Hybrid-MUSE: Mutating Faulty Programs For Precise Fault Localization

Seokhyeon Moon, Yunho Kim, Moonzoo Kim
CS Dept. KAIST, South Korea
{seokhyeon.moon, kimyunho}@kaist.ac.kr, moonzoo@cs.kaist.ac.kr

Shin Yoo
CS Dept. University College London, UK
shin.yoo@ucl.ac.uk

***Abstract*—This paper presents Hybrid-MUSE, a new fault localization technique that combines MUtation-baSEd fault localization (MUSE) and Spectrum-Based Fault Localization (SBFL) technique. The core component of Hybrid-MUSE, MUSE, identifies 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. This paper also proposes 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 51 faulty versions of the five real-world programs shows that Hybrid-MUSE outperforms the state-of-art fault localization technique significantly. For example, Hybrid-MUSE localizes a fault after reviewing 1.65% of executed statements on average, which is around 5.6 times more precise than the state-of-the-art SBFL technique, Op2.**

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 [15], [46], 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 [41].

SBFL has received much attention, with a heavy emphasis on designing new risk evaluation formulas [20], [40], [6], [44], but also on theoretical analysis of optimality and hierarchy between formulas [27], [42], [43]. However, it has also been criticized for their impractical accuracy and the unrealistic usage model that is the linear inspection of the ranking [30]. 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 Hybrid-MUSE, a combination of *MUtation-baSEd fault localization* and Spectrum-Based Fault Localization (SBFL), to overcome this problem. The core component of Hybrid-MUSE, MUSE [35], uses mutation analysis to uniquely capture the relationship between individual program statements and the observed failures for fault localization. It is free from the coercion of shared ranking from the block structure. Also, the combined SBFL technique adjusts the suspiciousness results of MUSE for cases where mutation analysis does not generate useful results.

Original mutation testing [4] evaluates the adequacy of a test suite based on its ability to detect artificially injected faults, i.e. syntactic *mutations* of the original program [3]. 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. 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 PUT. These two observations form the basis of the design of our new risk evaluation formula for fault localization (Section III-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 [26]. 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 [31].

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 Hybrid-MUSE against 51 faulty versions of five real-world programs. The results show that Hybrid-MUSE is, on average, about six times more accurate than Op2 [27], the current state-of-the-art SBFL technique. In addition, Hybrid-MUSE ranks the faulty statement at the top suspiciousness ranking for 18 out of 51 studied faulty versions (i.e., 35% (=18/51) of the faults are localized perfectly), and within the top 10 suspiciousness ranking for 38 faults (i.e., 75% (=38/51) of the faults are localized in 10 statements). Furthermore, the newly introduced LIL metric also shows that Hybrid-MUSE can be highly accurate, as well as confirming the observation made by Qi et al. [31].

The contribution of this paper is as follows:

- The paper presents a novel fault localization technique called Hybrid-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 Hybrid-MUSE using five non-trivial real world programs. The results demonstrate that Hybrid-MUSE is more accurate than widely studied SBFL techniques such as Jaccard [16], Ochiai [32], and Op2 [27]. The results from the empirical evaluation show that Hybrid-MUSE improves upon the best known SBFL technique by 5.6 times on average and ranks the faulty statement within the top 10 suspicious statements for 38 out of 51 subject program versions.

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

II. Spectrum-Based Fault Localization

Program spectra characterizes an execution information of PUT []. For example, Executable Statement Hit Spectrum (ESHS) records which statements are executed by a test cases.

The key idea of Spectrum-Based Fault Localization (SBFL) techniques is to utilize *correlation* between failing test cases and executed faulty entities (e.g., statements, branches) based on program spectra. SBFL expects the correlation between failing executions and faulty entities is higher than the correlation between failing executions and correct entities. Specifically, SBFL computes suspiciousness scores of entities by analyzing differences between the program spectrum of passing executions and those of failing executions. If a statement is used as the entity in SBFL, the suspiciousness of a statement s increases as the number of failing test cases that execute s increases. Conversely, the suspiciousness of s decreases as the number of passing test cases that execute the statement s increases. High suspiciousness of a statement indicates that the statement is likely to be a faulty one.

For example, Jaccard [16] calculates a suspiciousness score of a statement s by using the following metric:

$$Susp_{Jaccard}(s) = \frac{|f_P(s)|}{|f_P(s)| + |p_P(s)|} \quad (1)$$

where $f_P(s)$ and $p_P(s)$ are a set of failing test cases and a set of passing test cases that execute s respectively.

Similarly, the suspiciousness metrics of Ochiai [32] and Op2 [27] are as follows:

$$Susp_{Ochiai}(s) = \frac{|f_P(s)|}{\sqrt{|f_P| * (|f_P(s)| + |p_P(s)|)}} \quad (2)$$

$$Susp_{Op2}(s) = |f_P(s)| - \frac{|p_P(s)|}{|p_P| + 1} \quad (3)$$

where f_P and p_P are a set of failing test cases and a set of passing test cases for program P respectively. Section III-D explains how SBFL techniques localize a fault through a concrete example.

III. Hybrid Mutation-based Fault Localization Technique

Hybrid-MUSE utilizes both MUtation-baSEd fault localization (MUSE) and Spectrum-Based Fault Localization (SBFL) technique. In this section, we first present the core component of Hybrid-MUSE, MUSE, and then present Hybrid-MUSE, which is the combination of MUSE and SBFL.

