Ask the Mutants: Mutating Faulty Programs for Fault Localization

Seokhyeon Mun, Yunho Kim, Moonzoo Kim
Department of Computer Science
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea

Shin Yoo
Department of Computer Science
University College London
London, UK

***Abstract*—We present MUSE (MUtation-baSEd fault localization technique), a new fault localization technique based on mutation analysis. A key idea of MUSE is to identify a faulty statement by utilizing different characteristics of two groups of mutants–one that mutates a faulty statement and the other that mutates a correct statement. We also propose a new evaluation metric for fault localization techniques based on information theory, called Locality Information Loss (LIL): it can measure the aptitude of a localization technique for automated fault repair systems as well as human debuggers. The empirical evaluation using 14 faulty versions of the five real-world programs shows that MUSE localizes a fault after reviewing 7.4 statements on average, which is about 25 times more precise than the state-of-the-art SBFL technique Op2 on average.**

I. Introduction

Despite the advance in automated testing techniques, developers still spend a large amount of time to identify the root cause of program failures. This process, called Fault Localization (FL), is an expensive phase in the whole debugging activity [14, 41], because it usually takes human effort to understand the complex internal logic of the Program Under Test (PUT) and reason about the differences between passing and failing test runs. As a result, automated fault localization techniques have been widely studied.

One such technique is Spectrum-based Fault Localization (SBFL). It uses program spectra, i.e. summarized profile of test suite executions, to rank program statements according to their predicted risk of containing the fault. The human developer, then, is to inspect PUT following the order of statements in the given ranking, in the hope that the faulty statement will be encountered near the top of the ranking [35].

SBFL has received much attention, with a heavy emphasis on designing new risk evaluation formulas [6, 19, 36, 39], but also on theoretical analysis of optimality and hierarchy between formulas [26, 37, 38]. However, it has also been criticized for their impractical accuracy and the unrealistic usage model that is the linear inspection of the ranking [29]. This is partly due to the limitations in the spectra data that SBFL techniques rely on. The program spectrum used by these techniques is simply a combination of the control flow of PUT and the results from test cases. Consequently, all statements in the same basic block share the same spectrum and, therefore, the same ranking. This often inflates the number of statements needed to be inspected before encountering the fault.

This paper presents a novel fault localization technique called MUSE, *MUtation-baSEd fault localization technique*, to overcome this problem. MUSE uses mutation analysis to uniquely capture the relationship between individual program statements and the observed failures. It is free from the coercion of shared ranking from the block structure. The basic mutation testing is defined as artificial injection of syntactic faults [3]. However, we focus on what happens when we mutate an already faulty program and, particularly, the faulty program statement. Intuitively, since a faulty program can be repaired by modifying faulty statements, mutating (i.e., modifying) faulty statements will make more failed test cases pass than mutating correct statements. In contrast, mutating correct statements will make more passed test cases fail than mutating faulty statements. This is because mutating correct statements introduces new faulty statements in addition to the existing faulty statements in a PUT. These two observations form the basis of the design of our new metric for fault localization (Section II-A).

We also propose a new evaluation metric for fault localization techniques that is not tied to the ranking model. The traditional evaluation metric in SBFL literature is the Expense metric, which is the percentage of program statements the human developer needs to inspect before encountering the faulty one [25]. However, recent work showed that the Expense metric failed to account for the performance of the automated program repair tool that used various SBFL techniques to locate the fix: techniques proven to rank the faulty statement higher than others actually performed poorer when used in conjunction with a repair tool [30].

Our new evaluation metric, LIL (Locality Information Loss), actually measures the loss of information between the true locality of the fault and the predicted locality from a localization technique, using information theory. It can be applied to any fault localization technique (not just SBFL) and to describe localization of any number of faults.

Using both the traditional Expense metric and the LIL, we evaluate MUSE against 14 faulty versions of five real-world programs. The results show that MUSE is, on average, about 25 times more accurate than Op2, the current state-of-the-art SBFL technique. In addition, MUSE ranks the faulty statement at the top of the suspiciousness ranking for seven out of 14 studied faults, and within the top three for another three faults. In addition, the newly introduced LIL metric also shows that MUSE can be highly accurate, as well as confirming the observation made by Qi et al. [30].

The contribution of this paper is as follows:

- The paper presents a novel fault localization technique called MUSE: *Mutation-based Fault Localization*. It utilizes mutation analysis to significantly improve the precision of fault localization.

- The paper proposes a new evaluation metric for fault localization techniques called *Locality Information Loss* (LIL) based on information theory. It is flexible enough to be applied to all types of fault localization techniques and can be easily applied to multiple faults scenarios.
- The paper presents an empirical evaluation of MUSE using five non-trivial real world programs. The results demonstrate that MUSE is more accurate than widely studied SBFL techniques such as Jaccard, Ochiai, and Op2. The results from the empirical evaluation show that MUSE improves upon the best known SBFL technique by 25 times on average and ranks the faulty statement within the top 3 suspicious statements for 10 out of 14 subject program versions.

The remainder of this paper is organized as follows. Section II describes the mutation-based fault localization technique to precisely localize a fault. Section III explains the new evaluation metric LIL based on information theory. Section IV shows the experiment setup for the empirical evaluation of the techniques on the subject programs. Section V explains the experiment results regarding the research questions and Section VI discusses the results. Section VII presents related work and Section VIII finally concludes with future work.

II. Mutation-based Fault Localization Technique

Mutation testing [4] evaluates the adequacy of a test suite based on its ability to detect artificlally injected faults, i.e. syntactic *mutations* of the original program. The more of the injected faults are *killed* (i.e. detected) by the test suite, the better the test suite is believed to be at detecting unknown, actual faults.

A. Intuitions

Consider a faulty program P whose execution with some test cases results in failures. We propose to mutate P knowing that it already contains at leasts one fault. Let m_f be a mutant of P that mutates the faulty statement, and m_c one that mutates a correct statement. MUSE depends on the following two conjectures.

Conjecture 1: test cases that used to fail on P are more likely to pass on m_f than on m_c.

The first conjecture is based on the observation that m_f can only be one of the following three cases:

1) **Equivalent mutant** (i.e. mutants that syntactically change the program but not semantically), in which case the faulty statement remains faulty. Tests that failed on P should still fail on m_f.
2) **Non-equivalent** and **faulty**: while the new fault may or may not be identical to the original fault, we expect tests that have failed on P are still more likely to fail on m_f than to pass.
3) **Non-equivalent** and **not faulty**: in which case the fault is fixed by the mutation (with respect to the test suite concerned).

Note that mutating the faulty statement is more likely to cause the tests that failed on P to pass on m_f (case 3) than on m_c because a faulty program is usually fixed by modifying (i.e., mutating) a faulty statement, not a correct one. Therefore, the number of the failing test cases whose results change to pass will be larger for m_f than for m_c.

