrete random variable that represents the collection of program outputs that are produced. The distribution of O_j is induced by program executions of M_j. The supports for I_j and O_j are the respective input and output domains of the post-program, $\mathcal{PP}(M_j)$. The first new measures involve measuring the Sq, or NSq, of post-programs and averaging these values.

Definition 6. *ASq*

$$ASq = \frac{\sum_{j=1}^{m} Sq(I_j, O_j)}{m}$$

Definition 7. *ANSq*

$$ANSq = \frac{\sum_{j=1}^{m} NSq(I_j, O_j)}{m}$$

[2]A program state x is said to be reachable at program point p of P if there is an input to P that leads to the state being x at p.

2.3.2. Weighted Averages: WASq and WANSq

The motivation for interest in the above measures is that the FEP is caused by the post-programs and so an average, over these post-programs, should relate to the overall prevalence of FEP. However, these measures do not take into account the fact that, when considering a mutant M_j, some program state values at the mutation point might make a greater contribution than others since they are more likely to be met in testing.

As before, let T be the set of test inputs being used and consider test input tc_i in T and mutant M_j, In order to provide different weights to different state values at $\mathcal{M}_{\mathcal{P}}(M_j)$, we define a discrete random variable I_j^W such that the support for I_j^W is the set of states produced at $\mathcal{M}_{\mathcal{P}}(M_j)$ when using T and the probability assigned to a state σ in I_j^W is proportional to the number of test inputs in T that lead to state σ at $\mathcal{M}_{\mathcal{P}}(M_j)$. Similar to before, the random variable O_j^W is determined by I_j^W and the post-program.

Definition 8. *WASq*

$$WASq = \frac{\sum_{j=1}^{m} Sq(I_j^W, O_j^W)}{m}$$

Definition 9. *WANSq*

$$WANSq = \frac{\sum_{j=1}^{m} NSq(I_j^W, O_j^W)}{m}$$

Note that the difference between the definitions of ASq and WASq is the probability distributions that are being evaluated by Sq and NSq. In particular, ASq models the input domain as a uniform distribution, whilst WASq models the input domain based on a distribution that is induced from program executions. The same difference can be observed between ANSq and WANSq.

3. Experimental Design

In this section we start by giving the research questions addressed and we follow this with details of the experimental subjects. We then describe how the experiments were carried out.

3.1. Research Questions

We are interested in measures that can be used to estimate the testability of a program in terms of likelihood of FEP: Sq, NSq, ASq, ANSq, WASq, and WANSq. Note that the metrics are equivalent to their normalised counterparts for situations in which the size of the test suites are the same. However, some parts of our experiments are based on test suites that have different sizes, and so we would expect normalisation to have an impact for these parts of the experiments. This leads to the following research question for the six information theoretic measures.

Research Question 1. *To what extent are entropy loss measures correlated with Average Failed Error Propagation?*

This is the primary research question. We addressed it by producing values for Average Failed Error Propagation and the six information theoretic measures, based on experiments with a set of subject programs and mutants of these.

A number of choices, such as the mutants and test suite used, were made when computing measures. We would like to have measures that are relatively insensitive to such choices: this would provide additional confidence that the results generalise and so that the measures are of value in practice. The remaining research questions assess the robustness of the measures.

Different faulty versions of a program can produce different measures of Squeeziness and Normalised Squeeziness.

Research Question 2. *How sensitive is the calculation of Squeeziness and Normalised Squeeziness to mutations?*

This research question explores the extent to which Squeeziness and Normalised Squeeziness values vary for different faulty versions of the same program - and so to the choice of mutants used in the experiments.

Recall that a number of the measures were defined in terms of the execution of mutants of a program P; an alternative would have been to have defined them in terms of executions of P.

Research Question 3. *To what extent does the Squeeziness/Normalised Squeeziness of the original program differ from the Squeeziness/Normalised Squeeziness of its mutants?*

Ideally, we should also obtain similar values for a program and its mutants since this would indicate that measures used are relatively robust. If this is the case then the decision, to define measures in terms of mutants rather than the original program, will have had relatively little effect on the results of the experiments.

The calculation of measures also requires a test suite.

Research Question 4. *How stable are measures of Entropy to choice of test suite?*

If the measures are relatively unaffected by the choice of test suite then this provides confidence that the results were not influenced by this choice. It also provides additional evidence that useful estimates of measures can be obtained by random sampling.

3.2. Tools

A number of tools were used in the experiments. One of the main tools used was Milu [10], which was used to automatically generate mutants for the experiments. We selected Milu because it is one of the most widely used mutation testing tools for C, in software engineering research and has a higher degree of automation than other mutation testing tools [11]. GDB [12], a debugger for C, was used to collect state information. We also used CLOC [13], which is a tool for computing the number of lines of code in a program. Finally, we used an implementation of the Trace Alignment Algorithm, to identify mutation points; details regarding this algorithm can be found in Section 3.5.

3.3. Subject Programs

We acquired subject programs from three different sources, all of which contained open source C projects. One such source was CodeFlaws [14], which is a repository of programs that were developed for a programming competition. Another source includes a version of GRETL [15] that was hosted by Github User HelioGuilherme66

Project	**Number of Programs**	**Total number of LOC**	**Number of Mutants**
CodeFlaws	4	183	11
GRETL	13	493	91
WBMPlus	1	6	1

Table 1: Subject Programs

(the last commit in this version of GRETL was made on 29th August 2016), which is a library for economeasure analysis. The final source was WBMPlus, which is a library for calculating various properties of water. In particular, we acquired the core code base (and commits) from Github User Bmfekete [16] and the MFlib and CMlib libraries were obtained from Github User Kettner [17]. The last commit by Bmfekete [16] was made on 12th January 2018 and the last commit by Kettner was on 30 March 2015. Table 1 provides more details regarding our subject programs (note that the number of Lines of Code (LOC) presented were computed by CLOC [13]).

Table 1 shows that we used a total of 18 subject programs. We would like to note that we had an additional pool of 113 programs (some, but not all, were from the same projects that are presented in Table 1) that we intended to include in our experiments, but unfortunately were unable to for various reasons e.g. Milu was not able to generate viable mutants for many of these programs, and some of the programs were too expensive, in terms of execution time, to include in the experiments. These additional open source projects included Naev [18], which is a video game, and was last updated on 24th February 2019, R [19], which is a statistics library, TCAS [20], which is an air traffic control system, and TheAlgorithms [21], which is a collection of implementations of well-known algorithms (the latest commit for TheAlgorithms was 22nd March 2019).

We chose this sample of subject programs because it has the following desirable properties. First, the developers of these subject programs were not aware of the research, reducing the scope for experimental bias. Second, all of the programs are open source, which means that the experiments can be replicated. Third, all of the subject programs accept numeric inputs, simplifying (random) test input generation. Finally, different developers worked on different subject programs; this enhances the representativeness of the experiments.

Several minor modifications were made to the programs to ensure their compatibility with the tools used. To illustrate the nature of these modifications, the remainder of this section will present the set of changes made to one of the programs.

In one class of examples, the unmodified version of the program contained a main method that used the scanf function to read stdin (user input from keyboard) and assigned the user's input to a variable. We modified this main method by renaming it and replacing its call to the scanf function with a variable in the method's signature. In another class, the unmodified version of the program contained a print statement. We replaced this print statement by introducing a string that stored the contents that were to be passed to print statements and added a new print statement that prints this string just before program termination.

Ideally, we would have used programs and 'real' faulty versions of these. However, for a program P, we wanted to have a faulty version of P for each program point p of P. Although we were aware of repositories containing faulty versions of programs, we were not aware of any such sets of faulty versions of a program.

