An Analysis of the Relationship between Conditional Entropy and Failed Error Propagation in Software Testing

Kelly Androutsopoulos
Middlesex University, UK

David Clark
University College London, UK

Haitao Dan
University College London, UK

Robert M. Hierons
Brunel University, UK

Mark Harman
University College London, UK

ABSTRACT

Failed error propagation (FEP) is known to hamper software testing, yet it remains poorly understood. We introduce an information theoretic formulation of FEP that is based on measures of conditional entropy. This formulation considers the situation in which we are interested in the potential for an incorrect program state at statement s to fail to propagate to incorrect output. We define five metrics that differ in two ways: whether we only consider parts of the program that can be reached after executing s and whether we restrict attention to a single program path of interest. We give the results of experiments in which it was found that on average one in 10 tests suffered from FEP, earlier studies having shown that this figure can vary significantly between programs. The experiments also showed that our metrics are well-correlated with FEP. Our empirical study involved 30 programs, for which we executed a total of 7,140,000 test cases. The results reveal that the metrics differ in their performance but the Spearman rank correlation with failed error propagation is close to 0.95 for two of the metrics. These strong correlations in an experimental setting, in which all information about both FEP and conditional entropy is known, open up the possibility in the longer term of devising inexpensive information theory based metrics that allow us to minimise the effect of FEP.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools and code inspections*

General Terms

Information Theory, Experimentation, Verification.

Keywords

Program Analysis, Information Theory.

ICSE '14, May 31 – June 7, 2014, Hyderabad, India

1. INTRODUCTION

Coincidental correctness occurs when the program happens to produce the correct output for some input even though it has executed a fault; the program is *coincidentally* correct rather than *actually* correct. One of the causes of coincidental correctness is known as Failed Error Propagation (FEP) [17, 28, 31]. In this situation, a faulty statement is executed and the resulting internal computational state becomes faulty, but the differences between the faulty and correct state fail to be observed at output. We say that the error (the faulty state) has 'failed to propagate'.

Empirical studies have revealed that FEP inhibits effective software testing [4, 18, 22, 23, 24, 29] but it remains unclear how software testing could be better designed to ameliorate the problems it causes. In order to improve software testing, we need to reduce the probability that test cases will suffer from FEP. However, in order to do that, we need metrics that can help us to identify parts of a program that are more likely to lead to FEP.

Failed error propagation can occur for a number of reasons. For example, it might be that the faulty state is simply never inspected by the test oracle. In this case, the failure to propagate the error is caused by an inadequate oracle rather than by any inherent property of the program under test. Such failures of error propagation could be addressed by oracle improvement [26, 30].

A more interesting class of FEP occurs when the program itself removes traces of an error before it has had a chance to propagate to a point at which it can be observed. For this to occur, faulty state changes must become 'lost' along some paths through the program because state update functions along these paths 'squeeze out' the faulty information [7]. One obvious way in which this could occur is when a path contains a 'killing assignment', which overwrites the value of the variable with a constant.

A killing assignment is the most extreme example of a state update function squeezing out information. In general, *any* computation that reduces entropy of inputs will have the potential to 'squeeze out' error information and thereby lead to failed error propagation. This loss of error information suggests a connection between FEP and information theory, but this relationship has been little explored in the literature.

This paper introduces an information theoretic formulation of FEP. We introduce five different metrics, based on the computation of conditional entropy in a program, with these differing in the parts of the program considered. When considering a statement s, one natural approach is to exam-

ine a single path π since a test case will lead to a path being followed. However, a test case might lead to different paths in some idealised correct program P and the program under test P'; by only considering a single path we do not recognise the potential for FEP associated with this pair of paths to occur. We therefore also consider approaches that look at the part of the program that 'follows' s (the statements that can be reached from s).

In total, we introduce and evaluate five different metrics, experimentally evaluating each on 30 different programs. Our results are based on the execution of 7,140,000 test cases over 1,428 original and fault-seeded versions of the programs. In these experiments one in 10 test inputs suffered from FEP; these results are in line with those of Masri et alia [18]. Masri et alia also found that the potential for FEP differs significantly between programs, with over 60% of tests being affected by FEP in 13% of the programs studied [18]. For each of the five metrics, we performed an experimental evaluation of its relationship to FEP. We report Spearman rank correlations between each of the metrics and FEP.

The results of these experiments are promising, indicating strong Spearman rank correlations between several of our conditional entropy metrics and failed error propagation.

The primary contributions of this paper are as follows.

1. We introduce an information theoretic approach to FEP, defining associated metrics.

2. We experimentally investigate the correlation between the metrics and FEP, using 30 programs, including open source programs and laboratory benchmarks. The results reveal a Spearman rank correlation of over 0.95 for one metric and just under 0.95 for another.

The main motivation for this work is that a better understanding of FEP has the potential to lead to more effective testing. In particular, the strong correlations in an experimental setting in which all information about both FEP and conditional entropy is known opens up the possibility in the longer term of devising inexpensive information theory based metrics that allow us to minimise the effect of FEP when choosing test cases.

The rest of this paper is organised as follows. In Section 2 we briefly describe related work and in Section 3 we provide background information regarding program semantics and information theory. In Section 4 we outline how FEP and conditional entropy relate conceptually, with this feeding into research questions and associated hypotheses that are given in Section 5. Section 6 outlines the experiments and in Section 7 we give the results of these experiments. Finally, in Section 8, conclusions are drawn and potential lines of future work described.

2. RELATED WORK

Voas introduced the PIE framework and so explicitly recognised the need for an incorrect program state to propagate to output in order for a failure to occur [28]. The notion of FEP was also explored by Laski et alia [17], who called this error masking. They explored the concept and proposed the use of a mutation approach to estimate the sensitivity of a given test suite and program component C: this is the likelihood of an incorrect value produced by C being masked through FEP. The approach taken to mutation was to directly mutate the program state (change it to some randomly generated state).

Masri and Podgurski used experiments to explore variable dependence within a program and how this relates to (an estimate of) information flow [20]. They found that many cases where there is a dependence (through a mixture of control and data dependence), there was negligible information flow. This helps motivate our work, since the lack of true information flow could be one source of FEP (although FEP can occur even when there is a significant amount of information flow). It also suggests that if we just use dependence when, for example, choosing test cases to exercise program elements then we are likely to encounter FEP and so there is a need for alternatives.

Masri et alia explored factors that adversely affect coverage based fault localisation methods [18]. Such methods assign a 'suspiciousness' value to a program element s (such as a statement) in order to allow the developer to focus on those elements that are most likely to be responsible for observed failures. They do this based on how many failing tests execute an element s and how many passing tests execute s. In their study, which used 148 versions of ten Java programs seeded with faults, they found that FEP (which they called coincidental correctness) was relatively common but also the rates varied with program: in 13% of programs over 60% of tests that led to a corrupted state did not produce a failure while in 28% this effect was not observed.

Wang et alia [29] also considered the effect of FEP on fault localisation. However, their approach was quite different and involved producing multiple versions of each program element by adding in 'program patterns'. Each context pattern describes aspects of the control flow and data flow after this element. The results of experiments suggested that this approach reduces the effect of FEP on fault localisation. Potentially there would be value in adopting a similar approach to define coverage metrics.