In contrast, mutating correct statements is not likely to make more test cases pass. Rather, we expect an opposite effect, which is as follows:

Conjecture 2: test cases that used to pass on P are more likely to fail on m_c than on m_f.

Similarly to the case of m_f, the second conjecture is based on an observation that m_c can be either:

1) **Equivalent mutant**, in which case the statement remains correct. Tests that passed with P should still pass with m_c.
2) **Non-equivalent mutant**: by definition, a non-equivalent mutation on a correct statement introduces a fault, which is the original premise of mutation testing.

This second conjecture is based on the observation that a program is more easily broken by modifying (i.e., mutating) a correct statement than by modifying a faulty statement (case 2). Therefore, the number of the passing test cases whose results change to fail will be greater for m_c than m_f.

To summarize, mutating a faulty statement is more likely to cause more tests to pass than the average, whereas mutating a correct statement is more likely to cause more tests to fail than the average (the average case considers both correct and faulty statements). These two conjectures provide the basis for our MUtation-baSEd fault localization technique (MUSE).

B. Suspiciousness Metric of MUSE

Based on the two conjectures, we now define the suspiciousness metric for MUSE, μ. For a statement s of P, let $f_P(s)$ be the set of tests that covered s and failled on P, and $p_P(s)$ the set of tests that covered s and passed on P. With respect to a fixed set of mutation operators, let $mut(s) = \{m_1, \ldots m_k\}$ be the set of all mutants of P that mutates s with observed changes in test results. After each mutation $m_i \in mut(s)$, let f_{m_i} and p_{m_i} be the set of failing and passing tests on m_i respectively (f_P and p_P defined on P similarly). Given a weight α, the metric μ is defined as follows:

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|}\right) \tag{1}$$

The first term, $\frac{|f_P(s) \cap p_m|}{|f_P|}$, reflects the first conjecture: it is the proportion of tests that failed on P but now pass on a mutant m that mutates s over tests that failed on P. Similarly, the second term, $\frac{|p_P(s) \cap f_m|}{|p_P|}$, reflects the second conjecture, being the proportion of tests that passed on P but now fail on a mutant m that mutates s over tests that passed on P.

		Coverage of Test Cases (x, y)							Jaccard		Ochiai		Op2	
	`int max;` `void setmax(int x, int y){`	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\lvert f_P(s)\rvert$	$\lvert p_P(s)\rvert$	Susp.	Rank	Susp.	Rank	Susp.	Rank
s_1:	`max = -x; //should be 'max = x;'`	●	●	●	●	●	2	3	0.40	5	0.63	5	1.25	5
s_2:	`if(max < y){`	●	●	●	●	●	2	3	0.40	5	0.63	5	1.25	5
s_3:	`max = y;`	●	●		●	●	2	2	0.50	2	0.71	2	1.50	2
s_4:	`if(x*y<0)`	●	●		●	●	2	2	0.50	2	0.71	2	1.50	2
s_5:	`print(''diff.sign'');}`		●			●	1	1	0.33	6	0.50	6	0.75	6
s_6:	`print(max);}`	●	●	●	●	●	2	3	0.40	5	0.63	5	1.25	5
	Test Results	Fail	Fail	Pass	Pass	Pass								

			Test Result Changes							MUSE	
	Statements	Mutants	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\lvert f_P(s) \cap p_m\rvert$	$\lvert p_P(s) \cap f_m\rvert$	Suspiciousness	Rank
s_1:	`max = -x;`	`m1: max -= x-1;`			P→F			0	1	0.46	1
		`m2: max=x;`	F→P	F→P				2	0		
s_2:	`if(max < y){`	`m3: if(!(max<y)){`			P→F	P→F	P→F	0	3	0.09	2
		`m4: if(max==y){`	F→P			P→F		1	1		
s_3:	`max = y;`	`m5: max = -y;`				P→F	P→F	0	2	-0.16	5
		`m6: max = y+1;`				P→F	P→F	0	2		
s_4:	`if(x*y<0){`	`m7:if(!(x*y<0))`				P→F	P→F	0	2	-0.12	4
		`m8:if(x/y<0)`					P→F	0	1		
s_5:	`print(''diff.sign'');}`	`m9:return;`					P→F	0	1	-0.08	3
		`m10:;`					P→F	0	1		
s_6:	`print(max);}`	`m11:printf(0);}`				P→F	P→F	0	2	-0.20	6
		`m12:;}`			P→F	P→F	P→F	0	3		

Fig. 1: Example of how MUSE localizes a fault compared with different fault localization techniques

When averaged over $mut(s)$, they become the probability of test result change per mutant, from failing to passing and vice versa respectively.

Intuitively, the first term correlates to the probability of s being the faulty statement (it increases the suspiciousness of s if mutating s causes failing tests to pass, i.e. increase the size of $f_P(s) \cap p_m$), whereas the second term correlates to the probability of s *not* being the faulty statement (it decreases the suspiciousness of s if mutating s causes passing tests to fail, i.e. increase the size of $p_P(s) \cap f_m$).

Since it is more likely that a passing test case on P will fail on m than a failing test case on P will pass on m (i.e., breaking a program is easier than correcting the program), we expect the average of the second term to be different from that of the first term. In order to balance the two terms, we use the weight α to adjust the average values of the two terms to be the same. Thus, when we subtract the weighted second term from the first term as in Equation 1, we get the baseline of value 0. For a faulty statement, the first term is likely to be larger and the second term is likely to be smaller than for a correct statement.

To adjust the average of both terms, the value of α should be calculated as $\frac{f2p}{|mut(P)| \cdot |f_P|} \cdot \frac{|mut(P)| \cdot |p_P|}{p2f}$. Variable $f2p$ and $p2f$ denote the number of test result changes from failure to pass and vice versa between before and after all mutants of P, the set of which is $mut(P)$. Note that α can be calculated without *a priori* knowledge of the faulty statement.

C. An Working Example

Figure 1 presents an example of how MUSE localizes a fault. The PUT is a function called `setmax()`, which sets a global variable `max` (initialized to 0) with `x` if `x > y`, or with `y` otherwise. Statement s_1 contains a fault, as it should be `max =x`. Let us assume that we have five test cases ($tc1$ to $tc5$): the coverage of individual test cases are marked with black bullets (●). TC_1 and TC_2 fail because `setmax()` updates `max` with the smaller number, `y`. The remaining test cases pass. Thus, $|f_P| = 2$ and $|p_P| = 3$.

First, MUSE generates mutants by mutating only one statement at a time. For the sake of simplicity, here we assume that MUSE generates only two mutants per statement, resulting in a total of 12 mutants, $\{m_1, \ldots, m_{12}\}$ (listed under the "Mutants" column of Figure 1). Test cases change their results after the mutation as noted in the middle column. For example, TC_1, which used to fail, now passes on the two mutants, $m2$ and $m4$.