A. Intuitions of Mutation-based Fault Localization Technique

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 least one fault. Let m_f be a mutant of P that mutates the faulty statement, and m_c be 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. For a statement s of P, let $f_P(s)$ be the set of tests that covered s and failed 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 $mut(s) = \{m_1, \dots m_k\}$ be the set of all mutants of P that mutates s with observed changes in test results. For each mutant $m_i \in mut(s)$, let f_{m_i} and p_{m_i} be the set of failing and passing tests on m_i respectively. And let $f2p$ and $p2f$ be 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 $mut(P)$. The mutation-based fault localization metric $Susp_{MUSE}(s)$ is defined as follows:

$$Susp_{MUSE}(s) = \frac{1}{|mut(s)|+1} \times \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{f2p+1} - \frac{|p_P(s) \cap f_m|}{p2f+1}\right)$$

The first term, $\frac{|f_P(s) \cap p_m|}{f2p+1}$, reflects the first conjecture: it is the proportion of the number of tests that failed on P but now pass on a mutant m that mutates s over the total number of all failing tests that pass on a some mutant (since $f2p$ can be zero, we adds 1 to $f2p$ in the denominator to avoid division-by-zero). Similarly, the second term, $\frac{|p_P(s) \cap f_m|}{p2f+1}$, reflects the second conjecture, being the proportion of the number of tests that passed on P but now fail on a mutant m that mutates s over the total number of all passing tests that fail on a some mutant. When averaged over $|mut(s)|+1$ (again, we adds 1 to avoid division-by-zero), the first term and the second term 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 $p_P(s) \cap f_m$).

Note that MUSE does not implicitly assume that a fault lies in one statement. A partial fix of a multi-line spanning fault obtained by mutating one statement can still make the target program pass with a subset of failing tests and provide important information to localize a fault (Conjecture 1).

C. Suspiciousness Metric of Hybrid-MUSE

In addition to MUSE, Hybrid-MUSE utilizes a SBFL technique to adjust MUSE when MUSE suspiciousness metric (i.e., $Susp_{MUSE}(s)$) performs poorly in exceptional cases. For example, when MUSE fails to generate a mutant on s (i.e., $|mut(s)| = 0$), $\mu(s) = 0$ since $p_m = f_m = \emptyset$. As MUSE utilizes observed changes in test results, MUSE suspiciousness metric does not provide a meaningful suspiciousness score for s. In addition, if both MUSE and SBFL indicate a statement s is more suspicious than other statements, s can be more likely to faulty one than statements that are indicated by either of them.

Based on those intuitions, therefore, Hybrid-MUSE suspiciousness metric is defined as follows:

$$Susp_{Hybrid\_MUSE}(s) = norm\_susp(MUSE, s) + norm\_susp(sbfl, s) \quad (4)$$

where $norm\_susp(flt, s)$ is the normalized suspiciousness of a statement s in a FL technique flt, which is computed as $\frac{Susp_{flt}(s) - min(flt)}{max(flt) - min(flt)}$ where $min(flt)$ and $max(flt)$ is the minimum and maximum observed suspiciousness score computed by $Susp_{flt}(s)$ for all target statements [31]. $sbfl$ is a SBFL metric such as Jaccard, Ochiai, or Op2. Note that the purpose of normalization is to balance the suspiciousness scores of the both techniques. The suspiciousness normalization, which makes a suspiciousness range of FL technique 0 to 1, prevents a suspiciousness of a FL technique dominates the other one.

Since SBFL techniques does not require mutation analysis to compute suspiciousness of each statement, the combination of MUSE with a SBFL metric enables Hybrid-MUSE to assign meaningful suspiciousness to a statement s where $|mut(s)| = 0$.

		Spectrum 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)	$\|f_P(s)\|$	$\|p_P(s)\|$	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		Hybrid-MUSE	
	Statements	Mutants	TC_1 (3,1)	TC_2 (5,-4)	TC_3 (0,-4)	TC_4 (0,7)	TC_5 (-1,3)	$\|f_P(s) \cap p_m\|$	$\|p_P(s) \cap f_m\|$	Suspiciousness	Rank	Suspiciousness	Rank
s_1:	max = −x;	m1:max −= x−1; m2:max=x;	F→P	F→P	P→F			0 2	1 0	0.15	1	1.40	1
s_2:	if (max < y){	m3:if (!(max<y)){ m4:if (max==y){	F→P		P→F	P→F P→F	P→F	0 1	3 1	0.02	4	0.83	4
s_3:	max = y;	m5:max = −y; m6:max = y+1;				P→F P→F	P→F P→F	0 0	2 2	-0.07	3	1.07	3
s_4:	if (x∗y<0){	m7:if (!(x∗y<0)) m8:if (x/y<0)				P→F	P→F P→F	0 0	2 1	-0.05	2	1.14	2
s_5:	print (" diff . sign ");}	m9:return ; m10:;					P→F P→F	0 0	1 1	-0.03	6	0.21	6
s_6:	print (max);}	m11:printf (0);} m12:;}			P→F	P→F	P→F P→F	0 0	2 3	-0.08	5	0.40	5

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

D. An Working Example

Figure 1 presents an example of how Hybrid-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 spectrum 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 suspiciousness of s_1, $Susp_{MUSE}(s_1)$, is computed as $\frac{1}{(2+1)} \cdot \{(\frac{0}{3+1} - \frac{1}{19+1}) + (\frac{2}{3+1} - \frac{0}{19+1})\} = 0.15$, 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. 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 $|p_P| = 3$. The MUSE suspiciousness scores of other statements are computed as 0.02, -0.07, -0.05, -0.03, and -0.08 (See MUSE column).

After calculating MUSE suspiciousness for each statement s, normalized MUSE suspiciousness scores of statements are computed to combine MUSE suspiciousness with SBFL suspiciousness. The normalized suspiciousness of the statement s_1 in MUSE, $norm\_susp(MUSE, s_1)$, is computed as $\frac{0.15-(-0.08)}{0.15-(-0.08)} = 1$, where $Susp_{MUSE}(s_1)$ = 0.15 and $min(MUSE)$ = -0.08, and $max(MUSE)$ = 0.15. Similarly, $norm\_susp(MUSE, s_2)$ = 0.43, $norm\_susp(MUSE, s_3)$ = 0.07, $norm\_susp(MUSE, s_4)$ = 0.14, $norm\_susp(MUSE, s_5)$ = 0.21, and $norm\_susp(MUSE, s_6)$ = 0.