In mutation testing we judge the adequacy of test suite T for program p by executing T on mutants, which are produced by making small changes to p. A mutant m of p is weakly killed by test case t if the executions of m and p produced different sequences of program states. In contrast, m is strongly killed by t if the execution of m and p with t lead to different outputs. Thus, FEP corresponds to the difference between weak mutation testing and strong mutation testing; experiments have shown that there are significant differences between weak and strong mutation testing [22, 23], again indicating that FEP is relatively common.

Masri and Assi considered how test suites can be cleansed of coincidental correctness (FEP) [19]. Their approach assumes that the test cases have already been applied (and so are suitable for regression testing) and identifies program elements that appear in all failing runs but also in a percentage of runs that do not fail (the percentage must meet a threshold value): these are considered to be a potential source of coincidental correctness. Then test cases are removed from consideration in fault localisation based on which identified elements they contain.

Chen et alia [6] defined a measure that aims to approximate the probability of coincidental correctness using a syntax based metric. The results of experiments with five small programs (for sorting arrays) were positive and this suggests that such an approach is worth exploring further.

3. BACKGROUND

3.1 Control Flow Graphs (CFG)

A CFG is an alternative representation of the syntax of a program which makes explicit the execution paths that may be travelled in a program. We assume for simplicity that all nodes in the graph are of two types: nodes corresponding to state updates which have a single output edge, and nodes corresponding to control flow decision points which have two output edges, one labelled T and the other labelled F. Sometimes the convention is employed that state update nodes in CFGs are defined in terms of blocks of straight line code but here we assume that each individual statement or call in the program has its own node and outgoing edge.

DEFINITION 1. *(CFG). For a given programming language, a CFG is a pair, $\langle N, E\rangle$, where N is a set of nodes that contains a unique start node and two virtual nodes, n_s a virtual start node which has a single edge, e_s to the unique start node, n_e a virtual exit node with a unique virtual exit edge, e_e, with no successor node, and E is a set of labelled, directed edges, so that*

$$\begin{aligned} N_u &= \{n | n \text{ is an assignment or a function call }\} \\ N_c &= \{n | n \text{ is a control (Boolean) expression }\} \\ N &= N_u \cup N_c \cup \{n_s, n_e\} \\ L &= \{T, F\} \\ E_a &= \{(n_1, n_2) | n_1 \in N_u, n_2 \in N \text{ and an empty label}\} \\ E_c &= \{(n_1, n_2) | n_1 \in N_c, n_2 \in N \text{ and a label in } L\} \\ E &= E_a \cup E_u \cup \{e_s, e_e\} \end{aligned}$$

The virtual nodes, $\{n_s, n_e\}$, are merely for convenience and allow the inclusion of edges e_s and e_e in the CFG that are associated with the initial and exit program points.

DEFINITION 2. *(CFG paths). A* path *in the CFG is a sequence of nodes such that any two successive nodes form an edge in the CFG. An* execution path *in the CFG is a path whose first node is the virtual start node and either the path is infinite or it is finite and the final node is the exit node. A* prefix path *is any finite path whose first node is the virtual start node.*

Note that the set of prefix paths includes the set of finite execution paths.

We will use the notion of a *well-formed subgraph of a control flow graph*, i.e. a subgraph of a CFG which is itself a CFG. Not every subgraph of a CFG is well formed. For example, the subgraph reachable from a control node is not well formed in general as it may lack a unique start node.

3.2 Program Semantics

In what follows we informally set out some concepts from program semantics that occur in the course of our explanations of what we are measuring and why.

We assume a deterministic, imperative language such as C, C++, Java, et cetera. We assume a small step structured operational semantics (SOS) [21] for the programming language. An SOS formally describes, for a given input state, the sequence of state updates and branching decisions that occur along the execution path for the input. This sequence corresponds to a sequence of edges in the CFG that starts with the start node and is either infinite or is finite and terminates in the exit node, i.e. an execution path. On this basis we can associate each state in the SOS sequence with an edge in the CFG and each update instruction or branching decision with a node in the CFG. The SOS for a given program and a given input corresponds to an execution path in the CFG in which each edge is labelled with the states that occur after the execution or evaluation of a node in the CFG. We will refer to this set of CFG paths as the **Execution Semantics** of the CFG.

To define some notation for this idea we put the emphasis on prefix paths rather than execution paths in the CFG and informally define a property that relates a program, a path, an input, an edge on the path and the state reached at that edge using the input. $\Phi_P(\pi, t, e, x)$ is the property that there exists a (possibly incomplete) SOS sequence for input t to program P that corresponds to the prefix path π in the CFG for P and e is an edge in π and x is a state that occurs at e.

DEFINITION 3. *(Execution semantics for a path). Let Π be the set of prefix paths in the CFG of program P and let Σ be the set of all possible states that could occur in the executions of P. The execution semantics for a path in Π has type:*

$$[\![\ ]\!]^{ex} : \Pi \to E \to \Sigma \to 2^{\Sigma}$$

and is defined as

$$[\![\pi]\!]^{ex}_e(t) = \{x \mid \Phi_P(\pi, t, e, x)\}$$

The set of states is non-empty in the case that e is an edge in π and π is a prefix of the execution path for t; otherwise it is empty.

We define three abstractions of the execution semantics: a **Collecting Semantics** for **prefix paths**, for **Control Flow Graphs**, and for **Programs**. These simply collect up sets of states, arising from the execution semantics, which occur at edges in the CFG.

The collecting semantics for a prefix path in a CFG is given by the set of states that can occur at each edge in the path over all runs of the program.

DEFINITION 4. *(Collecting Semantics for a prefix path). The collecting semantics for a path in Π and an edge in E has type:*

$$[\![\ ]\!]^{pa} : \Pi \to E \to 2^{\Sigma}$$

and is defined as

$$[\![\pi]\!]^{pa}_e = \bigcup_{t \in \Sigma} [\![\pi]\!]^{ex}_e(t)$$

The collecting semantics for a CFG is also defined in terms of edges that occur in the CFG and collects the sets of states for all paths that pass through an edge.

DEFINITION 5. *(Collecting Semantics for a CFG). The collecting semantics for a CFG (at an edge in E) has type:*

$$[\![\ ]\!]^{cfg} : E \to 2^{\Sigma}$$

and is defined as

$$[\![e]\!]^{cfg} = \bigcup_{\pi \in \Pi} [\![\pi]\!]^{pa}_e$$

To define the collecting semantics of a program we employ the concept of a *program point*. In a formal semantics of a programming language these are taken to be points in the program syntax before or after the execution of a program construct or statement defined in the grammar of the language. In what follows we define them in terms of nodes in the CFG: a program point is a node in the CFG and the set of states that can occur at that program point is the collection of the CFG semantics of the edges that exit that node, that is, the program point occurs immediately after any node in the CFG and collects up all states that the program may be in, once control passes from this node.

This is consistent with Cousot's reachability semantics [11], an alternative way to present these semantic concepts, but for a fixed language.