3.4. Producing Mutants

The aim was to produce one mutant for each statement of P. Let $\mathcal{M} = \{M_1, M_2, ..., M_m\}$ be the resultant mutants of P, and so no two mutants in $\mathcal{M}$ have the same mutation point. We observed that $\mathcal{M}$ did not include a mutant for every statement of P and that there were two reasons for this. First, $\mathcal{M}$ was produced by the Milu [10] mutation testing tool, which could not generate mutants for certain program statements. Second, mutants that could not be compiled, crashed, or timed-out[3] were removed. Figure 1 shows the number of lines of code in each program, broken down by mutant generation information.

[3]A 10 minute time-out was used to allow the experiments to complete in a reasonable amount of time.

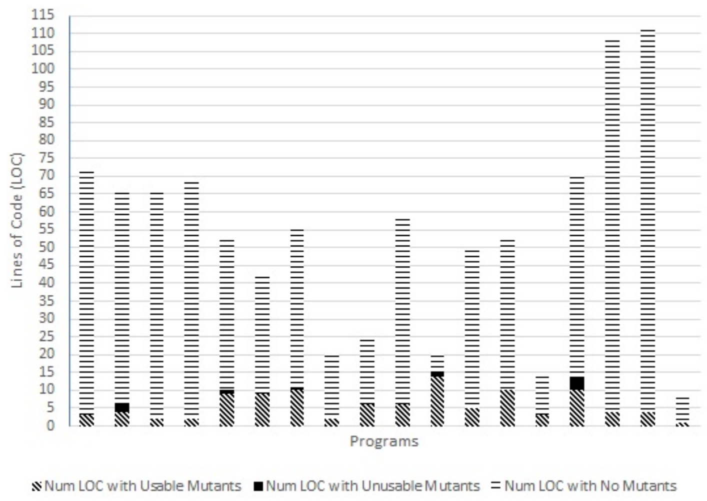


Figure 1: Number of lines of code in each program, broken down by mutant generation information.

3.5. Identifying the Mutation Point

One of the limitations of the Milu mutation testing tool is that it can modify the syntax of program statements that are not mutation points. For example, it might introduce brackets that have no effect on a given program statement; this is akin to refactoring. This means that it is impossible to confidently conclude that a given program statement is the mutation point based on the simple observation that corresponding program statements in the subject program and mutant are different. To that end, a more sophisticated means of identifying the mutation point was required.

Let P be a subject program, and M be a mutant of P. Also let P^{AST} and M^{AST} denote the Abstract Syntax Trees of P and M respectively. The Trace Alignment Algorithm [22] consists of two major steps. The first step is applying a well-known algorithm called the Tree Edit Distance algorithm to P^{AST} and M^{AST}. The Tree Edit Distance algorithm computes the minimum number of changes that need to be made to P^{AST} to transform it into M^{AST} and vice versa, and associates each node in P^{AST} and M^{AST} with a label that describes the required changes e.g. "Keep", "Added", "Deleted", or "Changed".

Let $Trace_i^P = \langle stmnt_1^P, stmnt_2^P, \ldots, stmnt_n^P \rangle$ be the sequence of program statements that were executed in response to the execution of P with test input tc_i. Similarly, let $Trace_i^M = \langle stmnt_1^M, stmnt_2^M, \ldots, stmnt_m^M \rangle$ be the sequence of program statements that were executed in response to the execution of M with the same test input tc_i. The second step of the Trace Alignment Algorithm is to filter out program statements from $Trace_i^P$ that were associated with "Added" or "Deleted" labels in P^{AST}, and to filter out program statements from $Trace_i^M$ that were associated with "Added" or "Deleted" labels in M^{AST}. The output of the Trace Alignment Algorithm is the filtered versions of $Trace_i^P$ and $Trace_i^M$. The first program statement in the filtered version of $Trace_i^P$ that is associated with a "Changed" label is the mutation point.

3.6. Experimental Procedure for RQ1

In these experiments, test suites containing 5000 test inputs were randomly generated for each subject program. We used a test suite of this size in order to obtain relatively accurate estimates of measures. Thus, when considering a subject program P from Section 3.3, we used a test suite $T = \{tc_1, tc_2, ..., tc_{5000}\}$. Let O denote the random variable that corresponds to the resultant outputs. If we let I be the random variable with support T and uniform distribution, then for P we computed Squeeziness $Sq(I, O)$ and Normalised Squeeziness $NSq(I, O)$. It is worth noting that we used such a large test suite since the aim was to provide an estimate of the probability of FEP.

For a program P, we tested each mutant with all test inputs in the corresponding test suite T. Based on this, it was possible to compute a value for the following measures described in Section 2.

- Average Probability of Failed Error Propagation (AVGFEP).
- Average Squeeziness (ASq).
- Average Normalised Squeeziness (ANSq).
- Average Weighted Squeeziness (WASq).
- Average Weighted Normalised Squeeziness (WANSq).

As a result, for each subject program P we obtained a value for every measure. In order to address the first research question, we therefore computed the rank correlation[4] between our notion of testability (AVGFEP) and the other measures.

3.7. Experimental Procedure for RQ2 and RQ3

These two research questions concern the choice of program used to compute the information theoretic measures: RQ2 explores the effect of the choice of mutant at a program point, while RQ3 concerns whether similar values are obtained when using P or a mutant.

As before, we used Milu to generate mutants. However, in this case we were interested in having multiple mutants for mutation points and so we used a larger set of mutants. In order to avoid scalability issues, we used at most 30 mutants for a given mutation points but there were program points where Milu was not able to produce this many mutants.

Similar to before, we computed values for the information theoretic measures. However, we now had up to 30 values for a mutation point, allowing us to explore both how values compared for the original program and a mutant and also how values compared if we considered different mutants with the same mutation point.

Figure 2 communicates the number of mutants per program, partitioned by the mutation point.

In this, each bar represents a subject program and the 'sections' of the x-axis denote the project from which a program was obtained e.g. the four left-most bars represent programs from CodeFlaws. The height of a bar gives the number of mutants generated for the corresponding program e.g. 19 mutants were generated for the program that is represented by the left-most bar. Mutants of a program can be grouped by the mutation point; the partitions of a bar, by colour fills, reflects these groups e.g. the left-most bar tells us that one mutation point had five mutants, another mutation point had another five mutants, and a third mutation point had nine mutants.

[4]We used rank correlations because these are non-parametric.

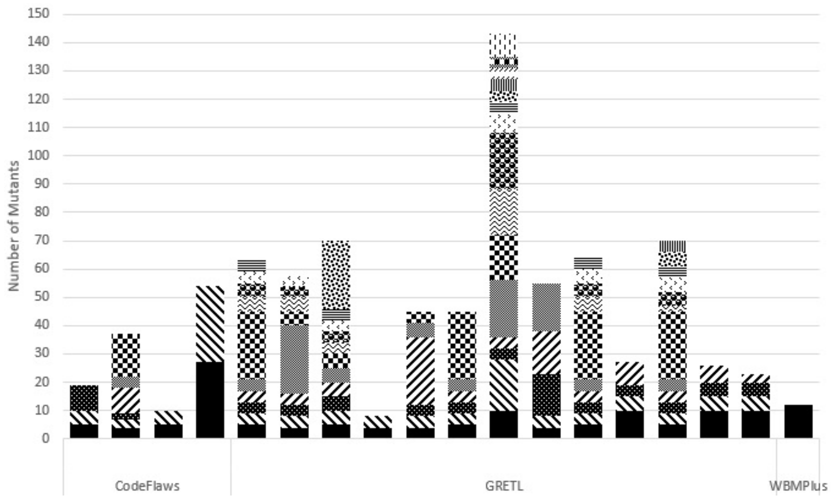


Figure 2: Number of mutants per program, partitioned by the mutation point.