In addition, normalized suspiciousness scores of statements in a $SBFL$ technique (e.g., Jaccard) are also computed. The normalized suspiciousness of the statement s_1 in the Jaccard, $norm\_susp(Jaccard, s_1)$, is computed as $\frac{0.40-(-0.33)}{0.50-(-0.33)} = 0.40$, where $Jaccard(s_1)$ = 0.40 and $min(Jaccard)$ = 0.33, $max(Jaccard)$ = 0.50 (see Susp. column of Jaccard). Similarly, $norm\_susp(Jaccard, s_2)$ = 0.40, $norm\_susp(Jaccard, s_3)$ = 1.00, $norm\_susp(Jaccard, s_4)$ = 1.00, $norm\_susp(Jaccard, s_5)$ = 0.00, and $norm\_susp(Jaccard, s_6)$ = 0.40.

Finally, Hybrid-MUSE computes suspiciousness score for each statement s, $Susp_{Hybrid-MUSE}(s)$, based on the normalized suspiciousness scores of s in the MUSE and in the SBFL (Jaccard). The Hybrid-MUSE computes s_1 as $1.00 + 0.40 = 1.40$ where $norm\_susp(MUSE, s_1) = 1.00$ and $norm\_susp(Jaccard, s_1) = 0.40$. The suspiciousness scores of the other five statements are computed as 0.83, 1.07, 1.14, 0.21, and 0.40, respectively (see Suspiciousness column of Hybrid-MUSE). The suspiciousness of the s_1 (i.e., the faulty statement) is the highest at 1.40, which places it at the top 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. For example, Jaccard assigns the highest suspiciousness scores (i.e., $\frac{2}{2+2} = 0.5$) to s_3 and s_4 and marked them as rank 2, while the faulty line (s_1) has a suspiciousness score $\frac{2}{2+3} = 0.4$ and the faulty line is marked as rank 5. This is because there are more passing test cases that execute s_1 (i.e., $tc3$, $tc4$, and $tc5$) than s_3 or s_4 (i.e., $tc4$ and $tc5$). For Jaccard, Ochiai, and Op2, passing test cases such as $tc3$ to $tc5$ make the suspiciousness of the faulty statement low, since they increase $|p_P(s)|$ in the denominator of the

suspiciousness metrics (Jaccard and Ochiai) or $|p_P(s)|$ in the negative term of the suspiciousness metric (Op2). Thus, a fault localization technique like Jaccard, Ochiai, and Op2 becomes imprecise as there are more passing test cases that execute faulty statements. It implies that MUSE can precisely locate certain faults that the state-of-the-art SBFL techniques cannot.

E. MUSE Framework

Figure 2 shows framework of the core component of Hybrid-MUSE, MUSE (**MU**tation-ba**SE**d fault localization). There are three major stages: *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. that contains at least one failing test case. After executing T on P, Hybrid-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. [1]

Step 2: MUSE generates mutants of P by mutating each of the statements selected at Step 1. MUSE can generate multiple mutants from a single statement since one statement may contain multiple mutation points [13]. 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 and calculates the suspiciousness of the target statements of P.

IV. 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 [26] 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 [19], [27], [44] as well as a theoretical framework that showed hierarchies between SBFL techniques [43], [42]. However, the Expense metric has been criticised for being unrealistic to be used by a human developer directly [30].

In an attempt to evaluate the precision of SBFL techniques, Qi et al. [31] compared SBFL techniques by measuring the Number of Candidate Patches (NCP) generated by GenProg [38] 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) [11], [38]. An interesting empirical observation by Qi et al. [31] is that Jaccard [16] produced lower NCP than Op2 [27], despite having been proven to always produce a lower ranking for the faulty statement than Op2 [42]. 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 [45] 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:

$$\mathcal{L}(s_i) = \begin{cases} 1 & (s_i = s_f) \\ \epsilon & (0 < \epsilon \ll 1, s_i \in S, s_i \neq s_f) \end{cases} \tag{5}$$

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 [45]. Since the *suspiciousness* score of a statement is supposed to correlate to the likelihood of the statement containing the fault, we convert the suspiciousness score given by an FL technique, $\tau : S \to [0, 1]$, into the probability of any member of S containing the fault, $P_\tau(s)$, as follows:

$$P_\tau(s_i) = \frac{\tau(s_i)}{\sum_{i=1}^{n} \tau(s_i)}, (1 \leq i \leq n) \tag{6}$$

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 [22] 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

[1] It is possible that a statement that is not executed by any failing test case of T can still be faulty if T fails to detect the fault. Hybrid-MUSE targets faults that have been detected by a given test suite T as many other FL technique do.

[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.

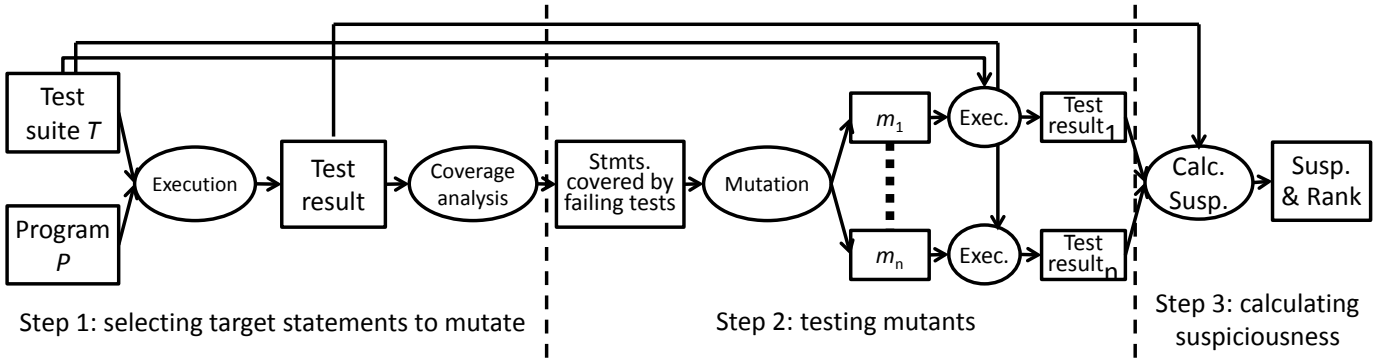

Fig. 2. Process of the mutation based fault localization of MUSE

calculated as follows:

$$D_{KL}(P_{\mathcal{L}}||P_{\tau}) = \sum_{i} \ln \frac{P_{\mathcal{L}}(s_i)}{P_{\tau}(s_i)} P_{\mathcal{L}}(s_i) \tag{7}$$

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 6 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 suppressed by an FL technique. That is, LIL can be used to explain the results of Qi et al. [31] 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.