DEFINITION 6. *(Collecting Semantics for a Program). The collecting semantics for a program (at a node in N) has type:*

$$[\![\ ]\!]^{pr} : N \to 2^{\Sigma}$$

and is defined as

$$[\![n]\!]^{pr} = \bigcup_{\{e \in E | e=(n,\_)\}} [\![e]\!]^{cfg}$$

We define various semantic concepts in terms of these definitions. The *set of reachable states for a prefix path* π is $[\![\pi]\!]^{pa}_{\omega(\pi)}$ where $\omega(\pi)$ is the edge following the final state update node of π.

The *set of input states for a program* is the collecting semantics for the program at the unique virtual start node, i.e. $[\![n_s]\!]^{pr}$ for $CFG(P)$. The *set of output states for a program* P is the collecting semantics at the virtual edge e_e following the virtual exit node, i.e. $[\![n_e]\!]^{pr}$.

Let Σ be the set of all possible states for a program. The *weakest precondition* for program P to terminate and satisfy ϕ is the largest $\psi \subseteq \Sigma$ so that restricting the set of inputs to ψ produces a subset of $\phi \subseteq \Sigma$ as the set of output states. We write $\psi = [P]\phi$. This can be extended to paths and CFGs in the obvious way.

3.3 Information Theory

We consider total, onto functions with fixed, finite, discrete domains. That is, a function f is equipped with a fixed input domain, I, and an output range, O, so that $f : I \to O$ and (overloading f) $O = fI$ and $I = f^{-1}O$.

We take a probabilistic view of the behaviour of f. We overload I and O to also represent random variables equipped with probability distributions, σ_I and σ_O respectively. Shannon [25] measured the information content, or entropy, of a random variable X with probability distribution p as follows.

DEFINITION 7. *Entropy of a random variable.*

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

Since random variable O is completely determined by f's action on I all the information in O stems from I [9] so $\mathcal{H}(I) - \mathcal{H}(O)$ is the amount of information destroyed by f. This is conditional entropy, the entropy of I conditional on knowledge of O, in the deterministic case. To emphasise the role that this loss of information is playing we also call this quantity *Squeeziness*. If the function is one to one there are no collisions and the Squeeziness of the function is 0 since $\mathcal{H}(I) = \mathcal{H}(O)$.

DEFINITION 8. *Squeeziness. The* Squeeziness *of total function $f : I \to O$, $Sq(f)$, is defined as the loss of information after applying f to I*

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

Note that squeeziness considered as a function takes two arguments, a domain (actually a random variable in the values of the domain) and a function applied to that domain.

The partition property of entropy [12] allows us to reformulate Squeeziness in a more useful way. Let $f^{-1}o$ be the random variable in the inverse image of $o \in O$. The inverse images of elements of O partition I. For each $o \in O$, $\sigma_O(o) = \sum_{i \in f^{-1}o} \sigma_I(i)$ so σ_O is the probability distribution for the random variable in the partitions induced by the inverse images. These inverse images partition the input space. By the partition property

$$\mathcal{H}(I) = \mathcal{H}(O) + \sum_{o \in O} p(o)\ \mathcal{H}(f^{-1}o)$$

hence

$$Sq(f, I) = \sum_{o \in O} p(o)\ \mathcal{H}(f^{-1}o)$$

The RHS of the equation is a weighted sum of terms and $\mathcal{H}(f^{-1}o)$ is the amount of information contained in (the random variable in) the set of elements mapped to a single output.

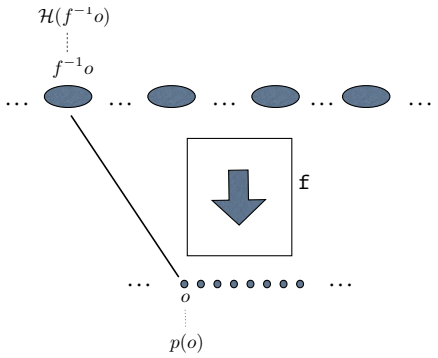

Figure 1: Loss of information: squeezing inverse images of outputs.

In what follows whenever the distribution on a set is not an induced one, e.g. it is a distribution on inputs, we always use a uniform distribution. This corresponds to the Maximum Entropy Principle and can be viewed as producing metrics in which inputs have equal weight.

4. FAILED ERROR PROPAGATION (FEP) AND CONDITIONAL ENTROPY

We argue that a very useful way of looking at FEP is to see it in terms of loss of information. This has been hinted at in studies by Masri, Woodward, and others without any strong conclusions [20, 31]. The nature of testing software, i.e. lack of knowledge of the error-free program, necessarily makes any analysis approximate rather than formal. In this case

the proof of our analytical pudding will be experiment rather than proof based. To overcome our ignorance of the ideal, error free program, from now on referred to as the "ghost" program, we make the following two strong assumptions.

ASSUMPTION 1. *There is a single error in the program under test.*

This is a fairly common, simplifying assumption. In order to state the second assumption we must first compare execution of a test input using the "ghost" program with execution of the same input using the Implementation Under Test (IUT).

Suppose the program we intended to write, the "ghost" program, is program P but the program we actually wrote, the IUT, is another program which we will call P'. P is a perfect oracle for P' which exhibits not only the desired input-output behaviour of P' but allows us to examine the *desired* internal states of P'. The difference between P and P' is that P' may contain faults. Let us assume that there is only one fault in P' and that it occurs in a single structural component, C' which corresponds to its fault-free version, C, in P. This is a scenario similar to mutation testing.

Now consider a test input, t, and the execution of each program on t. This is the situation illustrated by Figure 2. In the CFG corresponding to each program we can break the execution path into three parts: the upper path that precedes entry to the structural component, the structural component, and the lower path that succeeds the component. Since we assume that there is a single fault the respective upper paths will be the same, i.e. A and A' in Figure 2 are the same. Clearly C and C' are not the same and, in general, the succeeding or lower paths, B and B' are not the same. Let us assume that c is the path through C and that C has a final node, n, which is a state update node. In this case there is a single outgoing edge, e, and we expect c', C', n' and e' to play the corresponding roles in P'. For a covering path $\pi = A'.c'.B'$ in P' when describing the *upper* path we will mean $\pi_u = A'.c'$ and by the *lower* path we mean $\pi_l = B'$.

The execution semantics in P is then $[\![A.c.B]\!]^{ex}_{e}(t)$ and in P' is $[\![A'.c'.B']\!]^{ex}_{e'}(t)$. The path semantics in P is $[\![A.c.B]\!]^{pa}_{e}$ and in P' is $[\![A'.c'.B']\!]^{pa}_{e'}$. Finally, under the assumption that the final node of C (and C') is a state update node, the CFG collecting semantics at n and n' respectively and the program collecting semantics at e and e' respectively are the same (respectively). That is $[\![e]\!]^{cfg} = [\![n]\!]^{pr}$ and $[\![e']\!]^{cfg} = [\![n']\!]^{pr}$. In Figure 2, e is identified with program point pp and e' with program point pp'.

We will use these notations to frame hypotheses and describe the experiments in Sections 5 and 6.