3.8. Experimental Procedure for RQ4

Recall that Research Question 4 concerned how the choice of test suite affected the values produced for the information theoretic measures. In order to address this, for each program P we randomly generated 30 different test suites, with each test suite containing 100 test inputs. We then executed P on all of the test inputs in a test suite T and computed values for the information theoretic measures. This process led to 30 different values for each information theoretic measure, each produced using a different test suite.

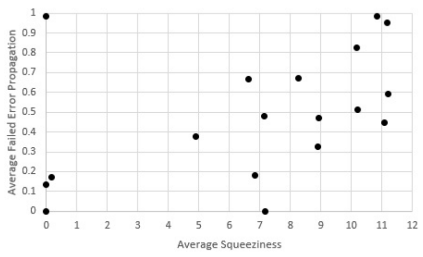
Note that previously we used 5000 inputs so that sampling leads to relatively accurate estimates of the true values of measures. We used smaller test suites, in the experiments that addressed RQ4, in order to explore the variability of the estimates with smaller test suites.

4. Results and Discussion

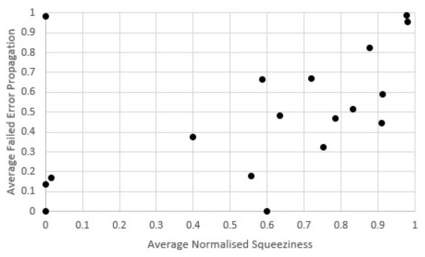
In this section we describe the results of the experiments and what they tell us about the research questions.

4.1. RQ1. To what extent are entropy loss measures correlated with Average Failed Error Propagation?

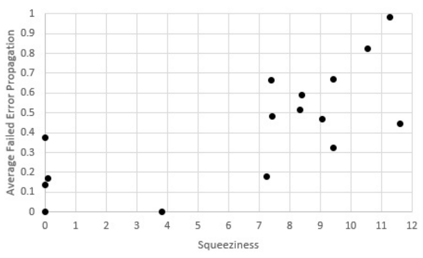
Recall, that the motivation for this question is that we are interested in the Average Failed Error Propagation, as a measure of testability, and so would like to have measures that correlate with this and also are less expensive to compute.



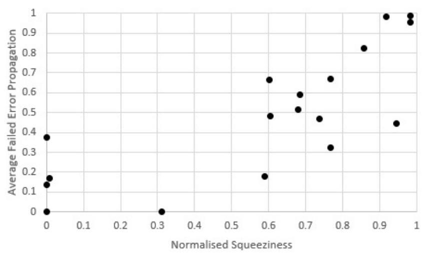
(a) ASq



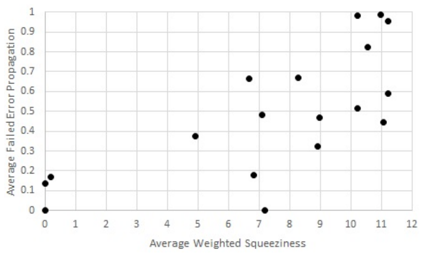
(b) ANSq



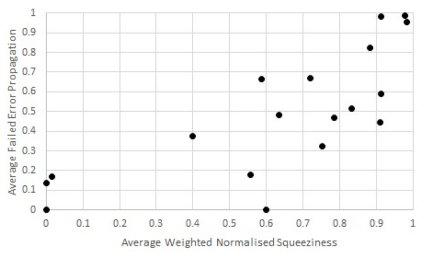
(c) Sq



(d) NSq



(e) WASQ



(f) WANSQ

Figure 3: The x-axis of a scatterplot pertains to one of the measures, the y-axis corresponds to Average Failed Error Propagation, and a marker represents a subject program.

As previously explained, for each subject program P, we produced at most one mutant per program point of P and used a test suite with 5000 randomly generated test inputs. Execution of the mutants and program allowed the Average Failed Error Propagation (AVGFEP) to be computed. Figure 3 plots these AVGFEP values against the information theoretic measures. More specifically, the x-axis of a scatterplot in Figure 3 pertains to one of the measures, the y-axis corresponds to Average Failed Error Propagation, and a marker represents a subject program. We now discuss the results for the three pairs of measures.

4.1.1. ASq and ANSq

Figures 3a and 3b show a tendency for larger values of AVGFEP to be associated with larger values of both ASq and ANSq. We conducted a Spearman's ρ test to measure the correlation between ASq and Average Failed Error Propagation (AVGFEP), and we obtained a correlation coefficient of 0.43 and a p-value of 0.073. This suggests that there is a moderate correlation between ASq and AVGFEP, and by implication ASq can predict AVGFEP to a certain extent.

Similarly, we conducted a Spearman's ρ test to measure the correlation between ANSq and AVGFEP. We obtained a correlation coefficient of 0.52 and a p-value of 0.028, which indicates that the correlation is still moderate but is higher than the correlation for ASq.

Clark et al. [7] observed that Normalised Squeeziness (NSq) is more effective than Squeeziness (Sq) for situations in which programs have variable input domain sizes, but performed comparably when there was no variation in the input domain size. The above results support this observation but, unlike the previous work [7], the results are for programs and not simulations.

4.1.2. Squeeziness and Normalised Squeeziness

Similar to above, Figures 3c and 3d show a tendency for larger values of AVGFEP to be associated with larger values of both Sq and NSq. In order to further explore this, we conducted two Spearman's ρ tests to measure the correlations between Sq and AVGFEP, and NSq and AVGFEP. We obtained the same correlation coefficient and p-values for both of the tests: the correlation coefficient was 0.77 and the p-value was 0.00021. It is unsurprising that we obtained the same values for Sq and NSq since, in this case, the input domains are identical and so normalisation simply involved dividing by a constant. Note that, in contrast, we obtained different results for ASq and ANSq since the number of test inputs for a mutation point can differ.

Observe that Sq and NSq have a substantially higher correlation with AVGFEP than the corresponding values observed with ASq and ANSq. These results are promising for two reasons. First, the efficacy of NSq has not previously been demonstrated on real programs, and these results show that NSq is effective for real programs. Second, Sq and NSq are substantially cheaper to compute than ASq and ANSq, so there is no trade-off associated with using Sq and NSq over ASq and ANSq.

4.1.3. WASq and WANSq

Figures 3e and 3f show similar patterns for the weighted averages. We conducted a Spearman's ρ test to measure the correlation between WASq and AVGFEP and obtained a p-value of 0.0016 and a correlation coefficient of 0.69. We also used Spearman's ρ to measure the correlation between WANSq and AVGFEP and acquired a correlation coefficient of 0.78 and a p-value of 0.00012.

WASq has a substantially stronger correlation than ASq and WANSq's correlation is much stronger than that produced with ANSq. This suggests that the additional information that is considered by WASq and WANSq can add value.

4.1.4. Summary

Table 2 shows the six correlation coefficients. One can see that WANSq obtains the strongest correlation coefficient of all of the measures, which indicates that it should be the preferred measure when accuracy is critical. However, WANSq is substantially more expensive to compute than NSq, which has a correlation that is only marginally weaker; thus NSq is preferable for situations in which it is acceptable to sacrifice accuracy for speed.

The expense of computing NSq increases on a linear scale, as program size increases. In contrast, the expense of WANSq increases much faster, since the increase in the number of mutants would affect scalability. This means that NSq might be preferable for situations that involve large programs and limited budget.

