Transforming Programs and Tests in Tandem for Fault Localization

XIA LI, The University of Texas at Dallas, USA
LINGMING ZHANG, The University of Texas at Dallas, USA

Localizing failure-inducing code is essential for software debugging. Manual fault localization can be quite tedious, error-prone, and time-consuming. Therefore, a huge body of research efforts have been dedicated to automated fault localization. Spectrum-based fault localization, the most intensively studied fault localization approach based on test execution information, may have limited effectiveness, since a code element executed by a failed tests may not necessarily have impact on the test outcome and cause the test failure. To bridge the gap, mutation-based fault localization has been proposed to transform the programs under test to check the impact of each code element for better fault localization. However, there are limited studies on the effectiveness of mutation-based fault localization on sufficient number of real bugs. In this paper, we perform an extensive study to compare mutation-based fault localization techniques with various state-of-the-art spectrum-based fault localization techniques on 357 real bugs from the Defects4J benchmark suite. The study results firstly demonstrate the effectiveness of mutation-based fault localization, as well as revealing a number of guidelines for further improving mutation-based fault localization. Based on the learnt guidelines, we further transform test outputs/messages and test code to obtain various mutation information. Then, we propose TraPT, an automated Learning-to-Rank technique to fully explore the obtained mutation information for effective fault localization. The experimental results show that TraPT localizes 65.12% and 94.52% more bugs within Top-1 than state-of-the-art mutation and spectrum based techniques when using the default setting of LIBSVM.

CCS Concepts: • **Software and its engineering → Software testing and debugging**;

Additional Key Words and Phrases: Fault localization, Mutation testing, Code transformation

ACM Reference Format:
Xia Li and Lingming Zhang. 2017. Transforming Programs and Tests in Tandem for Fault Localization. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 92 (October 2017), 30 pages. https://doi.org/10.1145/3133916

92

1 INTRODUCTION

In the software development, *fault localization* denotes the process of localizing the potential faulty code locations to help further fix the corresponding software faults. Due to the huge code volume in modern programs, fault localization is a time-consuming and error-prone phase. As a result, automated fault localization techniques have been widely studied in recent years [Abreu et al. 2007; Artzi et al. 2010; B Le et al. 2016; Fey et al. 2008; Griesmayer et al. 2007; Papadakis and Le Traon 2014; Xuan and Monperrus 2014]. The basic idea of fault localization is to rank code elements (e.g., program methods or statements) automatically according to the descending order of their suspiciousness values (i.e, probability to be faulty) to assist developers in debugging.

Authors' addresses: Xia Li The University of Texas at Dallas, USA, xxl124730@utdallas.edu; Lingming Zhang The University of Texas at Dallas, USA, lingming.zhang@utdallas.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Association for Computing Machinery.
2475-1421/2017/10-ART92
https://doi.org/10.1145/3133916

One of the most intensively studied fault localization methodologies is *spectrum-based fault localization* (also known as coverage-based fault localization), which uses the coverage information of failed/passed tests to measure the suspiciousness values of code elements [Abreu et al. 2007; Jones and Harrold 2005; Naish et al. 2011; Santelices et al. 2009; Wong et al. 2007]. The basic idea of spectrum-based fault localization is that the code elements that are primarily executed by failed tests are more suspicious than the elements that are primarily executed by passed tests. To date, various suspiciousness computation formulae have been utilized for spectrum-based fault localization, e.g., Tarantula [Jones and Harrold 2005], Ochiai [Abreu et al. 2006], and Jaccard [Abreu et al. 2007]. After the fault localization process, developers can manually go through the ranked list to find out the locations of program faults efficiently.