To return to Figure 2, it may be possible that the states associated with each edge in the execution semantics are the same, i.e. $[\![A.c]\!]^{ex}_{e}(t) = [\![A.c']\!]^{ex}_{e'}(t)$, but in general these will be different, corresponding to the Infection phase of the PIE scenario. However when FEP occurs we have that the output, o, is the same in each case.

Laski et alia observed that it is the behaviour of the subgraph whose input state corresponds to the edge e' in Figure 2 and which may be described as the subgraph reachable from the target node of e' which is somehow failing to Propagate the Infection to the output [17]. In P and P' these respective subgraphs are labelled Q and Q'. It is the joint behaviour of these that causes FEP. Laski and others further noted that these subgraphs are exactly the same except in the case that the component C' is within a loop. Then C may occur many times in Q corresponding to C' occurring many times in Q' but the CFG context in which they occur will be the same. In the case that the fault occurs in a node, n, which is a control expression there is no unique outward edge as per our earlier assumption and the subgraph is not a well formed CFG. This case did not occur in the paper by Laski et alia as they assumed that the fault was within a well formed program construct.

Suppose that the two subgraphs are the same ($Q = Q'$), then the collision where two different inputs produce the same output is an example of loss of information (conditional entropy) between inputs and outputs as discussed in a previous paper [7] and above in Section 3.3, so even if they are not the same but their information flow behaviour is very similar we can use the information flow behaviour of Q' to estimate how likely it is that FEP occurs. *This is the key idea in this paper* and our second assumption.

ASSUMPTION 2. *The sub programs Q and Q' following the error point in the "ghost" program and the IUT are essentially the same from an information flow quantity perspective.*

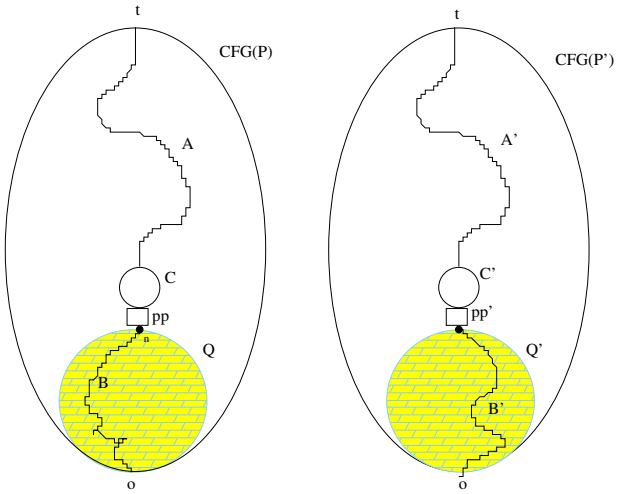

Figure 2: FEP scenario.

5. THE RESEARCH QUESTION

Our long term aim (beyond the scope of this paper) is to produce a set of lightweight, information flow based metrics which can support both coverage based testing and mutation based testing in generating test suites that minimally suffer from FEP. As Masri et alia have observed, different programs suffer from coincidental correctness to different extents [19]. However, we can attempt to optimise for a particular program. In what follows we consider the question of useful correlations from a coverage testing perspective.

There is only one research question:

RESEARCH QUESTION 1. *What are the useful correlations between the conditional entropies of different information flow channels in the IUT and the probability of FEP for a given, erroneous, program construct?*

In what follows we propose six answers to this research question in the form of hypotheses. In subsequent sections we will evaluate these answers experimentally. It will be useful to refer to Figure 2 when interpreting the hypotheses. The statements of the hypotheses refer to the following conventions:

5.1 Hypothesis 1

Consider an incorrect program construct, C' in an IUT P' containing program point pp' corresponding to edge e' as above, immediately following C'. Let Σ_I be the set of inputs to P' and let Q' be the sub program of P' reachable from pp' and the collecting semantics at pp' be the set of states $\Sigma_{pp'} = [\![e']\!]^{cfg}$. Let $[\![Q']\!]$ be the functional semantics of Q'.

HYPOTHESIS 1. *There is a correlation between the probability of FEP for all input states whose execution path includes e' and*

$$sq([\![Q']\!], [Q']([\![Q']\!]\Sigma_{pp'}))$$

We unpack and motivate the metric. Laski et alia have observed that failed error propagation is due to the behaviour of Q'. We propose that, given our assumptions, the more squeezy $[\![Q']\!]$ is on states that get mapped to the output of Q' applied to $\Sigma_{pp'}$, i.e. the higher the conditional entropy of $[\![Q']\!]$ on that domain, the higher the probability that any internal state chosen at pp' suffers from FEP. But what should we estimate as the domain for the function $[\![Q']\!]$? It has to be larger than $\Sigma_{pp'}$ as it needs to contain states that could occur at pp in P, the "ghost" program, i.e. states in Σ_{pp}. One way of estimating $\Sigma_{pp} \cup \Sigma_{pp'}$ is to consider the weakest precondition of the outputs of the IUT for all execution paths through pp' under $[\![Q']\!]$. This is the domain $[Q'](\![Q']\!]\Sigma_{pp'})$.

The consequence of a strong correlation here is that the squeeziness based metric would tell the tester whether or not it is necessary to use optimisation to minimise the probability of FEP.

5.2 Hypothesis 2

Consider the same setup as for the previous hypothesis. Similarly, let R' be the sub program of P' which is *backwardly* reachable from pp' so that $\Sigma_{pp'}$ is also the set of outputs from applying $[\![R']\!]$ to Σ_I.

HYPOTHESIS 2. *There is a correlation between the probability of FEP for all input states that reach pp' via the execution of R' and*

$$sq([\![R']\!], [R']\Sigma_{pp'}) + sq([\![Q']\!], [Q']([\![Q']\!]\Sigma_{pp'}))$$

except when $sq([\![Q']\!], [Q']([\![Q']\!]\Sigma_{pp'})) = 0$.

This is the same as hypothesis 1 but in addition we consider the sub program R'. This may potentially squeeze multiple input states in Σ_I onto states in $\Sigma_{pp'}$ that in turn suffer from FEP, multiplying the effect of Q'. To account for this we add the conditional entropy of $[\![R']\!]$ on the input states that get mapped to $\Sigma_{pp'}$. If $[\![Q']\!]$ has zero squeeziness on $[Q'](\![Q']\!]\Sigma_{pp'})$ there should be no possibility of FEP and no need to consider the multiplier effect.

The consequences of a strong correlation are the same as for Hypothesis 1. In this case we are merely interested in which correlation is the stronger. In the next two hypotheses we examine how we can provide an optimisation when the squeeziness metric is relatively high.

5.3 Hypothesis 3

Consider an execution path, π, in P' that covers the faulty component C'. In general there may be many inputs to P' that follow π and correspond to states at both pp' and the exit point for P'. Let the outputs that reach the exit point for P' along π be $[\![\pi]\!]\Sigma_I$.