The strengths of the correlations supports the following conclusion.

measure	Correlation Coefficient	P-Value
Sq	0.77	0.00021
NSq	0.77	0.00021
ASq	0.43	0.073
ANSq	0.52	0.028
WASq	0.69	0.0016
WANSq	0.78	0.00012

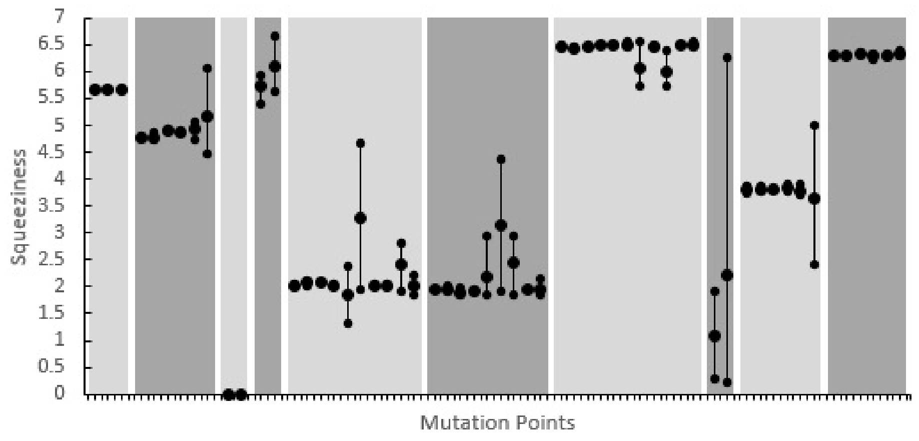
Table 2: Summary of Spearman's ρ correlation coefficients that were computed between each measure and Average Failed Error Propagation.

Entropy Loss measures have a moderate to strong correlation with Average Failed Error Propagation. WANSq obtains the strongest correlation coefficient, but is substantially more expensive to compute than NSq, which has a marginally weaker correlation coefficient.

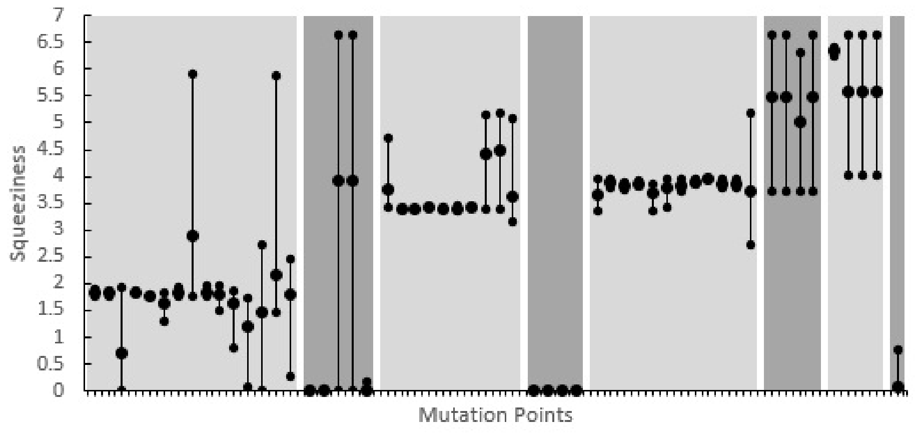
4.2. *RQ2. How sensitive are the calculations of Squeeziness and Normalised Squeeziness to mutations?*

Recall that we generated up to 30 mutants for each mutation point and we addressed Research Question 2 by looking at how Sq and NSq varied for the mutants produced at a single mutation point. The main reason for exploring this was that we wanted to know whether the choice of mutant, for a given mutation point, might have had a significant impact on results.

Figure 4 shows the variation in Sq values for the different mutants with the same mutation point. A value on the x-axis represents a single mutation point and the mutation points are grouped by program. For example, the first three intervals of the x-axis in Figure 4a represent one subject program. Three values are given for each mutation point and these are the minimum, mean, and maximum values of Sq obtained for mutants at that mutation point. For example, the last interval of the x-axis in Figure 4b represents a mutation point where the smallest value of Sq observed was 0, the largest value was 0.76, and the mean value was 0.065. The mutation points within a region



(a)

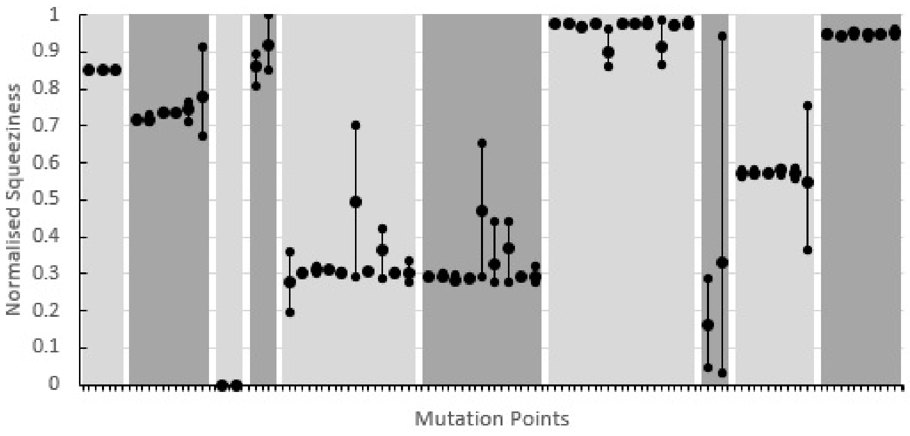


(b)

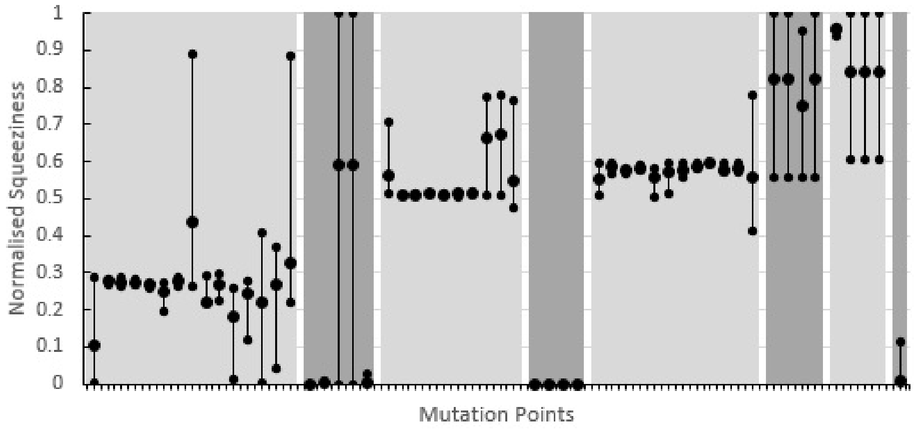
Figure 4: Each point on the x-axis represents a single mutation point and for each of these the three vertically values give the minimum, mean, and maximum Sq obtained for mutants at that mutation point. The x-axis is divided into intervals representing a subject program.

corresponding to a single program were sorted in ascending order of the number of mutants. Figure 5 gives the corresponding plot for NSq.

These figures indicate that Sq and NSq vary considerably more for some mutation points than others. This suggests that for some mutation points, one might obtain more accurate results, in terms of predicting testability, if one used multiple mutants. For example, ASQ and ANSQ might be enhanced by sampling multiple mutants per



(a)



(b)

Figure 5: This figure gives the data for NSq, in a manner analogous to Figure 4. Its worth noting that the ordering of x-axis in this figure differs from the ordering of the x-axis in Figure 4 for consecutive mutation points that have the same number of mutants.

mutation point, instead of just one. We intend to investigate this in future work.

Interestingly, the average Squeeziness and Normalised Squeeziness of the mutation points in a given program is very similar. This suggests that specific mutation points are not a particularly important determinant of Squeeziness and Normalised Squeeziness. Figures 4 and 5 also suggest that the extent to which Sq and NSq values vary at a mutation point is partly determined by the subject program.

Recall that Figure 2 describes the numbers of mutants used in this analysis. The number of mutants at each mutation point ranged from 2 to 27 (7.4 on average). The variation in number of mutants for a mutation point might thus be a possible explanation for the variation in Squeeziness/Normalised Squeeziness discussed above.