Based on the changed results of the test cases, MUSE calculates α as $\frac{f2p}{|mut(P)| \cdot |f_P|} \cdot \frac{|mut(P)| \cdot |p_P|}{p2f} = \frac{3}{12 \cdot 2} \cdot \frac{12 \cdot 3}{19} = 0.24$ over 12 mutants ($|mut(P)| = 12$). Since there are three changes from failure to pass, $f2p = 3$ (TC_1 and TC_2 on m_2 and TC_1 on m_4) while $|f_P| = 2$. Similarly, $p2f = 19$ (see the changed results of TC_3, TC_4, and TC_5), while $|p_P| = 3$.

Using $\alpha = 0.24$, MUSE calculates the suspiciousness of s_1 as $\frac{1}{2} \cdot \{(0/2 - 0.24 \cdot 1/3) + (2/2 - 0.24 \cdot 0/3)\} = 0.46$, where $|f_P(s_1) \cap p_{m_1}| = 0$ and $|p_P(s_1) \cap f_{m_1}| = 1$ for m_1 and $|f_P(s_1) \cap p_{m_2}| = 2$ and $|p_P(s_1) \cap f_{m_2}| = 0$ for m_2. MUSE calculates the suspiciousness scores of the other five statements as 0.09, -0.16, -0.12, -0.08, and -0.20. The suspiciousness of the s_1 is the highest at 0.46, which places it at the top of the ranking. In contrast, Jaccard, Ochiai, and Op2 choose s_3 and s_4 as the most suspicious statements, while assigning the 5th rank to the actual faulty statement s_1. The example shows that MUSE can precisely locate certain faults that the state-of-the-art SBFL techniques cannot.

D. MUSE Framework

Figure 2 shows the framework of MUtation-baSEd fault localization technique (MUSE). There are three major stages:

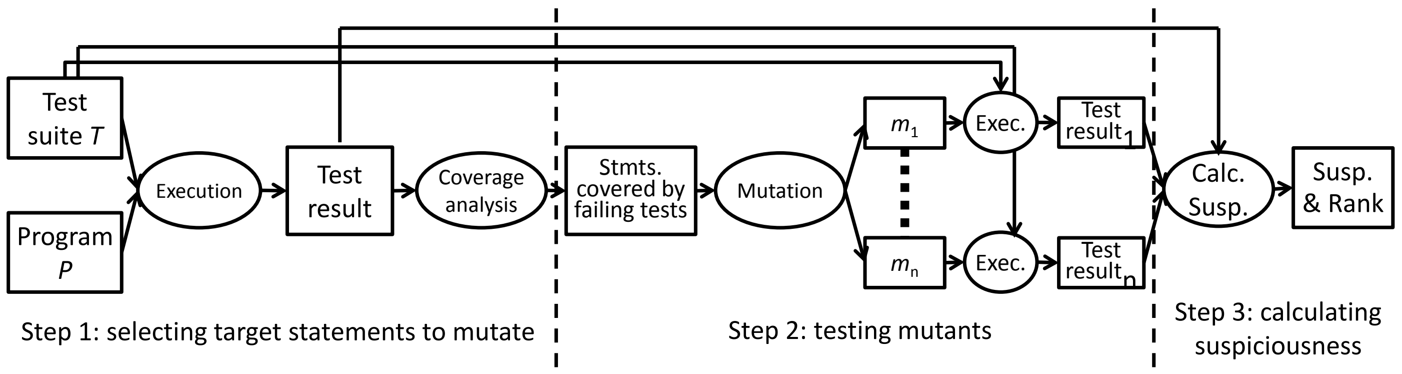

Fig. 2: Framework of MUtation-baSEd fault localization technique (MUSE)

selection of statements to mutate, *testing* of the mutants, and *calculation* of the suspiciousness scores.

Step 1: MUSE receives a target program P and a test suite T. After executing T on P, MUSE selects the target statements, i.e. the statements of P that are executed by at least one failing test case in T. We focus on only these statements as those not covered by any failing tests, can be considered not faulty with respect to T.

Step 2: MUSE generates mutant versions of P by mutating each of the statements selected at Step 1. MUSE may generate multiple mutants from a single statement since one statement may contain multiple mutation points [12]. Consequently MUSE tests all generated mutants with T and records the results.

Step 3: MUSE compares the test results of T on P with the test results of T on all mutants. This produces the weight α, based on which MUSE calculates the suspiciousness of the target statements of P[1].

III. LIL: Locality Information Loss

The output of fault localization techniques can be consumed by either human developers or automated program repair techniques. In SBFL literature, the human consumption model assumes the output format of ranking of statements according to their suspiciousness, which is to be linearly followed by humans until identifying the actual faulty statement. Expense [25] metric measures the portion of program statements that need to be inspected until the localization of the fault. It has been widely adopted as an evaluation metric for FL techniques [18, 26, 39] as well as a theoretical framework that showed hierarchies between SBFL techniques [37, 38]. However, the Expense metric has been criticised for being unrealistic to be used by a human developer directly [29].

In an attempt to evaluate the precision of SBFL techniques, Qi et al. [30] compared SBFL techniques by measuring the Number of Candidate Patches (NCP) generated by GenProg [34] automated program repair tool, with the given localization information.[2] Automated program repair techniques tend to bypass the ranking and directly use the suspiciousness scores of each statement as the probability of mutating the statement (expecting that mutating a highly suspicious statement is more likely to result in a potential fix) [10, 34]. An interesting empirical observation by Qi et al. [30] is that Jaccard [15] produced lower NCP than Op2 [26], despite having been proven to always produce a lower ranking for the faulty statement than Op2 [37]. This is due to the actual distribution of the suspiciousness score: while Op2 produced higher ranking for the faulty statement than Jaccard, it assigned almost equally high suspiciousness scores to some correct statements. On the other hand, Jaccard assigned much lower suspiciousness scores to correct statements, despite ranking the faulty statement slightly lower than Op2.

This illustrates that evaluation and theoretical analysis based on the linear ranking model is not applicable to automated program repair techniques. LIL metric can measure the aptitude of FL techniques for automated repair techniques as it measures the effectiveness of localization in terms of information loss rather than the behavioural cost of inspecting a ranking of statements. LIL metric essentially captures the essence of the entropy-based formulation of fault localization [40] in the form of an evaluation metric.

We propose a new evaluation metric that does not suffer from this discrepancy between two consumption models. Let S be the set of n statements of the Program Under Test, $\{s_1, \ldots, s_n\}$, s_f, $(1 \leq f \leq n)$ being the single faulty statement. Without losing generality, we assume that output of any fault localization technique τ can be normalized to [0, 1]. Now suppose that there exists an ideal fault localization technique, $\mathcal{L}$, that can always pinpoint s_f as follows:

$$\mathcal{L}(s_i) = \begin{cases} 1 & (s_i = s_f) \\ \epsilon & (0 < \epsilon \ll 1, s_i \in S, s_i \neq s_f) \end{cases} \tag{2}$$

Note that we can convert outputs of FL techniques that do not use suspiciousness scores in a similar way: if a technique τ simply reports a set C of m statements as candidate faulty statements, we can set $\tau(s_i) = \frac{1}{m}$ when $s_i \in C$ and $\tau(s_i) = \epsilon$ when $s_i \in S - C$.