V. Empirical Study I

To evaluate the effectiveness of our proposed techniques empirically, we have have designed the following research questions. In this study, the effectiveness will be evaluated via both traditional evaluation metric for FL, i.e. Expense metric [26], and newly introduced evaluation metric, i.e. LIL metric (Section IV). Expense metric measures % of executed statements to be examined to localize a faulty statement, if human developer inspects PUT following the order of statements in the ranking generated by a fault localization technique.

RQ1. Foundation of MUSE: *How many test results change from failure to pass and vice versa on a mutant generated by mutating a faulty statement, compared with a mutant generated by mutating a correct one?*

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

RQ2. Effectiveness of Hybrid-MUSE SBFL component: *How precise is Hybrid-MUSE compared with MUSE, which does not utilize SBFL, in terms of Expense and LIL metric?*

RQ2 evaluates the precision of Hybrid-MUSE (i.e., MUSE + Jaccard) and that of MUSE, which does not utilize SBFL for fault localization. The goal of this study is to evaluate the effectiveness of Hybrid-MUSE SBFL component.

RQ3. Precision in terms of Expense metric: *How precise is Hybrid-MUSE, compared with Jaccard, Ochiai, and Op2 in terms of Expense metric?*

RQ3 evaluates Hybrid-MUSE with the Expense metric metric against the three widely studied SBFL techniques – Jaccard, Ochiai, and Op2. Op2 [27] is *proven* to perform well in Expense metric; Ochiai [28] performs closely to Op2, while Jaccard [16] shows good performance when used with automated program repair [31].

RQ4. Precision in terms of LIL metric: *How precise is Hybrid-MUSE, compared with Jaccard, Ochiai, and Op2 in terms of LIL metric?*

RQ4 evaluates Hybrid-MUSE with the newly introduced metric, LIL, against the three SBFL techniques. The smaller the

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

TABLE I
SUBJECT PROGRAMS, # OF VERSIONS USED, THEIR AVERAGE SIZES IN LINES OF CODE (LOC), AND THE AVERAGE NUMBER OF FAILING AND PASSING TEST CASES

Subject program	# of ver. used	Size (LOC)	$\lvert f_P \rvert$	$\lvert p_P \rvert$	Description
`flex`	13	12,423.0	15.9	24.4	Lexical analyzer generator
`grep`	2	12,653.0	91.0	98.5	Pattern matcher
`gzip`	7	6,576.0	34.3	178.6	Compression utility
`sed`	5	11,990.0	43.4	235.0	Stream editor
`space`	24	9,129.0	22.8	130.2	ADL interpreter

LIL value is, the more precise the FL technique is.

RQ5. Relation between the precision and the number of mutants utilized: *How precise is Hybrid-MUSE with 1%, 10%, 40%, 70% and 100% of mutants utilized in terms of both the expense metric and the LIL metric?*

The core component of Hybrid-MUSE, MUSE, can generate a large number of mutants (see Table II) and consumes a large amount of time consequently. However, since Hybrid-MUSE also utilizes SBFL that does not depend of mutants utilized, Hybrid-MUSE may localize a fault enough precisely using only a subset of the mutants. RQ3 evaluates the relation between the mutant sampling rates and related precisions.

To answer the research questions, we performed a series of experiments by applying Jaccard, Ochiai, Op2, MUSE, and Hybrid-MUSE to the 51 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 2.05, and `space`, all of which are from the SIR benchmark suite [7].

Table I describes the target programs including the numbers of the faulty versions used, their sizes in Lines of Code, and the average numbers of failing and passing test cases for each faulty version. We have targeted all faulty versions of the five programs in the SIR benchmark suite (i.e., the total number of versions of the subject programs is 51). We used the test suites provided in the SIR benchmark. 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 statement 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 [24], which implements the mutation operators defined by Agrawal el al. [13]. For each statement (e.g., a variable or an operator), MUSE generates all mutants that can be generated by Proteum. (see Table II for the generated mutants).

For each mutant sampling rate $x\%$, MUSE randomly selects $x\%$ of mutants among the all mutants generated per statement. For example, if MUSE generates 85 mutants at line 50, MUSE randomly selects 9 (= $\lceil 85 \times 10\% \rceil$) mutants with the mutant sampling rate 10%. We repeated this mutant sampling experiment 30 times per subject faulty version to minimize bias from the random sampling process.

We implemented Hybrid-MUSE, MUSE, as well as Jaccard, Ochiai, and Op2, in 7,200 lines of C++ code. All experiments were performed on 25 machines equipped with Intel i5 3.6Ghz quad core CPUs and 8GB of memory running Debian Linux 6.05 64bits.