In order to investigate this further, we plotted the number of mutants generated for a mutation point against the Standard Deviation of the Squeeziness values that were produced for mutants at that mutation point. This is shown in Figure 6, in which the x-axis pertains to the number of mutants that were generated for a mutation point, the y-axis corresponds to the Standard Deviation of the Squeeziness values that were produced for mutants at a mutation point, and a marker represents a mutation point. Figure 7 is the corresponding figure for NSq. These figures suggest that there is relatively little relationship between the variation in either Sq or NSq and the number of mutants. In order to explore this further, we conducted a Spearman's ρ Test to assess the correlation between the number of mutants generated for a mutation point and the Standard Deviation of the Squeeziness. We obtained a correlation coefficient of 0.27 and a p-value of 0.0047; similar results were produced for NSq. This suggests that, while the variability of Squeeziness might be partially explained by the number of mutants, this is not the only factor.

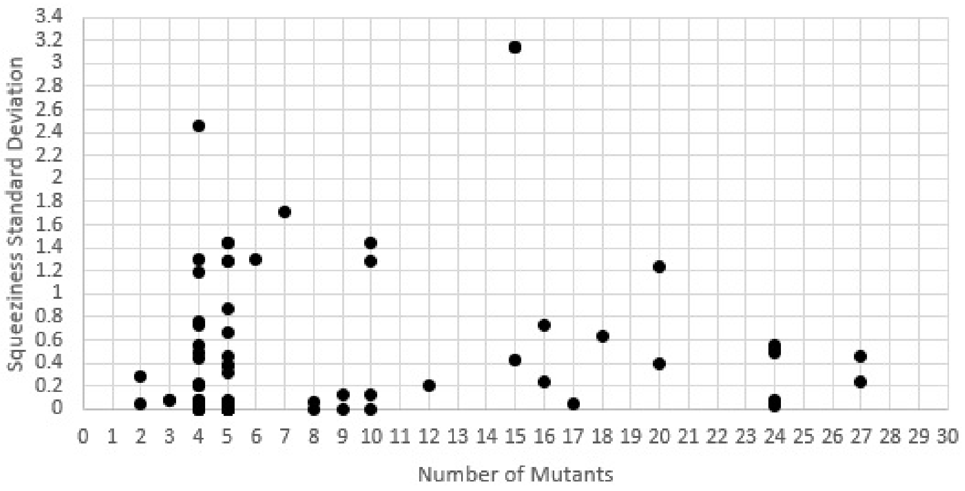


Figure 6: The x-axis pertains to the number of mutants that were generated for a mutation point, the y-axis corresponds to the Standard Deviation of the Squeeziness values that were produced for mutants at a mutation point, and a marker represents a mutation point.

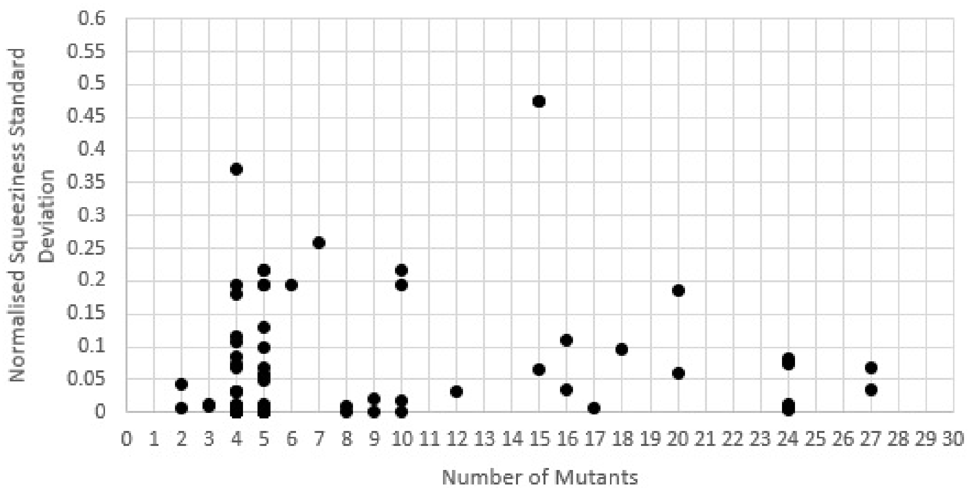


Figure 7: This figure is the analogue of Figure 6 for Normalised Squeeziness.

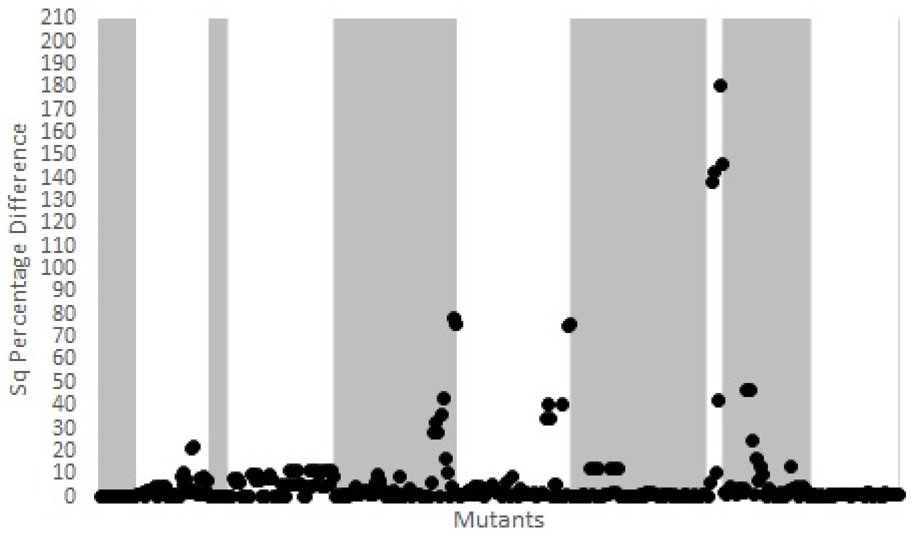
Some mutation points have greater variance in Squeeziness/Normalised Squeeziness than others. As a result, the use of multiple mutants for a mutation point might have benefits. However, the differences observed tended to be relatively small.

4.3. RQ3. To what extent does the Squeeziness/Normalised Squeeziness of the original program differ from the Squeeziness/Normalised Squeeziness of its mutants?

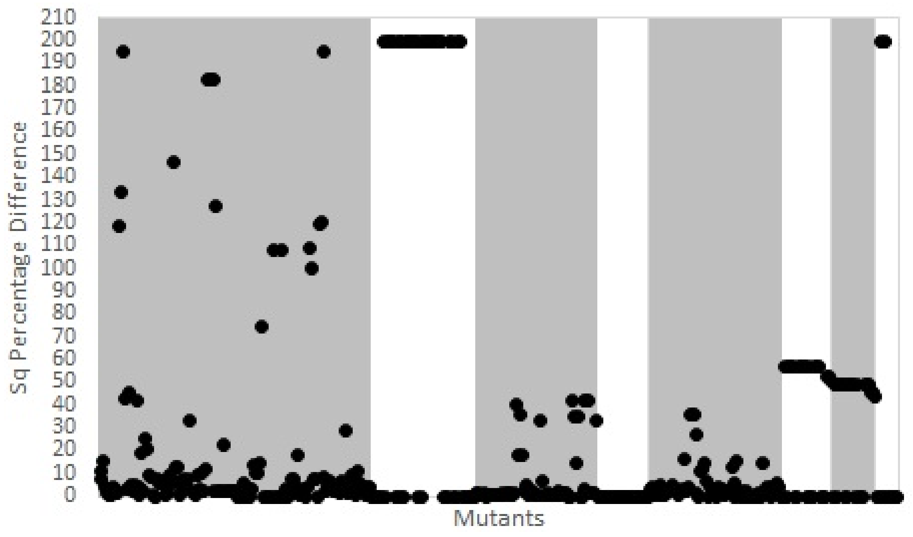
Recall that this research question relates to another robustness concern: how Sq and NSq vary between a program and its mutants. This was motivated by a desire to know whether the choice to use mutants, in computing the six information theoretic measures, had a significant impact (we might instead have used the original program).