We now cast the fault localization problem in a probabilistic framework as in the previous work [40]. Since the *suspiciousness* score of a statement is supposed to correlate to the likelihood of the statement containing the fault, we

[1]The minimal suspiciousness score is given to the other statements that are not executed by any of the failing test cases.

[2]Essentially this measures the number of fitness evaluation for the Genetic Programming part of GenProg; hence the lower the NCP score is, the more efficient GenProg becomes, and in turn the more effective the given localization technique is.

convert the suspiciousness score given by an FL technique, $\tau : S \rightarrow [0,1]$, into the probability of any member of S containing the fault, $P_\tau(s)$, as follows:

$$P_\tau(s_i) = \frac{\tau(s_i)}{\sum_{i=1}^{n} \tau(s_i)}, (1 \leq i \leq n) \tag{3}$$

This converts suspiciousness scores given by any τ (including $\mathcal{L}$) into a probability distribution, P_τ. The metric we propose is the Kullback-Leibler divergence [21] of P_τ from $P_\mathcal{L}$, denoted as $D_{KL}(P_\mathcal{L}||P_\tau)$: it measures the information loss that happens when using P_τ instead of $P_\mathcal{L}$ and is calculated as follows:

$$D_{KL}(P_\mathcal{L}||P_\tau) = \sum_i \ln \frac{P_\mathcal{L}(s_i)}{P_\tau(s_i)} P_\mathcal{L}(s_i) \tag{4}$$

We call this as Locality Information Loss (LIL). Kullbacl-Leibler divergence between two given probability distribution P and Q requires the following: both P and Q should sum to 1, and $Q(s_i) = 0$ implies $P(s_i) = 0$. We satisfy the former by the normalization in Equation 3 and the latter by always substituting 0 with ϵ *after* normalizing τ[3] (because we cannot guarantee the implication in our application). When these properties are satisfied, $D_{KL}(P_\mathcal{L}||P_\tau)$ becomes 0 when $P_\mathcal{L}$ and P_τ are identical. As with the Expense metric, the lower the LIL value is the more accurate the FL technique is. Based on Information Theory, LIL has the following strengths compared to the Expense metric:

- **Expressiveness**: unlike the Expense metric that only concerns the actual faulty statement, LIL also reflects how well the suspiciousness of non-faulty statements have been supressed by an FL technique. That is, LIL can be used to explain the results of Qi et al. [30] quantitatively.
- **Flexibility**: unlike the Expense metric that only concerns a single faulty statement, LIL can handle multiple locations of faults. For m faults (or for a fault that consists of m different locations), the distribution $P_\mathcal{L}$ will simply show not one but m spikes, each with $\frac{1}{m}$ as height.
- **Applicability**: Expense metric is tied to FL techniques that produce rankings, whereas LIL can be applied to any FL technique. If a technique assigns suspiciousness scores to statements, it can be converted into P_τ; if a technique simply presents one or more statements as candidate fault location, P_τ can be formulated to have corresponding peaks.

IV. Experimental Setup

We have designed the following three research questions to evaluate the effectiveness of MUSE in terms of the Expense metric [25] and the LIL metric (Section III):

RQ1. Foundation: *To what extent do failing test cases become passing ones on a mutant generated by mutating a faulty statement of a target program, compared with a mutant generated by mutating a correct statement? Also, to what extent do passing test cases become failing ones on a mutant by mutating a correct statement, compared with a mutant by mutating a faulty statement?*

RQ1 is to validate the conjectures in Section II-A, on which MUSE depends. If these conjectures are valid (i.e., more failing test cases become passing after mutating the faulty statement than a correct one, and more passing test cases become failing after mutating a correct statement than the faulty one), we can expect that MUSE will localize a fault precisely.

RQ2. Precision: *How precise is MUSE, compared with Jaccard, Ochiai, and Op2 in terms of the % of executed statements examined to localize a fault?*

Precision in terms of the % of program statements to be examined is the traditional evaluation criteria for fault localization techniques. RQ2 evaluates MUSE with the Expense metric against the three widely studied SBFL techniques – Jaccard, Ochiai, and Op2. Op2 [26] is *proven* to perform well in Expense metric; Ochiai [27] performs closely to Op2, while Jaccard [15] shows good performance when used with automated progrma repair [30].

RQ3. Information Loss: *How precise is MUSE, compared with Jaccard, Ochiai, and Op2 in terms of the Locality Information Loss (LIL) metric?*

RQ3 evaluates the precision of MUSE with the LIL metric introduced in Section III against the three SBFL techniques (Jaccard, Ochiai, and Op2). The smaller the LIL value is, the more precise the FL technique is.

To answer the research questions, we performed a series of experiments by applying Jaccard, Ochiai, Op2, and MUSE to the 14 faulty versions in five real world C programs. The following subsections describe the details of the experiments.

A. Subject Programs

For the experiments, we used five non-trivial real-world programs including `flex` version 2.4.7, `grep` version 2.2, `gzip` version 1.1.2, `sed` version 1.18, and `space`, all of which are from the SIR benchmark suite [7].

Table I describes the target programs including their sizes in Lines of Code, the faulty versions used, and the numbers of failing and passing test cases for each program version/fault pair. From the base versions listed above, we randomly selected three faulty versions from each program except `grep` where a failure is detected only in two faulty versions by the used test suite. `grep v3` and `space v19` have multiple faults and the other versions have one fault per each version. The fault ID of each version is presented in Table I (For the rest of the paper, we refer to these faulty versions with the term *version*). For `flex`, `grep`, and `space`, we used the coverage-adequate test suite provided by the SIR benchmark. `flex` and `grep` has only one coverage adequate test suite. For `space`, we randomly chose one coverage adequate test suite out of 1000 coverage-adequate test suites. For `gzip` and `sed`, we use

[3]ϵ should be smaller than the smallest normalized non-zero suspiciousness score by τ.

TABLE I: Subject Programs, Their Sizes in Lines of Code (LOC), and the Number of failing and passing test cases

Subjects	Ver.	Fault	Size	$\|f_P\|$	$\|p_P\|$	Description
`flex`	v1	`F_HD_1`	12,423	2	40	Lexical Analyzer
	v7	`F_HD_7`	12,423	1	41	Generator
	v11	`F_AA_3`	12,423	20	22	
`grep`	v3	`F_DG_4`	12,653	5	175	Pattern
	v11	`F_KP_2`	12,653	177	22	Matcher
`gzip`	v2	`F_KL_2`	6,576	1	211	Compression
	v5	`F_KP_1`	6,576	17	196	Utility
	v13	`F_KP_9`	6,576	3	210	
`sed`	v1	`F_AG_2`	11,990	42	316	Stream
	v3	`F_AG_17`	11,990	1	357	Editor
	v5	`F_AG_20`	11,990	64	81	
`space`	v19	N/A	9,129	8	145	ADL
	v21	N/A	9,126	1	152	Interpreter
	v28	N/A	9,126	46	107	

the universe test suite, because the SIR benchmark does not provide a coverage-adequate test suite for the two programs. In addition, we excluded the test cases which caused a target program version to crash (e.g., segmentation fault), since `gcov` that we used to measure coverage information cannot record coverage information for such test cases.