C. Threats to Validity

The primary threat to external validity for our study involves the representativeness of our object programs, since we have examined only 5 C programs in the SIR benchmark. While this may be the case, the SIR benchmark provides various real-world programs including real faults (i.e., faults that were identified during testing and operational use of the programs) and seeded faults (i.e., faults seeded by other researchers, not us) to support controlled experimentation: all of `space` faults are real faults, and all of `flex`, `grep`, `gzip`, and `sed` faults are seeded ones [7]. In addition, the benchmark has been widely used for fault localization studies [39], [40], [37], [33], [48], [17]. Thus, we believe that this threat to external validity is limited. The second threat involves the representativeness of the fault localization techniques that we applied (i.e., Jaccard, Ochiai, Op2). However, Jaccard performs well when used with a repair tool [31], and Ochiai is generally known to outperform other kinds of fault localization techniques [19], [1]. In addition, Op2 is theoretically proved to outperform other SBFL techniques on single fault programs [31]. Thus, we believe that this threat is also limited. The third possible threat is the assumption that we already have many test cases when applying SBFL techniques to a target program. Our rebuttal to this argument is that we can generate many test cases using automated test generation techniques (i.e., random testing, concolic testing, or test generation strategies for fault localization [34], [10], [21]). A primary threat to internal validity is possible faults in the tools that implement Jaccard, Ochiai, Op2, MUSE, and Hybrid-MUSE. We controlled this threat through extensive testing of our tool.

VI. RESULT OF THE EMPIRICAL STUDY I

A. Result of the Mutation

Table II shows the number of mutants generated per subject program version. On average, MUSE generates 98342.4 (=44411.5+53930.9) mutants per version and uses 44411.5 mutants, while discarding 53930.9 *dormant* mutants, i.e. those for which none of the test cases change their results. This translates into an average of 66.3 (=98342.4/1482.5) mutants

TABLE II
THE AVERAGE NUMBERS OF TARGET STATEMENTS, AVERAGE USED MUTANTS, AND AVERAGE DORMANT MUTANTS (MUTANTS THAT DO NOT CHANGE ANY TEST RESULTS) PER SUBJECT PROGRAM

Subject program	No. of target statements	No. of used mutants	No. of dormant mutants
`flex`	2,107.2	61,917.8	56,756.7
`grep`	1,508.0	499,008.0	149,627.5
`gzip`	398.6	16,209.7	24,265.1
`sed`	1,713.8	49,070.2	15,139.0
`space`	1,685.1	45,851.8	23,866.0
Average	1,482.5	44,411.5	53,930.9

TABLE III
THE AVERAGE NUMBERS OF THE TEST CASES WHOSE RESULTS CHANGE ON THE MUTANTS

Subject program	# of failing tests that pass after mutating:			# of passing tests that fail after mutating:		
	Correct stmts. (A)	Faulty stmts. (B)	(B)/(A)	Correct stmts. (C)	Faulty stmts. (D)	(C)/(D)
`flex`	0.0896	9.7947	109.3234	7.9664	3.8525	2.0756
`grep`	8.3053	38.6876	4.6582	13.2657	3.2245	4.1140
`gzip`	0.1042	3.6774	35.2893	87.8037	4.1328	21.2458
`sed`	1.4087	10.6872	7.5866	108.8567	30.1406	3.6116
`space`	0.0088	3.6996	419.1440	31.6902	15.1574	2.0907
Average	1.9833	13.3093	115.2003	49.9225	11.3016	6.6276

per considered target statement (Section III-E) [4]. The mutation and the subsequent testing of all mutants took 19 hours using the 25 machines, i.e. 22.4 (=(19×60)/51) minutes per each version of a target program, on average.

B. Regarding RQ1: Foundation of MUSE

Table III shows the numbers of the test cases whose results change on each mutant of the target programs. The second and 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. For example, on average, 0.0896 and 9.7947 failing test cases on `flex` pass on m_c and 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.

Table III provides supporting evidence for the conjectures of MUSE discussed in Section III. The number of the failing test cases on P that pass on m_f is 115.2 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 6.6 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 valid.

[4]The numbers of target statements of the subject programs are smaller than the sizes (LOC) of the programs because MUSE targets only statements that are executed by at least one failing test case.

C. Regarding RQ2: Effectiveness of Hybrid-MUSE SBFL component

Table IV presents the precisions of MUSE and Hybrid-MUSE [5] in terms of both Expense and LIL metric. More precise results are marked in bold. Presuming that a developer inspects statements according to their rankings generated by Hybrid-MUSE until reaching the fault (i.e., Expense metric), one can localize a fault after reviewing 1.65% of the executed statements on average. As indicated in Table IV, the average precision of Hybrid-MUSE is 4.08 (=6.74/1.65) times higher than that of MUSE. This answers to **RQ2**: The SBFL component of Hybrid-MUSE is effective enough to improve the precision of MUSE.

Considering that the LIL metric, MUSE is more precise than Hybrid-MUSE. As indicated in Table IV, the average LIL value of MUSE is 5.24, which is 1.12 (=5.86/5.24) times more precise than that of Hybrid-MUSE. However, the decreased precision (in terms of LIL) due to the combined SBFL component of Hybrid-MUSE is negligible, while considering that Hybrid-MUSE outperforms MUSE 4.08 times over the Expense metric. Thus, we consider Hybrid-MUSE as our best performer.

The main reason why MUSE performs worse compared with that of Hybrid-MUSE over the Expense metric is due to the faulty statements that do not have any used mutants. Since MUSE depends on the two conjectures that utilize observed test result changes by mutation (Section III-A), MUSE cannot precisely localize faulty statements that do not have any used mutants. The suspiciousness metric of MUSE assigns 0 suspiciousness to all non-mutated statements. In contrast, Hybrid-MUSE can precisely localize non-mutated faulty statements thanks to the SBFL component of Hybrid-MUSE, which is not utilized by MUSE. The suspiciousness metric of Hybrid-MUSE can assign meaningful suspiciousness scores to non-mutated statements using the SBFL component. Among the 51 faulty versions we studied, there are 7 faulty versions in which all the faulty statements are not-mutated: `flex v16, v17, v18, v19, gzip v14, space v4, and v37`. The average expense of Hybrid-MUSE on these versions is 5.27%, whereas that of MUSE is 27.53%. Hybrid-MUSE is 5.22 (=27.53/5.27) times more precise than MUSE on these versions. This provides supporting evidence to answer **RQ2**: Since SBFL component helps Hybrid-MUSE assign meaningful suspiciousness scores to non-mutated statements, the SBFL component of Hybrid-MUSE can improve the precision compared with MUSE over the Expense metric.