Figure 8 shows the differences between the Sq values for mutants and the original program. The x-axis gives the different mutants, while the y-axis gives the difference between the Squeeziness of a mutant and the original program. As in Figure 4, the background grey shading enables one to differentiate between subject programs. Figure 9 gives the corresponding results for Normalised Squeeziness. Figure 10 provides the proportional difference for the values represented in Figures 8 and 9.

A number of observations can be made from Figures 8, 9, and 10. First, the



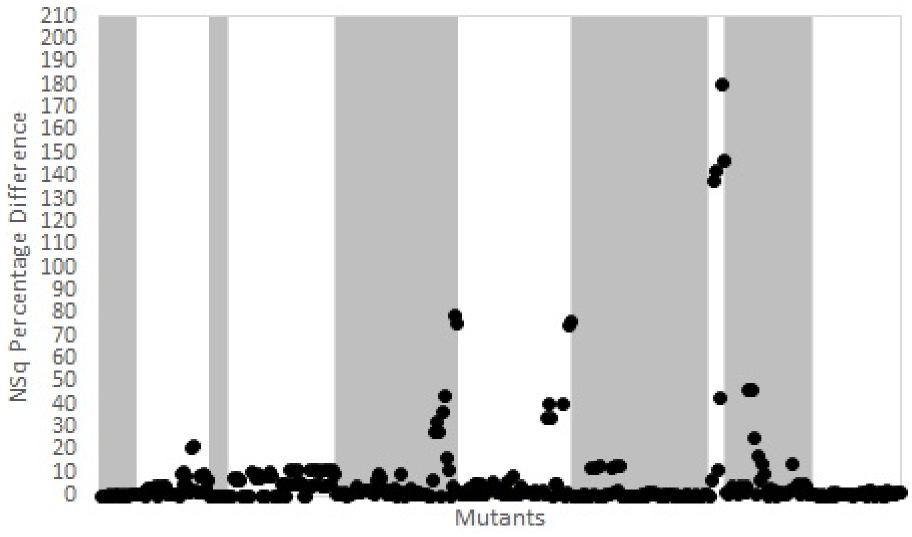
(a)



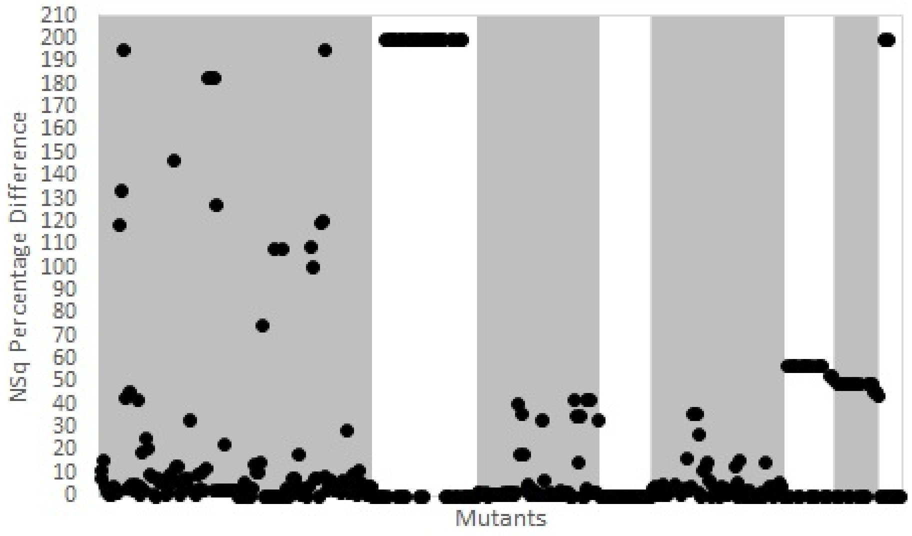
(b)

Figure 8: The points on the x-axis correspond to mutants and the y-axis is the difference between the Squeeziness of a mutant and the original program.

Squeeziness/Normalised Squeeziness of most mutants is very similar to that of the original program. This suggests that in most cases the choice to use mutants, rather than the original program, will have had little effect.



(a)



(b)

Figure 9: This figure is the analogue of Figure 8 for Normalised Squeeziness.

There are a number of exceptions however; these are depicted by the markers situated in high positions along the Y-Axis. It is worth mentioning that many of these exceptions relate to only a few subject programs; in particular, the 8th, 16th and 17th programs in Figure 10. A Squeeziness value is primarily determined by the distribution of a test suite and the outputs of this test suite. Since, all of the mutants used the same test suites, the distribution of the outputs is the only variable factor. This means that these exceptions were caused by mutants that more radically altered the distribution of the outputs.

In most cases, there is little difference between the Squeeziness/Normalised Squeeziness of the original program and a mutant. However, there are exceptions, and these exceptions seem to be more heavily concentrated in certain subject programs. Some mutants increase the amount of information loss in the program, while others reduce it.

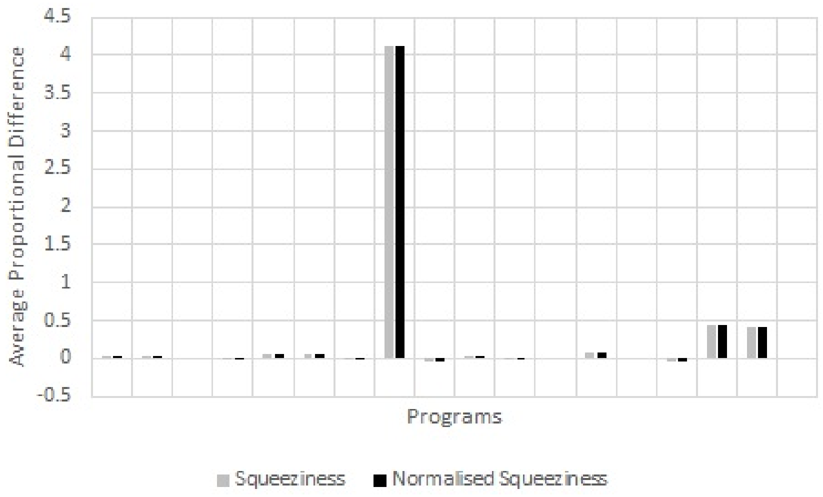
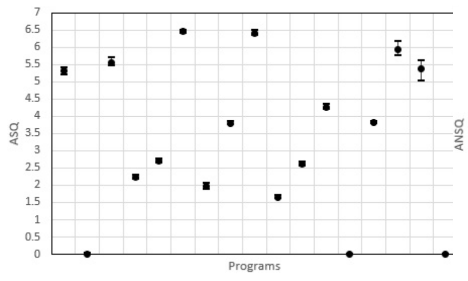


Figure 10: Let P be a program with m mutants, $\mathcal{M} = \{M_1, \ldots M_m\}$. Also let $InfMet$ be a function that computes either Squeeziness or Normalised Squeeziness. Finally, let $MVal = \frac{\sum_{i=1}^{m} InfMet(M_i)}{m}$ and $Oval = InfMet(Prog)$. The Average Proportional Difference of $Prog$ can be calculated as $\frac{(MVal-Oval)}{Oval}$. This graph shows the Average Proportional Difference of each Program.