HYPOTHESIS 3. *There is a correlation between the probability of FEP for all states that reach pp' via execution along π, i.e. $[\![\pi]\!]^{pa}_{e'}$, and*

$$sq([\![Q']\!], [Q']([\![\pi]\!]\Sigma_I))$$

Again, we unpack and motivate the conditional entropy metric. Previously we considered all states that reach the program point pp' immediately after executing the potentially faulty program component C'. Here we consider the states that reach the program point only via a single covering path for C', π. Examining Figure 2, an input to both P and P' will reach pp and pp' respectively via the same upper path (modulo C/C') but may have different lower paths. Under Assumption 2, both of the lower paths are paths in Q' so the degree of FEP for inputs to P' that reach $[\![\pi]\!]^{pa}_{e'}$ depends on the degree to which Q' is colliding states at pp and pp' to produce states in $[\![\pi]\!]\Sigma_I$. We estimate the states in $\Sigma_{pp} \cup \Sigma_{pp'}$ that are mapped onto $[\![\pi]\!]\Sigma_I$ by considering the weakest precondition for Q' but this time with a post condition of $[\![\pi]\!]\Sigma_I$.

The consequence of a strong correlation in this case is that it gives us a means to rank covering paths for C' using squeeziness. We can choose the least squeezy path and have confidence that, by using that path to generate a test input that covers C', we have maximised, or at least improved, the probability that our choice of test input will produce a failing test output in the case that C' has a bug.

5.4 Hypothesis 4

Let a path π covering a construct C' be expressed as the concatenation of two paths so that $\pi = \pi_u\pi_l$ as above in Section 4 , where π_u is the *upper path* that terminates at pp' (e') and π_l is the *lower path* that follows pp' (the part of π beginning at the target node of e'). There is the possibility that π_u squeezes inputs onto states at pp' which magnifies the degree of FEP caused by the squeeziness of Q' on $\Sigma_{pp'} \cup \Sigma pp$. In a way similar to Hypothesis 2 we add the conditional entropy of the upper path to improve the correlation.

HYPOTHESIS 4. *There is a correlation between the probability of FEP for all states that reach pp' via execution along π, i.e. $[\![\pi]\!]^{pa}_{e'}$, and*

$$sq([\![\pi_u]\!], [\pi_u][\![\pi]\!]^{pa}_{e'}) + sq([\![Q']\!], [Q']([\![\pi]\!]\Sigma_I))$$

As in Hypothesis 2, our aim is to test whether the addition of this upper path conditional entropy improves the correlation.

5.5 Hypothesis 5

In a previous paper we speculated that the squeeziness of π_l on $\Sigma_{pp'}$ would be correlated with the probability of fault masking for inputs that reach pp' [7].

HYPOTHESIS 5. *There is a correlation between the probability of FEP for all input states that reach pp' via execution along π and*

$$sq([\![\pi_l]\!], [\![\pi]\!]^{pa}_{e'}))$$

The consequences of this correlation are that we would only need to rank covering paths for a construct on the basis of the squeeziness of the lower path on the states that occur at pp' along path π. In comparison to metrics in earlier hypotheses this would be comparatively cheap to estimate.

5.6 Hypothesis 6

We know that for a function f and domain D where $sq(f, D) = 0$ that f must be one to one. So we expect that if a squeeziness metric is zero then f must propagate error states in D to the output of the function without any collisions, i.e. that the probability of FEP is zero. That is the theory. However Assumption 2 may weaken this so we need to test the hypothesis experimentally.

HYPOTHESIS 6. *Let $\epsilon > 0$ be a small number arbitrarily close to 0. Whenever $sq([\![Q']\!], [Q']([\![\pi]\!]\Sigma_I)) \leq \epsilon$ then $p(FEP) \leq \epsilon$.*

The consequence of this hypothesis being validated is that a squeeziness close to zero for a path means that we don't need to rank the covering paths but can simply use that path to generate a test input to cover the program construct.

6. EXPERIMENTAL SETUP

6.1 Subjects and Mutants

Three sources of subject programs are shown in Table 1 where we aggregate lines of code for each project, with all programs being written in C. The *toy* project contains 17 small programs that we implemented based on designs that aimed to demonstrate squeeziness. The other subject programs were taken from two real-world projects: the R project [14] for statistical computing and graphics, and the open source statistical package GRETL (Gnu Regression, Econometrics and Time-series Library) [10].

Table 1: Projects under investigation

Project	Function	total LoC	Mutants
Toy	17	810	383
R	10	221k	953
GRETL	3	286k	72

The two real-world projects contain many functions and so we chose entrance functions: those directly called by a user or a program from outside of the project. For *R*, as shown in Column 2 of Table 2, we selected the 10 functions with the most Lines of Code (LoCs) from the *nmath* library of *R*, a C Library of special mathematical functions. However, to simplify the experiments, we did not use functions that contained array variables. To add variety, another three subject programs were chosen from the *cephes* library of *GRETL* (see Table 2).

The functions from *R* and *GRETL* require other functions from their libraries. We therefore formed subject programs by (recursively) including the required functions, with Column 3 of Table 2 giving the numbers of functions involved. Columns 4 and 5 give the LoC and physically executable lines of code (SLoC) of the subject programs.

We generated different versions of programs by seeding a fault into each original subject program as done in mutation testing [16]. The faults were introduced by using the C mutation operator *OAAN* [3] that replaces an arithmetic operator with another, for example, - with +, or / with +. The mutation tool *SMT-C* was used to generate the mutants (mutated programs) and to run the mutation analysis [13]. Some of the mutants could not be compiled and so were not used in the experiments. To calculate FEP, both strong and weak mutation analysis were applied[1]. Finally, several subject programs (*rhyper*, *ptukey*, *qgamma*, *psi* and *qt*) led to too many mutants and in these cases we randomly selected 100 mutants. The numbers of mutants used in the experiment are given in Table 1.

Table 2: Real world statistical subject programs

Project	Function	C Files	LoC	SLoC
R	bratio	4	1667	1573
	rhyper	2	338	260
	gamma_cody	2	137	116
	ptukey	7	1360	674
	qgamma	14	2397	1312
	psi	2	261	119
	pnorm_both	1	315	178
	pnchisq_raw	1	275	181
	gammafn	5	527	264
	qt	1	234	124
GRETL	i0	2	270	108
	k0	5	688	267
	unity	3	335	147

6.2 Experimental Design

In the experiments, randomly chosen inputs were sampled from a uniform distribution. This allowed us to estimate quantities. For significance we have relied on a sample size which is relatively large over all programs and can be viewed as "all inputs" for a restricted input domain.

As all subject programs have numeric inputs, the *RngPack* library was used to generate the random numbers [2]. Initially, we generated inputs from the complete ranges of the input parameters but we found that most inputs led to special numbers (0, *NAN* and *INFINITE*) as output. To address this problem, for each program we limited the input domain from which inputs were chosen based on the first level guards of that program. This led to smaller input domains but useful test cases. For example, inputs for *gamma_cody* were drawn from $[-200, 200]$.