Although intensively studied, spectrum-based fault localization may have limited support for real-world debugging practice according to recent studies [Parnin and Orso 2011]. One key reason is that elements executed by the failed tests do not necessarily impact the program behavior and contribute to the test failure while faulty elements may also be executed by passed tests coincidentally. To bridge the gap between coverage and impact information, researchers proposed *mutation-based fault localization* [Moon et al. 2014; Papadakis and Le Traon 2012, 2015; Zhang et al. 2013], which transforms program source code based on mutation testing to check the impact of each code element on the test outcomes. *Mutation testing* was originally proposed to evaluate test effectiveness by injecting artificial faults into the program under test [DeMillo et al. 1978; Hamlet 1977; Jia and Harman 2011]. In mutation-based fault localization, mutation testing is used to inject changes to each code element to check its impact on the test outcomes. The first mutation-based fault localization technique is called Metallaxis [Papadakis and Le Traon 2012, 2015]. The basic idea of Metallaxis is that if one mutant incurs different failure outputs/messages for failed tests, the corresponding code element of this mutant may have high impact on failed tests, and may be the cause of the test failures. Metallaxis directly applies the existing spectrum-based formulae to the impact information to compute code element suspiciousness. Another representative mutation-based fault localization is called MUSE [Moon et al. 2014]. The basic heuristic of MUSE is that mutating faulty elements may mask the fault and make some failed tests pass, while mutating correct elements may lead to more faulty elements besides existing faulty elements, making more tests fail. Based on this insight, MUSE uses a newly designed formula for localizing faulty code elements.

While Metallaxis and MUSE have been demonstrated to outperform the traditional spectrum-based fault localization techniques [Moon et al. 2014; Papadakis and Le Traon 2015], the existing studies usually use seeded bugs or a small number of real bugs. For example, the real bugs used in the Metallaxis (38 real bugs) and MUSE (3 real bugs) work all come from Space, a C program with less than 10K lines of code [Moon et al. 2014; Papadakis and Le Traon 2015]. Therefore, there lack extensive studies investigating the effectiveness of mutation-based fault localization on a sufficient number of real bugs from modern real-world projects. In this paper, we perform an extensive study of the two representative mutation-based fault localization techniques (Metallaxis [Papadakis and Le Traon 2015] and MUSE [Moon et al. 2014]), on 357 real bugs from the modern Defects4J benchmark (with subjects ranging from 22K to 90K lines of code). Our study results firstly confirm that mutation-based fault localization techniques can significantly outperform state-of-the-art spectrum-based techniques for the majority of the studied real bugs, e.g., the MAR (Mean Average Ranking of bugs) is 44.99 for the spectrum-based Ochiai, but only 15.28 for Metallaxis with the Ochiai formula. Meanwhile, the study also reveals various limitations of mutation-based fault localization, e.g., mutation-based fault localization may be ineffective when too few or too many mutants can impact the outputs/messages of failed tests (Section 5.1).

Based on our study findings, besides transforming source code via mutation testing, we propose to also transform test messages/outputs and test code to capture detailed mutation information for more effective mutation-based fault localization. First, we transform the test message/output data to obtain different types of failure output/message information. In previous mutation-based techniques, MUSE only considers pass/fail information of tests, while Metallaxis only considers the detailed changed failure outputs/messages with stack traces. Actually, there are other different intermediate types of failure messages (such as simply printing exception types, or exception types with messages), based on which a mutant may have different sets of impacted tests (e.g., a mutant may change a test's exception message but not its exception type). Based on this intuition, we record different types of test failure information to investigate their impacts on mutation-based fault localization. Second, we further transform the test code to record the detailed execution result for each assertion within each test. The intuition of this extension is that in modern programming languages and testing paradigms (e.g., Java programs with JUnit tests), each test may consist of multiple assertions, while only one outcome is reported for all assertions at the traditional test level. Furthermore, if one assertion fails, the test execution will be aborted and the remaining assertions will not be executed. Therefore, we perform test code transformation to record the execution outcomes of all assertions, and utilize the detailed impact of each program element on each assertion to further improve mutation-based fault localization.