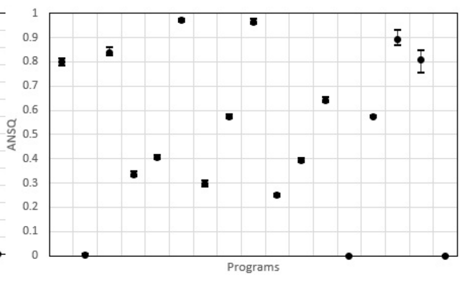
4.4. *RQ4. How stable are measures of Entropy to choice of test suite?*

Recall from Section 3.6 that, for a given program P and information theoretic measure, we calculated the measure 30 times based on P, each time with a different randomly generated test suite. Let DS_{ELM} denote the 30 ELM values that were produced for a measure ELM. In this section, we investigate the stability of Entropy Loss measures, based on the DS_{ELM} values.

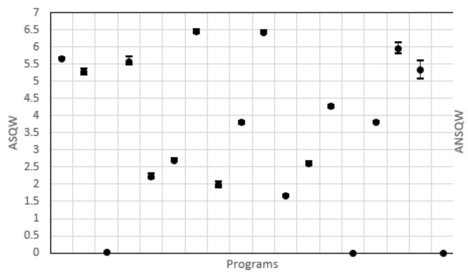
Figure 11 provides a visual representation of the DS_{ELM}. Each scatterplot corresponds to a different measure and each point on the x-axis of a given scatterplot denotes a program. For a given program, three data-points are presented: the minimum, average, and maximum values in the corresponding set DS_{ELM}. One can see from this that there were only small differences in the values computed. A few exceptions can also be observed; in these cases, it is likely that the test suites were too small to consistently provide an accurate approximation of the input domain.



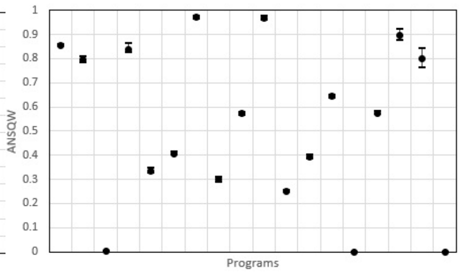
(a) DS_{ASq}



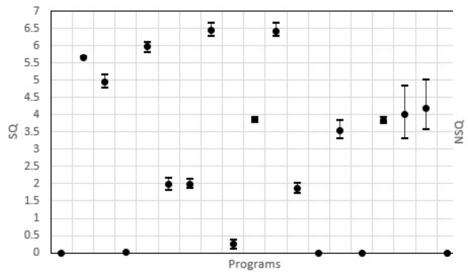
(b) DS_{ANSq}



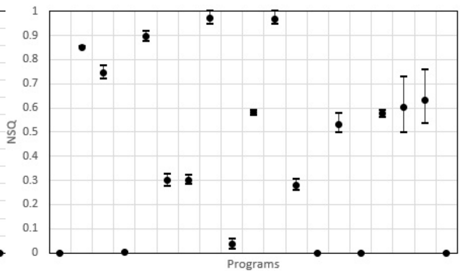
(c) DS_{WASq}



(d) DS_{WANSQ}



(e) DS_{SQ}



(f) DS_{NSQ}

Figure 11: Each plot relates to a different measure. The points on the x-axis represent the different subject programs and the three corresponding values give minimum, average, and maximum observed values of the measure.

	ASq	**ANSq**	**WASq**	**WANSq**	**Sq**	**NSq**
Min	0	0	0	0	0	0
Average	0.037	0.0055	0.034	0.0051	0.081	0.012
Max	0.13	0.020	0.14	0.021	0.36	0.054

Table 3: For each program P and measure ELM, we calculated the standard deviation of all of the ELM values obtained for P. For a measure ELM, the table gives the minimum, average, and maximum standard deviation values that were observed across all of the programs.

.

For each subject program P and measure ELM, we also calculated the standard deviation of all of the ELM values obtained for P. The minimum, average, and maximum standard deviation values observed across all of the programs is presented in Table 3.

Figure 11 and Table 3 indicates that all of the Entropy Loss measures are stable under variation of the test suite, and by implication are reliable measures. These results provide confidence in the overall results, since they are unlikely to have been affected by the choice of test suite. They also suggests that, in practice, one might use estimates of these measures.

All of the Entropy Loss measures are relatively stable under change of test suite.

5. Threats to Validity

There are a number of possible sources of internal threats to validity. The study relied on a number of tools, which might have faults. To minimise this threat, when we used third party tools, we used tools that have been widely used in similar work. Tools developed internally were thoroughly tested.

The experiments used the GDB debugger to capture state information. However, some of the state information captured was in a hexadecimal format and appeared to correspond to memory locations. If these memory locations were used in comparing behaviours then this might lead, for example, to a mutant being incorrectly classified as weakly killed. We therefore removed such state information. This could threaten the validity of the results because we might have discarded relevant state information.

Recall that the proposed information theoretic measures require data for every program statement. In some cases this was not possible, either because Milu was not able to generate mutants for a given program statement or the test input generator did not produce test input that reached the mutation point. To minimise this threat to validity, we used a large number of test inputs for each subject program, to maximise the probability that test suites reach every mutation point.

Finally, we observed that Milu modified the code of one of the mutants such that GDB's definition of a basic block was altered for this mutant. To circumvent this issue, we excluded this mutant.

There are also external threats to validity. First, we only used 18 subject programs and all of our subject programs only accept numeric inputs. It is unclear whether the results would generalise to non-numeric programs. However, we note that numeric programs form an important and widely used class of programs, and so our results have value for a large cross-section of programs. It's also worth noting that we chose subject programs that are relatively diverse in terms of domain.

The experiments were conducted on mutants instead of real faults. This is a threat to validity because mutants might not be representative of real faults. Unfortunately, it was not possible to use real faults in the experiments because there was not a sufficient number of real faults: most of the programs were only associated with one real fault but we required at least one fault for each program statement.

The Milu mutation testing tool was unable to generate mutants for every program statement, which limited the number of mutants that were considered in our experiments. This is a threat to generalisability, since a greater number of mutants might be available in other contexts. The introduction of handcrafted mutants might have alleviated this threat, but would then have increased the susceptibility of our results to experimental bias. We note that there are likely to be scenarios in the field, where budget pressures would limit the number of mutants that can be used, and so our results are likely to be representative of such scenarios.

Initially, large test suites, with 5000 test inputs, were used in order to obtain relatively accurate estimates of the probability of FEP. These were not intended to be representative of the types of test suites that will be used in practice. It is not entirely clear how the probability of FEP, with a randomly generated test suite, will relate to the probability of FEP for a test suite generated to achieve an objective such as coverage. We see this as a question for future work. It is worth noting, however, that approximately 90% of the mutants were killed under strong mutation and this suggests that the test suites provided relatively high statement coverage.

An additional threat is that the probability of FEP occurring at a particular program point was estimated based on a single mutant. Whether FEP occurs depends on the exact nature of the state disruption at a program point. However, estimations based on single mutant were already expensive. An alternative would be to consider a wide range of both seeded errors and real faults in order to estimate this probability more accurately. Such an undertaking will be very expensive and there is at present no obvious stopping point as the size of the potential error space is likely to be larger than the size of the program space.

6. Related Work

The notion of FEP was first formally explored in the context of coincidental correctness; the situation in which a fault in the SUT is executed but the output produced is correct. Here, the observation was that if the SUT has a fault then this will only be found in testing if the fault is executed (execution), the execution of the fault leads to an incorrect program state (infection), and the incorrect program state propagates to incorrect output (propagation). These three conditions are reflected in the PIE framework [1]. More recently, this has been generalised to the RIPR framework in which it is recognised that a fourth condition must be satisfied: a test oracle must recognise that the output is incorrect [2].