We then ran strong/weak mutation analysis using SMT-C and the randomly generated inputs. As the weak mutation analysis functionality of SMT-C is implemented based on the GNU Debugger (GDB) [1], it was possible to use GDB commands to extract the runtime program states, and this allowed us to extend SMT-C to support information flow analysis. We executed each subject program and each of its mutants with the same 5000 inputs and recorded the state after the mutation point (at program point pp in the original program P and pp' in the mutant P' in Figure 2), the state at the end of the program (o in the original program P and o' in the mutant P' in Figure 2), and the execution path taken by the mutant. We ignored any inputs that led to invalid

[1]See section 2 for a description of weak and strong mutation testing

outputs, such as those that are not a number, are infinite or that cause an exception. The executions of the more than 7 million test cases generated more than 7 gigabytes of raw result files. Given mutant P', the following shows how we calculated the probability of fault masking p(FEP) on the inputs used.

$$\text{p(FEP)} = \frac{\begin{array}{c}\#\text{ of tests that weakly kill } P' \\ \text{but do not strongly kill } P'\end{array}}{\#\text{ of tests that weakly kill } P'}$$

Note that the way we counted the total number of tests in the denominator varied between experiments. In EXP1 and EXP2 we counted the tests that reach pp' via any execution path through C'. In experiments EXP3, EXP4 and EXP5 we counted the number of tests that reach pp' by following a single execution path to output.

For a test input to weakly kill a mutant, it must lead to different program states for the mutant and the original program after the mutation point (in Figure 2 the states at pp in P and pp' in P' should be different). A test input strongly kills a mutant if it leads to this difference in program state propagating to the program's output (in Figure 2 o in P and o in P' are different). Therefore, p(FEP) computes the proportion of tests that cause a different program state after the mutation point at pp in P and pp' in P' but do not lead to a different output.

We classified tests according to whether they are able to propagate the seeded fault to the output. The tests that successfully propagate faults have the property EP (error propagation). The rest of the tests suffer from failed error propagation (FEP) and other coincidental correctness (CC1). Tests classified as CC2 suffered from anomalies. Table 3 lists the proportion of tests that have these properties. The significant statistic is that approximately 10% of tests across all program-mutant pairs suffered from FEP.

Table 3: The proportion of randomly generated tests for all subject programs that are weakly and strongly killed.

	Weakly Killed	Strongly Killed	Proportion
EP	Yes	Yes	84.73 %
FEP	Yes	No	9.85 %
CC1	No	No	4.89 %
CC2	No	Yes	0.44 %

Given our complete knowledge in the mutation testing scenario of the experiments we can replace $[Q'](\llbracket Q' \rrbracket \Sigma_{pp'})$ with the quantity it estimates, namely $\Sigma_{pp} \cup \Sigma_{pp'}$ in EXP1 and EXP2. Similarly we can use the appropriate subset of $\Sigma_{pp} \cup \Sigma_{pp'}$, calculated by direct examination of the data, to replace its estimation, $[Q'](\pi\Sigma_I)$ in EXP3, EXP4 and EXP5. This gives a maximal correlation, useful when seeking to compare correlations and rate the strength of correlations.

We now outline the experiments performed.

6.2.1 Experiment 1 (EXP1)

This experiment explored the strength of the correlation suggested in Hypothesis 1 by estimating the correlation between the probability of FEP for inputs reaching pp' and $sq(\llbracket Q' \rrbracket, \Sigma_{pp} \cup \Sigma_{pp'})$.

For each mutant, $sq(\llbracket Q' \rrbracket, \Sigma_{pp} \cup \Sigma_{pp'})$ can be calculated as follows. First, we set

$$s = \frac{1}{|\Sigma_{pp} \cup \Sigma_{pp'}|}$$

Then we set $S = |\Sigma_{pp} \cup \Sigma_{pp'}| s \log_2(s) = \log_2(s)$. Given output o, the probability for o given $\Sigma_{pp} \cup \Sigma_{pp'}$ is

$$t(o) = \frac{\#\text{ of states in } \Sigma_{pp} \cup \Sigma_{pp'} \text{that lead to } o}{|\Sigma_{pp} \cup \Sigma_{pp'}|}$$

and consequently $T = \sum_o t(o) * \log_2(t(o))$ and the squeeziness for Q' on $\Sigma_{pp} \cup \Sigma_{pp'}$ is $sq(\llbracket Q' \rrbracket, \Sigma_{pp} \cup \Sigma_{pp'}) = -S + T$

6.2.2 Experiment 2 (EXP2)

This experiment explored the strength of the correlation suggested in Hypothesis 2. The experiment is the same as EXP1 with the difference being that if $sq(\llbracket Q' \rrbracket, \Sigma_{pp} \cup \Sigma_{pp'}) \neq 0$ we add to it $sq(\llbracket R' \rrbracket, [R']\Sigma_{pp'})$, i.e. the squeeziness of R' on the inputs that R' maps to states at pp'.

For each mutant, the squeeziness of R' on the domain can be calculated as follows. The probability assigned to the inputs is $I = \log_2(|\Sigma_I|)$ (recall that Σ_I is the set of inputs used). The probability assigned to a state $\rho \in \Sigma_{pp'}$ is:

$$u(\rho) = \frac{\#\text{ of inputs that get mapped to } \rho \text{ by } R'}{|\Sigma_I|}$$

and we let $U = \sum_\rho u(\rho) * \log_2(u(\rho))$. Then the squeeziness of R' is: $sq(\llbracket R' \rrbracket, [R']\Sigma_{pp'}) = -I + U$.

If the squeeziness of Q' is non-zero, then the squeeziness of R' is added to it.

6.2.3 Experiment 3 (EXP3)

For a given path π, which executes the component of interest, p(FEP) for states occurring on the path at pp' correlates with $sq(\llbracket Q' \rrbracket, [Q'](\pi\Sigma))$. Given path π we will let $\Sigma^\pi_{pp'}$ and Σ^π_{pp} be the sets of states occurring on the path at pp' and pp respectively when following π in P' and its equivalent in P respectively.

For each path π in a mutant, associated values of p(FEP) and the squeeziness of Q', $sq(Q', [Q'](\pi\Sigma))$, can be calculated using an approach similar to that in the description of experiment EXP1 except that the values used are specific to π. Thus, we assign the probability

$$s = \frac{1}{|\Sigma^\pi_{pp} \cup \Sigma^\pi_{pp'}|}$$

Then we set $S = \sum s * \log_2(s)$. The probability for the corresponding outputs for π is given by:

$$t(o) = \frac{\#\text{ of states in } \Sigma^\pi_{pp} \cup \Sigma^\pi_{pp'} \text{ that lead to } o}{|\Sigma^\pi_{pp} \cup \Sigma^\pi_{pp'}|}$$

and consequently, $T = \sum_o t(o) * \log_2(t(o))$. The squeeziness for Q' with respect to π at $\Sigma^\pi_{pp} \cup \Sigma^\pi_{pp'}$ is then given by: $sq(\llbracket Q' \rrbracket, \Sigma^\pi_{pp} \cup \Sigma^\pi_{pp'}) = -S + T$.

6.2.4 Experiment 4 (EXP4)

This experiment explored the strength of the correlation suggested in Hypothesis 4. The experiment is the same as EXP3 except that we add to $sq(\llbracket Q' \rrbracket, \Sigma^\pi_{pp} \cup \Sigma^\pi_{pp'}))$ the squeeziness of the upper path, π_u, on the inputs to the program that follow execution path π. The latter can be expressed as $sq(\llbracket \pi_u \rrbracket, [\pi](\llbracket \pi \rrbracket \Sigma))$ since $[\pi_u]\llbracket \pi \rrbracket^{pa}_{e'} = [\pi](\llbracket \pi \rrbracket \Sigma)$.