[5]We have utilized Jaccard, Ochiai, and Op2 as the SBFL component of Hybrid-MUSE to adjust MUSE (see Section III-C). We found that 'MUSE + Jaccard' outperforms other combinations over LIL metric, and performs comparably over Expense metric (i.e., On average, the Expense and LIL of 'MUSE + Jaccard' are 1.65% and 5.86 respectively while those of 'MUSE + Ochiai' are 1.62% and 6.18 and those of 'MUSE + Op2' are 1.80% and 6.67 respectively). Thus, from now on, we call 'MUSE + Jaccard' as Hybrid-MUSE for the sake of simplicity.

TABLE IV
COMPARISON OF MUSE AND HYBRID-MUSE

Subject Program	% of Executed Stmts Examined		Rank of Faulty Stmt		Locality Information Loss	
	MUSE	Hybrid-MUSE	MUSE	Hybrid-MUSE	MUSE	Hybrid-MUSE
flex	17.72	**4.38**	490.62	**121.00**	**6.12**	8.30
grep	1.62	**0.91**	29.00	**16.00**	6.71	**6.18**
gzip	7.58	**0.84**	109.71	**12.14**	**3.03**	4.06
sed	1.16	**0.45**	25.00	**10.00**	6.30	**5.99**
space	5.63	**1.67**	126.33	**48.88**	**4.04**	4.74
Average	6.74	**1.65**	156.13	**41.60**	**5.24**	5.86

TABLE V
AVERAGE PRECISIONS OF JACCARD, OCHIAI, OP2 AND HYBRID-MUSE

Subject Program	% of Executed Stmts Examined				Rank of Faulty Stmt				Locality Information Loss			
	Jaccard	Ochiai	Op2	Hybrid-MUSE	Jaccard	Ochiai	Op2	Hybrid-MUSE	Jaccard	Ochiai	Op2	Hybrid-MUSE
flex	23.66	23.25	19.37	**4.38**	458.62	447.77	350.15	**121.00**	8.86	9.00	9.09	**8.30**
grep	2.76	2.73	2.58	**0.91**	48.00	47.50	44.50	**16.00**	**5.34**	5.94	5.83	6.18
gzip	6.76	6.68	6.68	**0.84**	97.86	96.86	96.86	**12.14**	4.69	4.85	5.84	**4.06**
sed	11.57	11.57	11.42	**0.45**	249.40	249.40	246.20	**10.00**	6.06	6.35	6.70	**5.99**
space	4.08	3.61	6.20	**1.67**	125.75	110.42	198.04	**48.88**	5.26	5.78	6.47	**4.74**
Average	9.76	9.57	9.25	**1.65**	195.92	190.39	187.15	**41.60**	6.04	6.38	6.79	**5.86**

TABLE VI
AVERAGE PRECISION OF HYBRID-MUSE ON TARGET SUBJECTS CONTAINING MULTIPLE FAULTS

Target program	# of versions			% of executed stmts examined until the first fault is detected		
	w/ single fault	w/ multi faults	Total	Avg. of single fault versions	Avg of multiple faults versions	Total avg.
flex	11	2	13	5.07	0.59	4.38
grep	1	1	2	1.31	0.50	0.91
gzip	6	1	7	0.96	0.07	0.84
sed	2	3	5	0.23	0.60	0.45
space	10	14	24	0.78	2.30	1.67
SUM/AVG	30	21	51	1.67	0.81	1.65

D. Regarding RQ3: Precision in terms of the Expense metric

Table V presents the precision evaluation of Jaccard, Ochiai, Op2, and Hybrid-MUSE with the proportion of executed statements required to be examined before localizing the fault (i.e. the Expense metric). The most precise results are marked in bold. As indicated in the Table, the average precision of Hybrid-MUSE is 5.9 (=9.76/1.65), 5.8 (=9.57/1.65), and 5.6 (=9.25/1.65) times higher than that of Jaccard, Ochiai, and Op2, respectively. In addition, Hybrid-MUSE produces the most precise results for 46 out of the 51 studied faulty versions. This provides quantitative answer to **RQ3**: Hybrid-MUSE can outperform the state-of-the-art SBFL techniques over the Expense metric.

In response to Parnin and Orso [30], we also report the absolute rankings produced by Hybrid-MUSE, i.e. the actual number of statements that need to be inspected before encountering the faulty statement. Hybrid-MUSE ranks the faulty statements of the 18 faulty versions at the *top*, and ranks the faulty statement of 38 versions among the *top 10*. On average, Hybrid-MUSE ranks the faulty statements among the top 41.6 statements, which is 4.5 (=187.15/41.60) times more precise than the best performing SBFL technique, Op2. We believe Hybrid-MUSE is precise enough that its results can be used by a human developer in practice.

In addition, Table VI shows that Hybrid-MUSE is also precise on target versions containing multiple faults. This is an advantage over conventional SBFL techniques which are knows to be less precise on target programs containing multiple faults in general. We guess that this advantage is due to the partial fixes that Hybrid-MUSE generates, each of which separately captures the control and data dependencies relevant to each fault in a target program (see Section VII-A).