A number of empirical studies have found that coincidental correctness can have a major impact on testing [3, 4]. For example, Masri et al. found that, within a set of programs studied, most were affected by coincidental correctness and in some cases many tests were affected. In particular, for 13% of the programs they studied, 60% or more of the tests were affected [23, 24, 25].

If one considers the RIPR framework, it is clear that coincidental correctness can occur through there being no infection, there being no propagation, or propagation occurring but not being recognised by the test oracle used. Failed Error Propagation (FEP) represents the second of these: a faulty state that occurs during the execution of an SUT does not propagate to the output.

This has motivated a number of researchers to study means of alleviating FEP. Alshahwan and Harman [26] conjectured that test cases that produce different outputs are more likely to take different paths through the program and that this diversity might help avoid FEP. This motivated them to develop a new test selection criterion called Output Uniqueness, which attempts to maximise the number of test cases in the test suite that produce unique outputs. The results of their evaluation of Output Uniqueness were promising.

FEP is caused by two or more program states being mapped to the same program output and this represents a loss of information. This observation, that FEP represents a loss of information, led to the definition of a measure, Squeeziness (Sq), that measures the information (entropy) loss, between two program points, that results from the execution of a program [5, 6]. It was found that there is a strong correlation between the probability of FEP (for a fault at a program point p of program P) and the Squeeziness of the program that is formed by starting the execution of P at p (the post-program) [6]. This demonstrates that Sq can be used to reason about the probability of FEP associated with a given program point. However, it was observed that if we compare programs with different size program states then the use of Sq can be misleading. This led to the definition of Normalised Squeeziness (NSq), which measures the *proportion* of entropy lost in program execution. Simulations demonstrated that NSq is more effective than Sq when comparing programs (functions) with different input domains [7]. However, this is the first paper to evaluate NSq on real programs. In addition, previous work has not investigated how the Sq and NSq of a program P relate to the probability of FEP in P as a whole, where a fault might occur in any statement.

7. Conclusions

This paper explored a number of information theoretic measures and how they relate to a notion of testability (the probability of Failed Error Propagation occurring in testing). The measures considered included two previously defined measures (Squeeziness and Normalised Squeeziness) and four novel, more fine-grained, measures.

The measures were assessed through experiments with a number of case studies. The results were promising, with a strong rank correlation being found between testability and several of the measures. This indicates that there is potential to use these measures as part of a process that prioritises testing or estimates how much testing is required.

Mutants were used to estimate the testability of a program and the experiments used randomly generated test suites. We therefore carried out additional experiments that explored how sensitive the measures are to the choice of mutants and the choice of test suite. Importantly, it was found that changes in mutant or test suite led to only relatively small changes in estimates. This suggests that the measures, and so the experimental results, are relatively robust.

An important outcome of the experiments was that the Normalised Squeeziness of the SUT was one of the most effective measures. This is also one of the simpler measures to compute or estimate, suggesting that Normalised Squeeziness might be particularly suitable for use. In addition, the results regarding sensitivity to choice of test suite suggest that it may be possible to obtain useful estimates of Normalised Squeeziness through random sampling.

There are a number of lines of future work. First, there would be value in carrying out additional experiments. For example, the use of mutants was motivated by the need to have many faults for each experimental subject, but it would be interesting to see whether similar results are obtained with real faults. It would also be interesting to see how results are affected by, instead of using randomly generated test suites, using test suites generated in order to achieve an objective such as coverage. In addition, it would be interesting to explore the use of the measures in test prioritisation techniques or in the process of estimating how much testing will be required.

References

[1] J. Voas, PIE: a dynamic failure-based technique, IEEE Transactions on Software Engineering 18 (8) (1992) 717–727.

[2] N. Li, J. Offutt, Test oracle strategies for model-based testing, IEEE Transactions on Software Engineering 43 (4) (2017) 372–395.

[3] T. Apiwattanapong, R. A. Santelices, P. K. Chittimalli, A. Orso, M. J. Harrold, MATRIX: maintenance-oriented testing requirements identifier and examiner, in: P. McMinn (Ed.), Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), IEEE Computer Society, 2006, pp. 137–146.

[4] R. A. Assi, C. Trad, M. Maalouf, W. Masri, Coincidental correctness in the Defects4J benchmark, Software Testing, Verification and Reliability 29 (3) (2019).

[5] D. Clark, R. M. Hierons, Squeeziness: An information theoretic measure for avoiding fault masking, Information Processing Letters 112 (8-9) (2012) 335–340.

[6] K. Androutsopoulos, D. Clark, H. Dan, R. M. Hierons, M. Harman, An analysis of the relationship between conditional entropy and failed error propagation in software testing, in: Proceedings of the 36th International Conference on Software Engineering, ACM, New York, USA, 2014, pp. 573–583.

[7] D. Clark, R. M. Hierons, K. Patel, Normalised squeeziness and failed error propagation, Information Processing Letters 149 (2019) 6–9.

[8] T. M. Cover, J. A. Thomas, Elements of Information Theory: Second Edition, John Wiley & Sons, 2006.

[9] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (5) (2010) 649–678.

[10] Y. Jia, M. Harman, Milu: A customizable, runtime-optimized higher order mutation testing tool for the full C language, in: Proceedings of the Testing: Academic

and Industrial Conference - Practice and Research Techniques, IEEE, Windsor, UK, 2008, pp. 1–5.

[11] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, I. Medina-Bulo, Evaluation of mutation testing in a nuclear industry case study, IEEE Transactions on Reliability 67 (4) (2018) 1406–1419.

[12] GNU Project, GDB: The GNU Project Debugger, `https://www.gnu.org/software/gdb/` (2020).

[13] AlDanial, CLOC, `https://github.com/AlDanial/cloc` (2020).

[14] S. H. Tan, J. Yi, Yulis., S. Mechtaev, A. Roychoudhury, Codeflaws: a programming competition benchmark for evaluating automated program repair tools, in: Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, Buenos Aires, Argentina, 2017, pp. 180–182.

[15] A. Cottrell, R. Lucchetti, Gnu Regression, Econometrics and Time-series Library, `https://github.com/HelioGuilherme66/gretl` (2016).

[16] B. M. Fekete, Z. Tessler, R. Stewart, Water Balance/Transport Model, `https://github.com/bmfekete/WBMplus` (2018).

[17] A. Kettner, The WBMsed model, `https://github.com/csdms-contrib/wbmsed` (2015).

[18] T. G. Community, Naev, `https://github.com/naev/naev` (2019).

[19] T. R. Community, The R Project for Statistical Computing, `https://www.r-project.org/` (2019).

[20] T. Ostrand, TCAS, `https://sir.csc.ncsu.edu/php/previewfiles.php` (2005).

[21] T. G. Community, All Algorithms implemented in C, `https://github.com/TheAlgorithms/C` (2019).

[22] G. Jahangirova, Oracle Assessment, Improvement and Placement, https://discovery.ucl.ac.uk/id/eprint/10072699/1/Jahangirova_10072699_Thesis.pdf (2019).

[23] W. Masri, R. A. Assi, M. El-Ghali, N. Al-Fatairi, An Empirical Study of the Factors That Reduce the Effectiveness of Coverage-based Fault Localization, in: Proceedings of the 2nd International Workshop on Defects in Large Software Systems, ACM, NY, USA, 2009, pp. 1–5.

[24] W. Masri, R. A. Assi, Prevalence of coincidental correctness and mitigation of its impact on fault localization, ACM Transactions on Software Engineering and Methodology 23 (1) (2014) 1–28.

[25] R. A. Assi, C. T. M. Maalouf, W. Masri, Does the testing level affect the prevalence of coincidental correctness?, CoRR abs/1808.09233 (2018). arXiv:1808.09233.
URL http://arxiv.org/abs/1808.09233

[26] N. Alshahwan, M. Harman, Coverage and fault detection of the output-uniqueness test selection criteria, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, 2014.