We then further propose TraPT (**Tra**nsforming **P**rograms and **T**ests in tandem for fault localization) to combine the strengths of various captured mutation information via Learning-to-Rank techniques [Liu 2009]. For each program element, we compute a set of different suspiciousness values based on mutation-based fault localization using different mutation information (e.g., obtained via different failure message types and levels) as well as traditional spectrum-based techniques. Then, such suspiciousness values can be treated as the features/attributes to predict whether each code element is buggy or not based on historical bug data. Our experimental results demonstrate that TraPT (with the default setting of LIBSVM [Chang and Lin 2011]) incorporating various failure message types can localize 94.52% and 65.12% more bugs within Top-1 than state-of-the-art spectrum and mutation based techniques, respectively. Also, including assertion-level mutation information can further improve the fault localization results of TraPT by 14.77% in localizing Top-1 bugs. Finally, our experimental results also show that historical bug data from other projects can help boost the fault localization results even more. In summary, the paper makes the following contributions:

- **Study.** We present an extensive study on 357 real bugs from Defects4J to evaluate state-of-the-art mutation-based fault localization techniques, MUSE and Metallaxis. The study results firstly confirm the effectiveness of mutation-based fault localization on real bugs, and also reveal various guidelines for further improving mutation-based fault localization.
- **Extensions.** Based on the guidelines learnt from our study, we propose two extensions to further transform test outputs/messages and test code to obtain useful mutation information for better fault localization. The experimental results show that different failure message types (obtained via test output/message transformation) and different mutation levels (obtained via test code transformation) can all potentially help with fault localization.
- **Learning-to-Rank Technique.** We further propose TraPT, a Learning-to-Rank technique to incorporate various mutation information obtained from our extensions to help with better fault localization. The experimental results show that TraPT can greatly outperform existing state-of-the-art spectrum-based and mutation-based techniques.

2 BACKGROUND

In this section, we introduce the preliminaries on spectrum-based fault localization (Section 2.1) and mutation-based fault localization (Section 2.2). Finally, we also illustrate the basic ideas of spectrum-based and mutation-based fault localization using a simple example program (Section 2.3).

2.1 Spectrum-based Fault Localization

Spectrum-based fault localization [Abreu et al. 2006, 2007; Jones and Harrold 2005; Liblit et al. 2005; Naish et al. 2011; Santelices et al. 2009; Wong et al. 2007] is one of the most intensively studied approaches for fault localization. This approach takes the coverage information of all tests in the test suite (including passed and failed ones) as input, and applies various formulae (e.g., based on statistical analysis or other heuristics) to compute the suspiciousness value of each code element (e.g., statement or method). The insight is that code elements primarily executed by failed tests are more suspicious than the ones primarily executed by passed tests. The output of the approach would be a ranked list of code elements for manual inspection. To date, a number of spectrum-based fault localization techniques have been proposed, e.g., Tarantula [Jones and Harrold 2005], Ochiai [Abreu et al. 2006], Jaccard [Abreu et al. 2007], SBI [Liblit et al. 2005], and so on. Almost all the proposed techniques rely on the following information: (1) the set of all failed/passed tests, i.e., T_f/T_p, (2) the set of failed/passed tests executing element e, i.e., $T_f(e)/T_p(e)$, and (3) the set of failed/passed tests that do not execute element e, i.e., $T_f(\bar{e})/T_p(\bar{e})$. For example, the suspiciousness value of element e based on the SBI formula will be calculated as $Susp(e) = \frac{|T_f(e)|}{|T_f(e)|+|T_p(e)|}$. Then, developers can go through the ranked list to manually identify the actual faulty elements efficiently. The higher the faulty elements get ranked, the less effort the developers may spend in identifying the faults.

2.2 Mutation-based Fault Localization

One issue of spectrum-based fault localization is that even though some code elements can be covered by failed tests, they may not have any impact on the program's correctness and contribute to the failures. To improve spectrum-based fault localization, mutation-based fault localization [Moon et al. 2014; Papadakis and Le Traon 2012, 2014, 2015] is proposed to mutate the subject programs to check the impact of each code element on the test outcomes. Metallaxis [Papadakis and Le Traon 2015] and MUSE [Moon et al. 2014] are two representative mutation-based fault localization techniques. The two techniques both transform the program source code based on mutation testing and then analyze the impact of each mutant on tests.