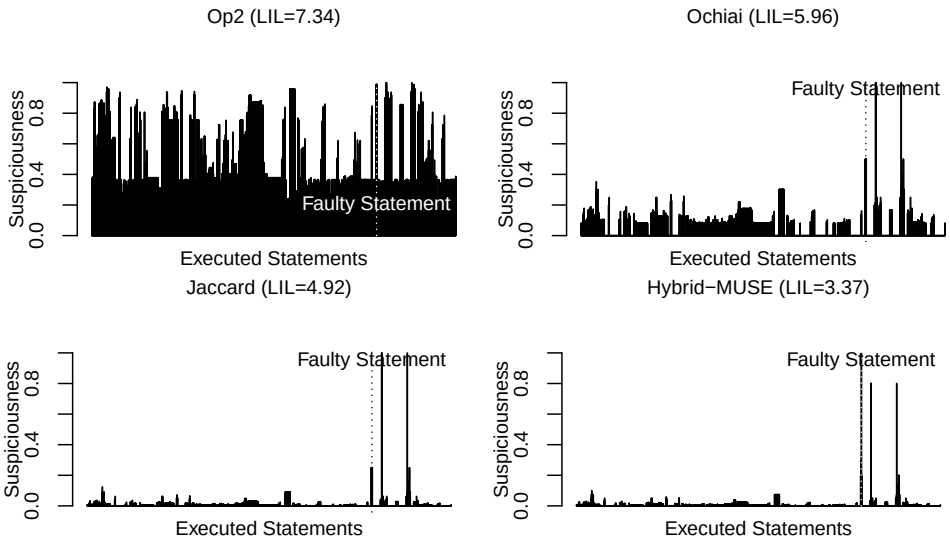

Fig. 4. Comparision of distributions of normalized suspiciousness score across target statements of `space v21`

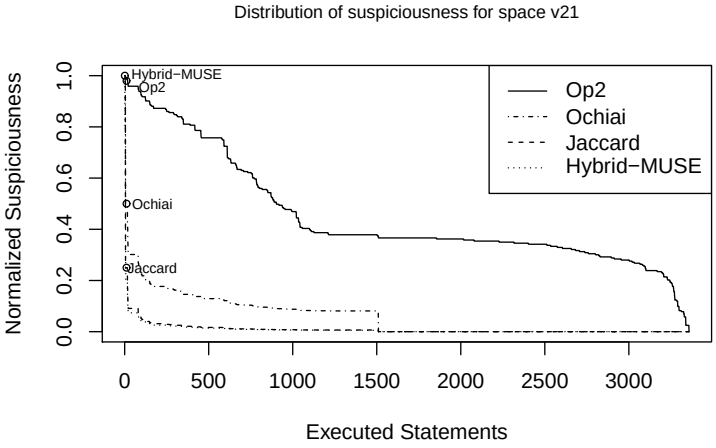

Fig. 3. Distribution of normalized suspiciousness scores for `space v21` in descending order

E. Regarding RQ4: Precision in terms of the LIL metric

The LIL column of Table V shows the precision of Jaccard, Ochiai, Op2, and Hybrid-MUSE in terms of the LIL metric, calculated with $\epsilon = 10^{-17}$. The best results (i.e. the lowest values) are marked in bold. The LIL metric value of MUSE is 5.86 on average, which is 1.03 (=6.04/5.86), 1.09 (=6.38/5.86), and 1.16 (=6.79/5.86) times more precise than those of Jaccard, Ochiai, and Op2. In addition, the LIL metric values of Hybrid-MUSE are the smallest ones on the 29 out of the 51 subject program versions. This answers **RQ4**: Hybrid-MUSE can outperform the state-of-the-art SBFL techniques over the newly proposed LIL metric.

Figure 3 shows the advantage of the Hybrid-MUSE over the SBFL techniques in a different aspect. It plots the normalized suspiciousness scores for each target statement of `spacev21` in a descending order using Jaccard, Ochiai, Op2, and Hybrid-MUSE. The circles indicate the location of the faulty statement. While all techniques assign, to the faulty statement, suspiciousness scores that rank near the top (the ranks of the faulty statement in Jaccard, Ochiai, Op2, and Hybrid-MUSE are 16, 16, 16, and 1, respectively), it is the suspiciousness of correct statements that differentiates the techniques. When normalized into $[0, 1]$, Hybrid-MUSE assigns values less than 0.3 to all correct statement except four statements (This is because the SBFL component of Hybrid-MUSE (i.e., Jaccard) assigns high suspiciousness scores to those four correct statements). In contrast, the SBFL techniques assign values much higher than 0. For example, 4.3% of the executed statements are assigned suspiciousness higher than 0.9 by Op2, while 26.9% 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 **RQ4**: Hybrid-MUSE does perform better than the state-of-the-art SBFL techniques when evaluated using the LIL metric.

Table V also independently confirms the results obtained by Qi et al. [31]. Our new evaluation metric, LIL, confirms the same observation as Qi et al. by assigning Jaccard a lower LIL value of 6.04 than that of Op2, 6.79 (see Section IV for more details).

F. Regarding RQ5: Relation between the precision and the number of mutants utilized

Although Hybrid-MUSE localizes a fault almost six times more precisely than SBFL techniques (see Section VI-C), Hybrid-MUSE can consume a large amount of time to test 98342.4 mutants for a target program, on average (see Table II). If Hybrid-MUSE utilizes only a subset of mutants, it can be faster although precision may decrease. Table VII shows such trade-off between mutant sampling rates and the

TABLE VII
AVERAGE PRECISIONS OF HYBRID-MUSE PER MUTANT SAMPLING RATE

Mutant Sampling Rate	% of Executed Stmts Examined	Rank of a Faulty Stmt	LIL
1%	6.20	123.50	6.26
10%	4.60	95.53	6.11
40%	2.79	61.21	5.96
70%	1.83	45.59	5.90
100%	1.65	41.60	5.86

precisions (i.e., the expense and LIL metric).

As indicated in Table VII, Hybrid-MUSE with mutant sampling rate 1% still outperforms the all SBFL techniques, i.e. the expense of Hybrid-MUSE with mutant sampling rate 1% is 6.20% which is smaller than the expense of Op2 (9.25%). And Hybrid-MUSE with mutant sampling rate 40% outperforms the all SBFL techniques in terms of LIL value, i.e., the LIL of MUSE with sampling rate 40% is 5.96% which is smaller than the LIL of Jaccard (6.04). Table VII also shows that the precision of Hybrid-MUSE increases as mutant sampling rate of Hybrid-MUSE increases. Thus, a user may choose appropriate mutant sampling rate to achieve fault localization result precise enough for his/her purpose fast.