B. *Mutation and Fault Localization Setup*

We use `gcov` to measure the statement coverage achieved by a given test case. Based on the coverage information, MUSE generates mutants of the PUT, each of which is obtained by mutating one statement that is covered by at least one failing test case. We use the Proteum mutation tool for the C language [23], which implements the mutation operators defined by Agrawal el al. [12]. For each mutation point in a statement (e.g., a variable or an operator), MUSE generates at most one mutant using Proteum.

We implemented MUSE, as well as Jaccard, Ochiai, and Op2, in 4,200 lines of C++ code. All experiments were performed on 10 machines equipped with Intel i5 3.6Ghz CPUs and 8GB of memory running Debian Linux 6.05.

V. Result of The Experiments

A. *Result of the Mutation*

Table II shows the number of mutants generated per subject program version. On average, MUSE generates 20693.0 (=14557.7+6135.3) mutants per version and uses 14557.7 mutants, while discarding 6135.3 *dormant* mutants, i.e. those for which none of the test cases change their results, on average. [4] This translates into an average of 6.3 mutants per considered target statement. The mutation and the subsequent testing of all mutant versions took 10 hours using the 10 machines.

B. *Regarding RQ1: Validity of the Conjectures*

Table III shows the numbers of the test cases whose results change on each mutant of the target programs. The second and

[4] `sed v5` has no dormant mutant because the fault of `sed v5` is non-deterministic one (i.e., it dynamically allocates an smaller amount of memory than necessary through `malloc()`).

TABLE II: The number of target statements, used mutants, and dormant mutants (Those that do not change any test results) per subject

Subjects	Target Stmt.	Used Mutants	Dormant Mutants
`flex v1`	2,769	29,030	7,375
`flex v7`	2,773	28,575	7,411
`flex v11`	2,766	30,366	8,532
`grep v3`	1,982	18,127	10,201
`grep v11`	1,685	12,029	26,425
`gzip v2`	1,448	1,172	835
`gzip v5`	1,419	2,054	1,896
`gzip v13`	1,450	1,238	887
`sed v1`	2,228	13,215	4,813
`sed v3`	2,224	6,307	2,367
`sed v5`	2,151	23,552	0
`space v19`	3,360	14,489	4,919
`space v21`	3,358	9,708	2,790
`space v28`	2,843	13,946	7,443
Average	2318.3	14557.7	6135.3

TABLE III: The numbers of the test cases whose results change on the mutants

Subject programs	# of Failing Tests that Pass after Mutating:			# of passing tests that fail after mutating:			α
	Correct Stmts. (A)	Faulty Stmts. (B)	(B)/(A)	Correct Stmts. (C)	Faulty Stmts. (D)	(C)/(D)	
`flex v1`	0.0002	1.2727	6155.6	15.7270	8.8182	1.8	0.0009
`flex v7`	0.0002	0.6667	2721.1	16.3644	0.0000	N/A	0.0007
`flex v11`	0.0026	14.2857	5421.3	5.1064	3.5714	1.4	0.0013
`grep v3`	0.1299	0.4792	3.7	30.7825	8.0625	3.8	0.1490
`grep v11`	8.9740	85.8181	9.6	0.1942	0.0000	N/A	5.7939
`gzip v2`	0.0095	0.5625	59.1	113.3410	1.0000	113.3	0.0322
`gzip v5`	0.0611	15.1111	247.2	64.7306	0.1111	582.6	0.0227
`gzip v13`	0.0000	2.7000	N/A	109.2140	0.0000	N/A	0.0141
`sed v1`	0.0095	0.0000	0.0	189.3610	6.1111	31.0	0.0004
`sed v3`	0.0040	0.2500	63.0	238.7950	91.5000	2.6	0.0062
`sed v5`	0.3556	31.8333	89.5	12.6217	12.0690	1.0	0.0365
`space v19`	0.0105	4.6667	444.5	45.7808	13.1667	3.5	0.0057
`space v21`	0.0000	0.3333	N/A	65.6796	1.0000	65.7	0.0002
`space v28`	0.0114	23.0000	2016.5	31.2257	26.5000	1.2	0.0016
Average	0.6835	12.9271	1435.9	67.0660	12.2793	73.4	0.4332

the third columns show the average numbers of failing test cases on P which subsequently pass after mutating a correct statement (i.e. m_c), or a faulty statement (i.e. m_f), respectively. The fifth and the sixth columns show the average numbers of the passing test cases on P which subsequently fail on m_c and m_f respectively. For example, on average, out of the 17 failing test case of `gzip v5`, 0.0611 and 15.1111 failing test cases on `gzip v5` pass on m_c and m_f respectively.

Table III provides supporting evidence for the conjectures of MUSE discussed in Section II. The number of the failing test cases on P that pass on m_f is 1435.9 times greater than the number on m_c on average, which supports the first conjecture. Similarly, the number of the passing test cases on P that fail on m_c is 73.4 times greater than the number on m_f on average, which supports the second conjecture. Based on these results, we claim that both conjectures are true.

One interesting observation is that the first conjecture seems to be more effective than the second conjecture in its capability to distinguish a faulty statement from correct statements: the average ratio of the number of the failing test cases that change to the passing one on m_f over the number on m_c (i.e. 1435.9) is 19 times greater than the average ratio of the passing test cases that change their results on m_c over the number on m_f (i.e. 73.4).

C. Regarding RQ2: Precision of MUSE in terms of the % of executed statements examined to localize a fault

Table IV presents the precision evaluation of Jaccard, Ochiai, Op2, and MUSE with the proportion of executed statements required to be examined before localizing the fault (i.e. the Expense metric) [5]. The most precise results are marked in bold. Following the ranking produced by MUSE, one can localize a fault after examining 0.38% of the target statements on average. The average precision of MUSE is 25.68 (=9.67/0.38), 24.61 (=9.27/0.38), and 20.09 (=7.56/0.38) times higher than that of Jaccard, Ochiai, and Op2, respectively. In addition, MUSE produces the most precise results for 11 out of the 14 studied faulty versions. This provides quantitative answer to **RQ2**: MUSE can outperform the state-of-the-art SBFL techniques over the Expense metric.

In response to Parnin and Orso [29], we also report the absolute rankings produced by MUSE, i.e. the actual number of statements that need to be inspected before encountering the faulty statement. MUSE ranks the faulty statements of the seven faulty versions (`flex v1,v11`, `gzip v2,v5,v13`, and `space v21,v28`) at the top and ranks the faulty statement of another three versions (`flex v7`, `sed v3`, and `space v19`) among the top three. On average, MUSE ranks the faulty statement among the top 7.43 places, which is 24.86 (=184.64/7.43) times more precise than the best performing SBFL technique, Op2. We believe MUSE is precise enough that its results can be used by a human developer in practice.