Let $\Sigma_\pi = [\pi](\llbracket\pi\rrbracket\Sigma)$. The information content of the inputs that travel down path π is $I = \log_2(|\Sigma_\pi|)$. The probability assigned at pp' is:

$$r(\rho) = \frac{\#\text{ of times that } \exists s \in \Sigma_\pi \,.\, \llbracket\pi\rrbracket(s) = \rho}{|\Sigma_\pi|}$$

and this leads to the term $U = -\sum_\rho r(\rho) * \log_2(r(\rho))$. Then the squeeziness of π_u is given by $sq(\llbracket\pi_u\rrbracket, \Sigma_\pi) = I - U$.

6.2.5 Experiment 5 (EXP5)

This experiment assessed the correlation between the probability of FEP on states in $\llbracket\pi\rrbracket^{pa}_{e'}$ and the squeeziness of π_l on these states.

For each path π in mutant P', p(FEP) for states that occur on π at pp', $\llbracket\pi\rrbracket^{pa}_{e'}$, can be calculated as in Experiments EXP3 and EXP4. To calculate squeeziness $sq(\llbracket\pi_l\rrbracket, [\pi_l](\pi\Sigma))$ we have to calculate the probability distributions on $\llbracket\pi\rrbracket^{pa}_{e'}$ at pp' and $\llbracket\pi\rrbracket\Sigma_I$ at the end of the program. To calculate the latter, let $\Sigma_\pi = [\pi](\llbracket\pi\rrbracket\Sigma_I)$. It is sufficient to determine, for each output o, how many inputs that follow π lead to o.

The probability assigned to o in $\llbracket\pi\rrbracket\Sigma_I$ for a given path is:

$$m(o) = \frac{\#\text{ of inputs from } \Sigma_\pi \text{ that lead to } o}{|\Sigma_\pi|}$$

and we set $M = -\sum_o m(o) * \log_2(m(o))$. The probability assigned to state ρ at pp' for π is:

$$l(\rho) = \frac{\#\text{ of inputs from } \Sigma_\pi \text{ that lead to } \rho}{|\Sigma_\pi|}$$

and we set $L = -\sum_\rho l(\rho) * \log_2(l(\rho))$. Then the squeeziness is calculated as: $sq(\llbracket\pi_l\rrbracket, [\pi_l](\pi\Sigma)) = L - M$.

6.2.6 Experiment 6 (EXP6)

We examined various upper bounds "near zero" and found the maximum observed value for $sq(\llbracket Q'\rrbracket, \Sigma_{pp} \cup \Sigma_{pp'}))$ less than the bound and its corresponding value for p(FEP) using the pairs calculated for EXP1. We looked at a small sample of three bounds: 0.1, 0.01 and 0.001, but examined a large number of pairs for each bound.

7. RESULTS

The experiments, inevitably, have limitations. We did not use a variety of mutation operators that, for example, delete statements, replace boolean relations with others, or replace boolean subexpressions with true or false. However, our approach is independent of types of faults as it is semantics based and considers program points and incorrect states. We only generated mutants with a single fault while in practice programs may contain multiple faults but this is a common assumption in testing. We aim to address this in future work.

We sampled inputs rather than considering all inputs. This was the only way to make the experiments practical. This necessity may serendipitously be the foundation of a sampling approach to estimating squeeziness metrics in the future. Although we did not consider formal statistical guarantees, the sample size across all programs was large.

Information from both pp from P and pp' from P' was used, when in practice we would only have pp' from P'. When testing a program P', p(FEP) can never be known in practice as we don't have P. Since our objective in this paper was to use strength of correlation to find the most suitable metrics using knowledge of pp was not a drawback.

Table 4: Spearman's Rank Correlation Coefficient for all programs.

Experiment	Correlation
EXP1	0.715267
EXP2	0.699165
EXP3	0.955647
EXP4	0.948299
EXP5	0.031510

Table 5: Spearman's Rank Correlation Coefficient for statistical programs.

Experiment	Correlation
EXP1	0.974459
EXP2	0.974459
EXP3	0.998526
EXP4	0.998526
EXP5	-0.001361

Consider the correlations found in Experiments 1-4. The experiments computed the different metrics and Table 4 gives the Spearman's Rank Correlation Coefficient for all programs. In order to understand the contributions from the small set of toy programs we wrote ourselves and the real world programs we looked at the correlations for these two groups separately. Table 5 gives the results for the statistical programs, and Table 6 gives the results for the small programs. Interestingly, we have very strong correlations for Experiments 1-4 and these are particularly strong for the larger, real world, programs. Experiments 2 and 4 had lower correlation values than Experiments 1 and 3, suggesting that the important correlations are with squeeziness of Q' on different domains and that contributions from the upper program are not significant, in fact retrograde. In contrast, Experiment 5 did not reveal a correlation, rather invalidating the suggestion of this metric in our IPL paper [7]. These results give a very strong correlation for between information theoretic metrics and both FEP and FEP along a particular path.

We reproduce here the three plots of rank correlations corresponding to the table entries for Experiment 2. Figure 3 plots the ranks given to squeeziness and p(FEP) when calculating Spearman's Rank Correlation Coefficient. Figure 4 plots the ranks given to squeeziness and p(FEP) for the statistical programs. Figure 5 plots the ranks given to squeeziness and p(FEP) for small programs.

Table 6: Spearman's Rank Correlation Coefficient for small programs.

Experiment	Correlation
EXP1	0.705367
EXP2	0.686284
EXP3	0.761889
EXP4	0.666140
EXP5	0.005787

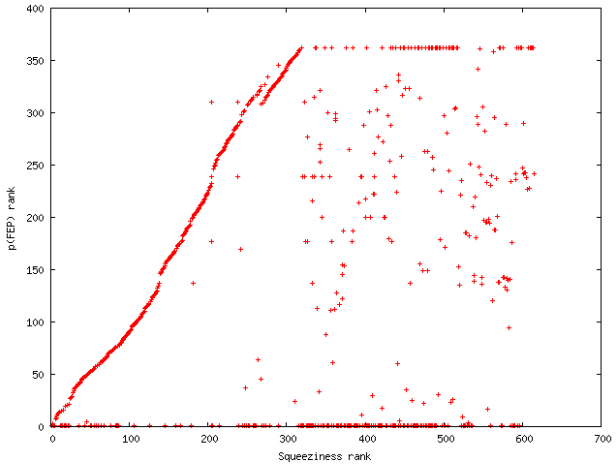

Figure 3: The rank correlation of p(FEP) and Squeeziness for all programs (EXP2).