VII. DISCUSSIONS

A. Why does Hybrid-MUSE work well?

As shown in Section VI-C and Section VI-E, Hybrid-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 the core component of Hybrid-MUSE, MUSE, 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 nonetheless captures the chain of control and data dependencies that are relevant to the failure and provides a guidance towards the location of the fault.

B. Hybrid-MUSE and Test Suite Balance

One advantage Hybrid-MUSE has over SBFL is that Hybrid-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 [10], [21], [18].

Hybrid-MUSE does not require the test suite to have many passing test cases. To illustrate the point, we purposefully calculated the Hybrid-MUSE without any test cases that passed before mutation (this effectively means that we only use the first term of the MUSE metric and Jaccard metric without the passing test case term). On average, Hybrid-MUSE ranked the faulty statement within the top 12.93% on average, and among the 1% for 35 out of 51 faulty versions we studied. In contrast, SBFL techniques Jaccard, Ochiai, and Op2 that considered all passing and failing test cases rank the faulty statement among the 1% for 14 faulty versions. Hybrid-MUSE is 2.5 ($= 35/14$) times more precise than the SBFL techniques, when considering the faulty versions that require a developer to inspect less than 1% of executed statements to locate the faulty statement.

More interestingly, Hybrid-MUSE does not require the test suite to have many failing test cases. Considering that previous work [21], [18] focused on producing more failing test cases to improve the precision, this is an important observation. We purposefully calculated Hybrid-MUSE 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 [6], Hybrid-MUSE could still rank the faulty statement among the top 27.37% on average. Intuitively, SBFL techniques require many failing executions to identify where a fault is, whereas Hybrid-MUSE is relatively free from this constraint because it also identifies where a fault *is not*.

This advantage is due to the fact that the core component of Hybrid-MUSE, 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, Hybrid-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. [31]. 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 (not Hybrid-MUSE, which is a combination of MUSE and SBFL), 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. [12]. GenProg-FL [31] 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 VIII 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 [31] (we

[6]Since Jaccard metric without the failing test case term assigns 0 to all executed statements, Hybrid-MUSE without any test cases that failed before mutation only uses the second term of the MUSE metric to compute suspiciousness scores.

TABLE VIII
EXPENSE, LIL, AND NCP SCORES ON `LOOK UTX 4.3`

Fault localization technique	% of executed statements examined	Locality information loss (LIL)	Average of 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

obtained the average NCP score from 100 runs). Table VIII 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 (MUSE : 3.52 < Op2 : 3.77 < Ochiai : 3.83 < Jaccard : 3.89) and the NCP scores (MUSE : 25.3 < Op2 : 31.0 < Ochiai : 32.2 < 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.

VIII. Related Work

Among the wide range of fault localization approaches including program state analysis [14], [46] and machine learning [2], the most widely studied has been the spectrum-based approaches [41], [23]. SBFL techniques such as Tarantula [20] and Ochiai [28] have been extensively studied both empirically [19], [32] and theoretically [42]. Other techniques expand the input data to call sequences [6] and *du*-pairs [33]. While the spectrum-based approaches are limited in their accuracy by the basic block structure [36], 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 [30]. Other techniques attempt to improve the accuracy of SBFL using dynamic data dependency and causal-inference model [8], [9], or combining the SAT-based fault localization [25] with the spectrum-based ones [10].

The idea of generating *diverse program behaviours* to localize a fault more effectively has been utilized by several studies. For example, Cleve and Zeller [14] 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. [48], 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. [17] 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 [48], [17], [5] in particular depend on the first conjecture of MUSE) or rely on a different conjecture [14]. Moreover, MUSE does not require any other infrastructure than a mutation tool, because it *directly* changes program source code to utilize the conjectures (Section V-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 [42]. In the empirical evaluation, however, MUSE outperformed Op2 SBFL metric [27], which is the known best SBFL technique.

Yoo showed that risk evaluation formulas for SBFL can be automatically evolved using Genetic Programming (GP) [44]. Some of the evolved formulas were proven to be equivalent to the known best metric, Op2 [43]. 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 [29]. 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. [47], 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.

IX. Conclusion and Future Work

We have presented Hybrid-MUSE, a new fault localization technique that combines MUtation-baSEd fault localization (MUSE) and Spectrum-Based Fault Localization (SBFL) technique. The core component of Hybrid-MUSE, MUSE, localizes faults 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. Also, the SBFL component of Hybrid-MUSE helps to localize faulty statements more precisely, by adding suspiciousness of SBFL to that of MUSE. The results of empirical evaluation show that Hybrid-MUSE can not only outperform the state-of-the-art SBFL techniques significantly, but also provide a practical fault localization solution. Hybrid-MUSE is more than 5.6 times precise compared to Op2, which is the best

known SBFL technique; Hybrid-MUSE also ranks the faulty statement at the top for 18 out of the 51 faulty versions, and among the top ten for another 38 versions. In addition, we confirmed that Hybrid-MUSE improves the precision of MUSE 4.08 times over the Expense metric thanks to the combined SBFL component. 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.

Future 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] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98. IEEE Computer Society, 2007.

[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. Causal inference for statistical fault localization. In *ISSTA*, 2010.

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

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

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

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

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

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

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

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

[24] 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.

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

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

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

[28] 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.

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

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

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

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

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

[34] S.Artzi, J.Dolby, F.Tip, and M.Pistoia. Directed test generation for effective fault localization. In *ISSTA*, 2010.

[35] S.Moon, Y.Kim, M.Kim, and S.Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, 2014.

[36] 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.

[37] S.Yoo. Evolving human competitive spectra-based fault localisation techniques. 2012.

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

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

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

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

[42] 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*, 22(4):31, 2013.

[43] 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.

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

[45] 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.

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

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

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