Metallaxis. Metallaxis makes the assumption that mutants of same program element frequently exhibit similar behaviors and mutants of diverse program elements exhibit different behaviors. Since a fault can also be viewed as a mutant, it may be similar to other mutants of same element and can be located by examining these mutants based on the above observation. Metallaxis treats the mutants that can impact the detailed test outputs/messages as being able to impact the tests (Note that for a passing test, any mutant causing it to fail can be treated as changing the test failure message from `NULL` to non-`NULL`, thus impacting the tests). In this way, mutants impacting failed tests indicate that their corresponding code elements may have caused the test failures, whereas mutants impacting passed tests indicate that their corresponding code elements may not be faulty (otherwise the passed tests would have failed). Then Metallaxis extended spectrum-based fault localization formulae, treating all mutants impacting the tests as covered elements while the others as uncovered elements, to calculate the suspiciousness value of each mutant. At last, the maximum suspiciousness value of mutants of a corresponding code element is returned as the suspiciousness value of the code element. Assume that the SBI formula is applied to Metallaxis, the suspiciousness

of element e can be calculated as:

$$\max_{m \in M(e)} \left(\frac{|T_f^{(m)}(e)|}{|T_f^{(m)}(e)| + |T_p^{(m)}(e)|} \right) \tag{1}$$

In the formula, $M(e)$ denotes the set of all mutants on element e, $|T_f^{(m)}(e)|$ denotes the number of failed tests that have been impacted by mutant m, $|T_p^{(m)}(e)|$ denotes the number of passed tests that have been impacted by mutant m.

MUSE. The basic insight of MUSE is two-fold: (1) mutating faulty elements may cause more failed test cases to pass than mutating correct elements; (2) mutating correct elements may cause more passed test cases to fail than mutating faulty elements. The reason is that mutating faulty elements may mask the fault and make some failed tests pass, while mutating correct elements may lead to more faulty elements besides existing faulty elements, making more tests fail. Therefore, based on this insight, MUSE computes the suspiciousness value of each program element e, i.e., $Susp(e)$, based on the following formula:

$$\frac{1}{|M(e)|} \sum_{m \in M(e)} \left(\frac{|T_f^{(m)}(e)|}{|T_f|} - \alpha * \frac{|T_p^{(m)}(e)|}{|T_p|} \right) \tag{2}$$

In the formula, $M(e)$ is the set of all mutants on element e, T_p/T_f denotes the set of originally passed/failed tests, $T_f^{(m)}(e)$ denotes the set of originally failed tests that pass with mutant m inserted, and $T_p^{(m)}(e)$ denotes the set of originally passed tests that fail with mutant m inserted. Thus, $\frac{|T_f^{(m)}(e)|}{|T_f|}$ is the proportion of failed tests that are changed into passed after mutant m mutates element e to all the originally failed tests, reflecting the first insight of MUSE. $\frac{|T_p^{(m)}(e)|}{|T_p|}$ is the proportion of passed tests that are changed into failed after mutant m mutates e to all originally passed tests, reflecting the second insight of MUSE. The weight α is used to balance the above two proportions, and is calculated as $\frac{f2p}{|T_f|} * \frac{|T_p|}{p2f}$, where $f2p$ denotes the total number of failed tests changed into passed while $p2f$ denotes the total number of passed tests changed into failed during mutation testing.

2.3 Example

In this section, we use an example program shown in Figure 1 to illustrate existing spectrum-based and mutation-based fault localization techniques (e.g., SBI [Liblit et al. 2005], MUSE [Moon et al. 2014] and Metallaxis [Papadakis and Le Traon 2015]). In the example program, the left half presents its source code for dealing with bank account operations such as `getBalance`, `withdraw`, and `deposit`, while the right half presents its corresponding test suites consisting of three JUnit tests. Note that in the code, there is a fault in the `deposit` method (marked with underline), where `saving=saving-v` should be `saving=saving+v`. Due to this fault, tests