D. Regarding RQ3: Precision of MUSE in terms of the Locality Information Loss

The LIL column of Table IV shows the precision of Jaccard, Ochiai, Op2, and MUSE in terms of the LIL metric, calculated with $\epsilon = 10^{-16}$. The best results (i.e. the lowest values) are marked in bold. The LIL metric value of MUSE is 2.87 on average, which is 1.96 (=5.63/2.87), 2.06 (=5.91/2.87), and 2.28 (=6.55/2.87) times more precise than those of Jaccard, Ochiai, and Op2. In addition, the LIL metric values of MUSE are the smallest ones on the eleven out of the 14 subject program versions. This answers **RQ3**: MUSE can outperform the state-of-the-art SBFL techniques over the newly proposed LIL metric.

One interesting observation is that MUSE produces Expense and LIL values that correlates relatively well. The versions whose absolute ranking of faulty statement is equal to or less than 3, and whose LIL metric is less than 2.66, are the following 10 versions: `flex v1,v7,v11`, `gzip v2,v5,v13`, `sed v3`, and `space v19,v21,v28`. For another three versions (`grep v3,v11` and `sed v1`), both the Expense and LIL metric values perform worse than the other techniques, although not significantly.

In contrast, Expense and LIL metric often do not agree with each other for the SBFL techniques. Consider `space v21`: Jaccard, Ochiai, and Op2 produces the same Expense value of 0.45%. However, their LIL values are all different (Jaccard: 4.80 < Ochiai: 5.79 < Op2: 7.45). A similar pattern is observed in other subject versions (`flex v7`, `grep v11`, `gzip v2,v5,v13`, `sed v1,v3`, `space v19,v21`). In

[5]The α values used for each subject are listed in the last column of Table III.

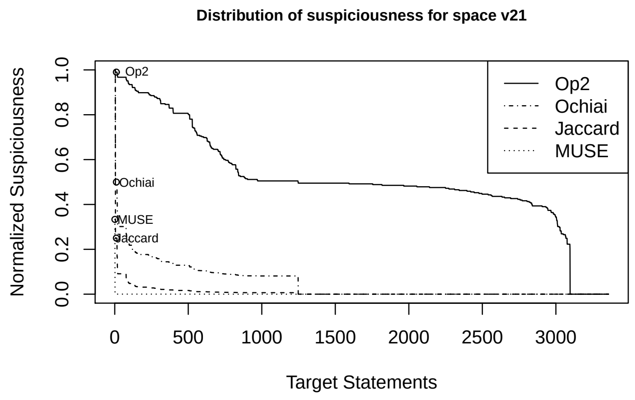

Fig. 3: Normalized suspiciousness scores from `space v21` in descending order

case of `grep v3`, the Expense metric and the LIL metric directly conflict with each other. With respect to the Expense values, Op2 produces the best result, while Jaccard produces the worst result (Jaccard: $1.06 >$ Ochiai: $1.01 >$ Op2: 0.71). However, with respect to the LIL values, Jaccard produces the best result, while Op2 the worst (Jaccard: $5.15 <$ Ochiai: $5.57 <$ Op2: 6.19).

Figure 3 illustrates this phenomenon in more detail. It plots the normalized suspiciousness scores for each target statement of `space v21` in a descending order [6]. The circles indicate the location of the faulty statement. While all techniques assign, to the faulty statement, suspiciousness values that rank near the top, it is the suspiciousness of correct statements that differentiates the techniques. When normalized into $[0, 1]$, MUSE assigns values less than 0.00024 to all correct statements. In contrast, the SBFL techniques assign values much higher than 0. For example, 4.8% of the target statements are assigned suspiciousness higher than 0.9 by Op2, while 37.2% are assigned values higher than 0.5. Figure 4 presents the distribution of suspiciousness in `space v21` for individual techniques to make it easier to observe the differences. This provides supporting evidence to answer **RQ3**: MUSE does perform better than the state-of-the-art SBFL techniques when evaluated using the LIL metric. Figure 4 also intuitively illustrates the strength of the LIL metric over the Expense metric.

This independently confirms the results obtained by Qi et al. [30]. Our new evaluation metric, LIL, confirms the same observation as Qi et al. by assigning Jaccard a lower LIL value of 4.80 than that of Op2, 7.45 (see Section III for more details).

VI. Discussions

A. Why does it work well?

As shown in Section V-C and Section V-D, MUSE demonstrates superior precision when compared to the state-of-the-art SBFL techniques. In addition to the finer granularity of statement level, the improvement is also partly because MUSE

[6]The normalized suspiciousness of a statement s in an FL technique τ, $norm\_susp_{\tau}(s)$ is computed as $(susp_{\tau}(s)-min(\tau))/(max(\tau)-min(\tau))$ where $min(\tau)$ and $max(\tau)$ is the minimum and maximum observed suspiciousness for all statements [30].

TABLE IV: Precision of Jaccard, Ochiai, Op2, and MUtation-baSEd fault localization technique (MUSE)

Subject	% of executed stmts examined				Rank of a faulty stmt				LIL			
Program	Jaccard	Ochiai	Op2	MUSE	Jaccard	Ochiai	Op2	MUSE	Jaccard	Ochiai	Op2	MUSE
`flex v1`	49.48	45.04	32.01	**0.04**	1,371	1,248	887	**1**	8.25	7.79	7.64	**1.28**
`flex v7`	3.60	3.60	3.60	**0.07**	100	100	100	**2**	5.65	6.43	7.49	**1.22**
`flex v11`	19.76	19.54	13.51	**0.04**	547	541	374	**1**	7.27	7.38	7.29	**1.59**
`grep v3`	1.06	1.01	**0.71**	1.87	21	20	**14**	37	**5.15**	5.57	6.19	5.92
`grep v11`	3.44	3.44	3.44	**1.60**	58	58	58	**27**	**5.25**	6.06	5.30	7.19
`gzip v2`	2.14	2.14	2.14	**0.07**	31	31	31	**1**	5.10	4.45	6.23	**1.66**
`gzip v5`	1.83	1.83	1.83	**0.07**	26	26	26	**1**	4.22	4.51	5.12	**1.88**
`gzip v13`	1.03	1.03	1.03	**0.07**	15	15	15	**1**	2.99	3.48	5.68	**0.70**
`sed v1`	**0.54**	**0.54**	**0.54**	0.90	**12**	**12**	**12**	20	**4.08**	4.83	5.63	6.72
`sed v3`	2.56	2.56	2.56	**0.13**	57	57	57	**3**	6.66	6.37	6.96	**2.66**
`sed v5`	37.84	37.84	37.15	**0.28**	814	814	799	**6**	7.10	7.19	7.11	**4.80**
`space v19`	**0.03**	**0.03**	**0.03**	0.06	**1**	**1**	**1**	2	5.12	5.76	6.52	**2.15**
`space v21`	0.45	0.45	0.45	**0.03**	15	15	15	**1**	4.80	5.79	7.45	**0.40**
`space v28`	11.57	10.66	6.89	**0.04**	329	303	196	**1**	7.14	7.21	7.05	**1.96**
Average	9.67	9.27	7.56	**0.38**	242.64	231.50	184.64	**7.43**	5.63	5.91	6.55	**2.87**