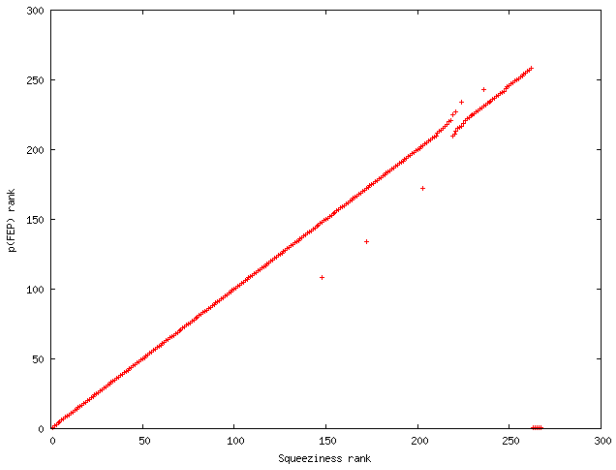

Figure 4: The rank correlation of p(FEP) and Squeeziness for statistical programs (EXP2).

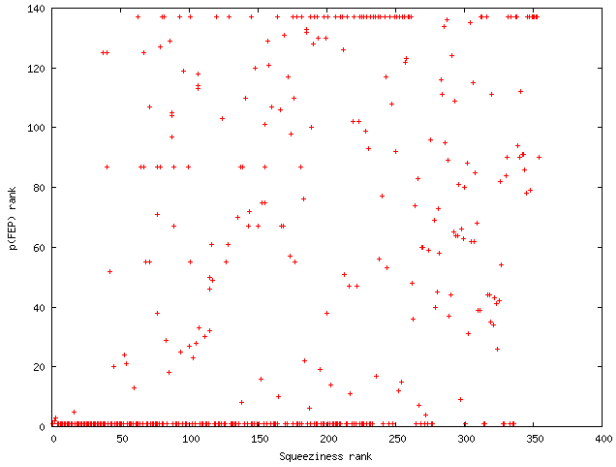

Figure 5: The rank correlation of p(FEP) and Squeeziness for small programs (EXP2).

Table 7: Maximum p(FEP) for all programs

sq(Q') Range	Max sq(Q)	Max p(FEP)
≤ 0.1	0.090683	0.090683
≤ 0.01	0.001120	0.001120
≤ 0.001	0.000800	0.000200

The plots show that the strong correlation for EXP2 is in the most part derived from the larger, real world statistical programs.

Table 7 gives the maximum values for squeeziness of Q and p(FEP) for a given "small" bound on $sq(Q)$ values over all programs using information from EXP1. The results validate theory and could be used to identify program constructs in an implementation under test that are highly unlikely to suffer from FEP.

8. CONCLUSIONS AND FURTHER WORK

Our analysis and the results of our experiments have shown that we can interpret Failed Error Propagation during software testing using conditional entropy based metrics on the Implementation Under Test. This is a novel use of Quantified Information Flow, a concept whose applications have to date been in the area of secure information flow [8, 15].

An enormous vista of possible future work now beckons. Having identified useful metrics, the next task is to repeat the experiments using estimates of the weakest preconditions that appear in the metrics. We expect that the rank correlations will be weaker but still highly significant. The success of these experiments would put the utility of the approach beyond doubt. Beyond that lies the problem of estimating the metrics. Estimating entropy for the absolute values used in the final experiment will be more difficult but not impossible [5].

An interesting challenge in the long run would be application in a concolic testing scenario such as that offered by *Pex* [27]. We believe this paper lays the foundation for significant future improvement in test suite effectiveness.

9. REFERENCES

[1] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/, Accessed in 2013.

[2] RngPack 1.1a. http://www.honeylocust.com/RngPack/, Accessed in 2013.

[3] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the C programming language. Technical report, Department of Computer Sciences, Purdue University, 1989.

[4] T. Apiwattanapong, R. A. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006)*, pages 137–146, Windsor, UK, 2006. IEEE.

[5] K. Chatzikokolakis, T. Chothia, and A. Guha. Statistical measurement of information leakage. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 390–404. Springer, 2010.

[6] J. Chen, Q. Li, J. Zhao, and X. Li. Test adequacy criterion based on coincidental correctness probability. In *the Second Asia-Pacific Symposium on Internetware*, Internetware '10, pages 20:1–20:4, Suzhou, China, 2010. ACM.

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

[8] D. Clark, S. Hunt, and P. Malacaria. Quantitative analysis of the leakage of confidential data. In A. D. Pierro and H. Wiklicky, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier, 2002.

[9] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321 – 372, 2007.

[10] A. Cottrell, R. Lucchetti, et al. GNU regression, econometrics and time-series library. http://gretl.sourceforge.net, Accessed in 2013.

[11] P. Cousot and R. Cousot. *Building the Information Society*, chapter Basic Concepts of Abstract Interpretation. Springer US, 2004.

[12] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley Interscience, 1991.

[13] H. Dan and R. M. Hierons. SMT-C: A Semantic Mutation Testing Tools for C. In *the Fifth International Conference on Software Testing, Verification and Validation*, pages 654–663, Montreal, Canada, Apr. 2012. IEEE.

[14] R. Gentleman, R. Ihaka, et al. The R project for statistical computing. http://www.r-project.org, Accessed in 2013.

[15] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *the Twenty-Sixth Annual Computer Security Applications Conference*, pages 261–269, Austin, Texas, USA, 2010. ACM.

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

[17] J. W. Laski, W. Szermer, and P. Luczycki. Error masking in computer programs. *Software Testing, Verification and Reliability*, 5(2):81–105, 1995.

[18] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In *the 2nd International Workshop on Defects in Large Software Systems*, DEFECTS '09, pages 1–5, Chicago, Illinois, USA, 2009. ACM.

[19] W. Masri and R. A. Assi. Cleansing test suites from coincidental correctness to enhance fault-localization. In *the Third International Conference on Software Testing, Verification and Validation*, pages 165–174, Paris, France, 2010. IEEE.

[20] W. Masri and A. Podgurski. Measuring the strength of information flows in programs. *ACM Transactions on Software Engineering and Methodology*, 19(2), 2009.

[21] H. R. Nielson and F. Nielson. *Semantics with Applications*. Wiley Professional Computing, 1993.

[22] A. J. Offutt and S. D. Lee. How strong is weak mutation? In *the Fourth Symposium on Testing, Analysis, and Verification*, pages 200–213, Victoria, British Columbia, Canada, 1991. ACM.

[23] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[24] R. A. Santelices and M. J. Harrold. Applying aggressive propagation-based strategies for testing changes. In *the Fourth International Conference on Software Testing, Verification and Validation*, pages 11–20, Berlin, Germany, 2011. IEEE.

[25] C. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948. Available on-line at http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html.

[26] M. Staats. The influence of multiple artifacts on the effectiveness of software testing. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10. ACM, 2010.

[27] N. Tillmann and J. de Halleux. Pex–White Box Test Generation for .NET. *Tests and Proofs*, pages 134–153, 2008.

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

[29] X. Wang, S.-C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *the 31st International Conference on Software Engineering*, pages 45–55, Vancouver, British Columbia, Canada, 2009. IEEE.

[30] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 International Conference on Software Engineering*. ACM, May 2013.

[31] M. R. Woodward and Z. A. Al-Khanjari. Testability, fault size and the domain-to-range ratio: An eternal triangle. In *the international symposium on Software testing and analysis*, ISSTA '00, pages 168–172, Portland, Oregon, USA, 2000. ACM.