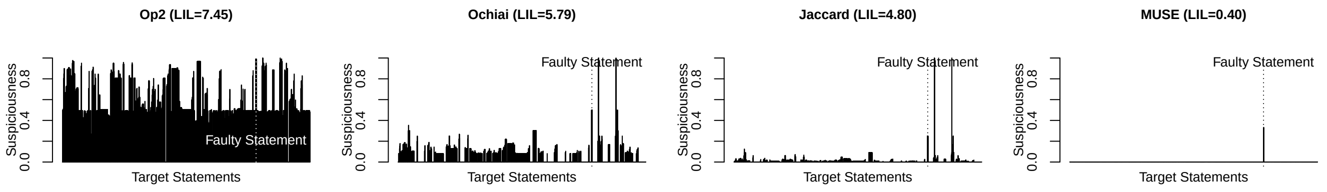

Fig. 4: Comparision of distributions of normalized suspiciousness score across target statements of `space v21`

directly evaluates where (partial) fix can (and cannot) potentially exist instead of predicting the suspiciousness through program spectrum. In a few cases, MUSE actually finds a fix, in a sense that it performs a program mutation that will make all test cases pass (this, in turn, increases the first term in the metric, raising the rank of the location of the mutation). However, in other cases, MUSE finds a *partial* fix, i.e. a mutation that will make only some of previously failing test cases pass. While not as strong as the former case, a partial fix nontheless captures the chain of control and data dependencies that are relevent to the failure and provides a guidance towards the location of the fault.

B. MUSE and Test Suite Balance

One advantage MUSE has over SBFL is that MUSE is relatively freer from the proportion of passing and failing test cases in a test suite. In contrast, SBFL techniques benefit from having a balanced test suite, and have been augmented by automated test data generation work [9, 17, 20].

MUSE does not require the test suite to have many passing test cases. To illustrate the point, we purposefully calculated MUSE metric without any test cases that passed before mutation (this effectively means that we only use the first term of the metric). On average, MUSE ranked the faulty statement within the top 5.09%, which outperforms SBFL techniques that considered all passing and failing test cases: MUSE is still 1.90 (=9.67/5.09), 1.82 (=9.27/5.09) and 1.49(=7.56/5.09) times more precise than Jaccard, Ochiai, and Op2 respectively.

More interestingly, MUSE does not require the test suite to have many failing test cases. Considering that previous work [17, 20] focused on producing more failing test cases to improve the precision, this is an important observation. We purposefully calculated MUSE metric without any test cases that failed before mutation: although this translates into an unlikely use case scenario, it allows us to measure the differentiating power of the second conjecture in isolation. When only the second term of the MUSE metric is calculated (with $\alpha = 1$), MUSE could still rank the faulty statement among the top 14.62% on average, and among the top 2% for seven out of 14 faulty versions we studied. Intuitively, SBFL techniques require many failing executions to identify where a fault is, whereas MUSE is relatively free from this constraint because it also identifies where a fault *is not*.

This advantage is due to the fact that MUSE utilizes two separate conjectures, each of which is based on the number of failing and passing test cases respectively. Thus, even if a test suite has almost no failing or passing test cases, MUSE can localize a fault precisely.

C. LIL Metric and Automated Bug Repair

LIL metric is better at predicting the performance of an FL technique for automated program repair tools than the traditional ranking model. The fact that the ranking model is not suitable has been demonstrated by Qi et al. [30]. We performed a small case study with the GenProg-FL tool by Qi et al., which is a modification of the original GenProg tool. We applied Jaccard, Ochiai, Op2, and MUSE, to GenProg-FL in order to fix `look utx 4.3`, which is one of the subject programs recently used by Le Goues et al. [11]. GenProg-FL [30] measures the NCP (Number of Candidate Patches generated before a valid patch is found in the repair process) of each FL technique where the suspiciousness score of a statement s is used as the probability to mutate s.

TABLE V: Expense, LIL, and NCP scores on `look utx 4.3`

FL Tech.	Expense	LIL	Avg. NCP over 100 runs
MUSE	11.25	3.52	25.3
Op2	42.50	3.77	31.0
Ochiai	42.50	3.83	32.2
Jaccard	42.50	3.89	35.5

Table V shows the Expense, the LIL and the NCP scores on `look utx 4.3` by the four fault localization techniques we have evaluated. For the case study, we generated 30 failing and 150 passing test cases randomly and used the same experiment parameters as in GenProg-FL [30] (we obtained the average NCP score from 100 runs). Table V demonstrates that the LIL metric is useful to evaluate the effectiveness of an FL technique for the automatic repair of `look utx 4.3` by GenProg-FL: the LIL scores ($\text{MUSE} : 3.52 < \text{Op2} : 3.77 < \text{Ochiai} : 3.83 < \text{Jaccard} : 3.89$) and the NCP scores ($\text{MUSE} : 25.3 < \text{Op2} : 31.0 < \text{Ochiai} : 32.2 < \text{Jaccard} : 35.5$) are in agreement.

A small LIL score of a localization technique indicates that the technique can be used to perform more efficient automated program repair. In contrast, the Expense metric values did not provide any information for the three SBFL techniques. We plan to perform further empirical study to support the claim.

VII. Related Work

Among the wide range of fault localization approaches including program state analysis [13, 41] and machine learning [2], the most widely studied has been the spectrum-based approaches [22, 35]. SBFL techniques such as Tarantula [19] and Ochiai [27] have been extensively studied both empirically [18, 31] and theoretically [37]. Other techniques expand the input data to call sequences [6] and *du*-pairs [32]. While the spectrum-based approaches are limited in their accuracy by the basic block structure [33], MUSE is as accurate as the unit of mutation, which is a single statement in the study presented in this paper. With the improved accuracy, the empirical results showed that MUSE can overcome the criticism of inadequate accuracy shared by SBFL techniques [29]. Other techniques attempt to improve the accuracy of SBFL using dynamic data dependency and causal-inference model [1, 8], or combining the SAT-based fault localization [24] with the spectrum-based ones [9].

The idea of generating *diverse program behaviours* to localize a fault more effectively has been utilized by several studies. For example, Cleve and Zeller [13] search for program states that cause the execution to fail by replacing states of a neighbouring passing execution with those of a failing one. If a passing execution with the replaced states no longer passes, relevant statements of the states are suspected to contain faults. Zhang et al. [43], on the other hand, change branch predicate outcomes of a failing execution at runtime to find suspicious branch predicates. A branch predicate is considered suspicious if the changed branch outcome makes a failing execution pass. Similarly, Jeffrey et al. [16] change the value of a variable in a failing execution with the values with other executions; Chandra et al. [5] simulate possible value changes of a variable in a failing execution through symbolic execution. Those techniques are similar to MUSE in a sense that generating diverse program behaviours to localize faults. However, they either *partially* depend on the conjectures of MUSE (some [5, 16, 43] in particular depend on the first conjecture of MUSE) or rely on a different conjecture [13]. Moreover, MUSE does not require any other infrastructure than a mutation tool, because it *directly* changes program source code to utilize the conjectures (Section IV-B).

Since mutation operators vary significantly in their nature, mutation-based approaches such as MUSE may not yield itself to theoretical analysis as naturally as the spectrum-based ones, for which hierarchy and equivalence relations have been shown with proofs [37]. In the empirical evaluation, however, MUSE outperformed Op2 SBFL metric [26], which is the known best SBFL technique.

Yoo showed that risk evaluation formulas for SBFL can be automatically evolved using Genetic Programming (GP) [39]. Some of the evolved formulas were proven to be equivalent to the known best metric, Op2 [38]. While current MUSE metrics are manually designed following human intuition, they can be evolved by GP in a similar fashion.

Papadakis and Le-Traon have used mutation analysis for fault localization [28]. However, instead of measuring the impact of mutation on partial correctness as in MUSE (i.e. the conjecture 1), Papadakis and Le-Traon depend on the similarity between mutants in an attempt to detect unknown faults: variations of existing risk evaluation formulas were used to identify suspicious mutants. Zhang et al. [42], on the other hand, use mutation analysis to identify a fault-inducing commit from a series of developer commits to a source code repository: their intuition is that a mutation at the same location as the faulty commit is likely to result in similar behaviours and results in test cases. Although MUSE shares a similar intuition, we do not rely on tests to exhibit similar behaviour: rather, both of MUSE metrics measures what is the *differences* introduced by the mutation. Given the disruptive nature of the program mutation, we believe MUSE is more robust.

VIII. Conclusion and Future Work

We have presented MUSE, a fault localization technique based on mutation analysis. Based on the two conjectures we introduced, MUSE not only increases the suspiciousness of potentially faulty statements but also decreases the suspiciousness of potentially correct statements. The results of empirical evaluation show that MUSE can not only significantly outperform the state-of-the-art SBFL techniques, but also provide a practical fault localization solution. MUSE is more than 25 times precise compared to Op2, which is the best known SBFL technique; MUSE also ranks the faulty statement at the top for seven out of the 14 faulty versions, and among the top three for another three versions. The paper also presents Locality Information Loss (LIL), a novel evaluation metric for FL techniques based on information theory. A case study shows that it can be better at predicting the performance of an FL technique for automated program repair.

Futurue work includes optimization of the mutation analysis, as well as in-depth study of the impact of different mutation operators. We also plan to extend our tool to support code visualization and inspection features in an integrated environment.

References

[1] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *ISSTA*, 2010.

[2] Y. Brun and M. D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, 2004.

[3] T. A. Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, 1980.

[4] T. A. Budd. Mutation analysis: Ideas, examples, problems and prospects. In *Proceedings of the Summer School on Computer Program Testing*, 1981.

[5] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *ICSE*, 2011.

[6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight bug localization with ample. In *AADEBUG*, 2005.

[7] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE*, 10(4):405–435, 2005.

[8] G.K.Baah, A.Podgurski, and M.J.Harrold. Mitigating the confounding effects of program dependences for effective fault localization. In *ESEC/FSE*, 2011.

[9] D. Gopinath, R. Zaeem, and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *ASE*, 2012.

[10] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[11] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TOSEM*, 38(1):54–72, 2012.

[12] H.Agrawal, R.A.DeMillo, B.Hathaway, W.Hsu, W.Hsu, E.W.Krauser, R.J.Martin, A.P.Mathur, and E.Spafford. Design of mutant operators for the c programming language. Technical Report SERC-TR-120-P, Purdue University, 1989.

[13] H.Cleve and A.Zeller. Locating causes of program failures. In *ICSE*, 2005.

[14] I.Vessey. Expertise in debugging compute programs. *Inter.J.of Man-Machine Studies*, 23(5):459–494, 1985.

[15] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull. Soc. vaud. Sci. nat*, 37:547–579, 1901.

[16] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA*, 2008.

[17] W. Jin and A. Orso. F3: fault localization for field failures. In *ISSTA*, 2013.

[18] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, 2005.

[19] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization for fault localization. In *ICSE*, Software Visualization Workshop, 2001.

[20] J.Rößler, G.Fraser, A.Zeller, and A.Orso. Isolating failure causes through test case generation. In *ISSTA*, 2012.

[21] S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 1951.

[22] M.A.Alipour. Automated fault localization techniques: a survey. Technical report, Oregon State University, 2012.

[23] J. C. Maldonado, M. E. Delamaro, S. C. Fabbri, A. da Silva Simão, T. Sugeta, A. M. R. Vincenzi, and P. C. Masiero. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation testing for the new century*. 2001.

[24] M.Jose and R.Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *PLDI*, 2011.

[25] M.Renieres and S.P.Reiss. Fault localization with nearest neighbor queries. In *ASE*, 2003.

[26] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *TOSEM*, 20(3):11:1–11:32, August 2011.

[27] A. Ochiai. Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions. *Bull. Jpn. Soc. Sci. Fish.*, 22(9):526–530, 1957.

[28] M. Papadakis and Y. Le-Traon. Using mutants to locate "unknown" faults. In *ICST*, Mutation Workshop, 2012.

[29] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA*, 2011.

[30] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *ISSTA*, 2013.

[31] R.Abreu, P.Zoeteweij, and A. Gemund. An evaluation of similarity coefficients for software fault localization. In *PRDC*, 2006.

[32] R.Santelices, J.A.Jones, Y.Yu, and M.J.Harrold. Lightweight fault-localization using multiple coverage types. In *ICSE*, 2009.

[33] F. Steimann, M. Frenkel, and R. Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *ISSTA*, 2013.

[34] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[35] E. Wong and V. Debroy. A survey of software fault localization. Technical Report UTDCS-45-09, University of Texas at Dallas, November 2009.

[36] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai. Effective fault localization using code coverage. In *COMPSAC*, 2007.

[37] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *TOSEM*, 2013 (*to appear*).

[38] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman. Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In *SSBSE*. 2013.

[39] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In *SSBSE*. 2012.

[40] S. Yoo, M. Harman, and D. Clark. Fault localization prioritization: Comparing information-theoretic and coverage-based approaches. *TOSEM*, 22(3):19:1–19:29, July 2013.

[41] A. Zeller. Isolating cause-effect chains from computer programs. In *ESEC/FSE*, 2002.

[42] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*, 2013.

[43] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE*, 2006.