Table 1. Spectrum-based fault localization

	Statements	Coverage			$\lvert T_f(e)\rvert$	$\lvert T_p(e)\rvert$	SBI	Rank
		TC1	TC2	TC3				
	`BankAcnt(String a){`							
s1	`account=a;`	✓	✓	✓	2	1	0.67	8
s2	`saving=100;`	✓	✓	✓	2	1	0.67	8
s3	`bank="ABank";}`	✓	✓	✓	2	1	0.67	8
	`double getBalance(){`							
s4	`return saving; }`	✓	✓	✓	2	1	0.67	8
	`double withdraw(double v){`							
s5	`if(saving>=v) {`	✓		✓	2	0	1.00	3
s6	`saving = saving-v;`	✓		✓	2	0	1.00	3
s7	`return v;`	✓		✓	2	0	1.00	3
	`}else{`							
s8	`return 0;} }`				0	0	0.00	9
	`void deposit (double v){`							
s9	`saving = saving-v; }`	✓	✓	✓	2	1	0.67	8
		F	P	F				

```
public class BankAcnt{
 private static String bank;
 public String account;
 private double saving;
 public BankAcnt(String a){
   account=a;
   saving=100;
   bank=''ABank'';
 }
 public double getBalance(){
   return saving;
 }
 public double withdraw (double v) {
  if(saving>=v) {
   saving = saving-v;
   return v;
  }else{
   return 0;
   }
 }
 public void deposit (double v){
    saving = saving-v;// FAULT
 }
}
```

```
public class TestBankAcnt{
 @Test
 public void TC1() {
   BankAcnt acnt=new BankAcnt(``acnt1'');
   double old_balance=acnt.getBalance();
   double amnt=acnt.withdraw(20);
   acnt.deposit(amnt);
   assertEquals(old_balance, acnt.getBalance(),0.01);
   acnt.withdraw(0);
   assertTrue(acnt.getBalance()>0); }
 @Test
 public void TC2() {
  BankAcnt acnt=new BankAcnt(``acnt1'');
  acnt.deposit(0);
  assertEquals(100,acnt.getBalance(),0.01); }
 @Test
 public void TC3() {
  BankAcnt acnt1=new BankAcnt(``acnt1'');
  BankAcnt acnt2=new BankAcnt(``acnt2'');
  double amount=acnt1.withdraw(80);
  assertEquals(20,acnt1.getBalance(),0.01);
  acnt2.deposit(amount);
  assertEquals(180,acnt2.getBalance(),0.01); }
}
```

Fig. 1. Example code and corresponding test suite.

TC1 and TC3 fail while only TC2 passes. Next, we will show how different fault localization techniques perform in localizing the fault given the test failure information.

SBI. Traditional spectrum-based fault localization techniques [Abreu et al. 2006, 2007; Jones and Harrold 2005; Liblit et al. 2005; Naish et al. 2011; Santelices et al. 2009; Wong et al. 2007] have been intensively studied. They mainly utilize the coverage information and test pass/fail results to determine the suspiciousness of each code element. The basic insight is that code elements that are primarily executed by failed tests rather than passed tests have potentially higher probability of being faulty. Table 1 presents the results of traditional SBI technique [Liblit et al. 2005] in localizing the fault in Figure 1. In the table, Columns 1-2 present the executable statements under consideration, with the faulty statement marked in red. Column 3 presents the coverage information of each test (✓ denotes that the corresponding element is covered by the corresponding test), with the final row presenting the test outcomes (F for failed while P for passed). Column 4 presents the number of failed/passed tests that execute each code element, i.e., $|T_f(e)|/|T_p(e)|$ for each element e. Then, based on the SBI formula ($Susp_{SBI}(e) = \frac{|T_f(e)|}{|T_f(e)|+|T_p(e)|}$), Column 5 presents the suspiciousness value and rank[1] for each code element. Unfortunately, the technique computes the faulty element (i.e., s9) as the least suspicious one among the executed statements. The reason is that although not all executed statements contribute to test failures, they are all executed by the failed tests coincidentally, while the faulty statement s9 is also executed by the passed test without triggering any failure.

Metallaxis. To achieve more precise fault localization, mutation-based fault localization (e.g., Metallaxis [Papadakis and Le Traon 2015] and MUSE [Moon et al. 2014]) mutates each code element to check its impact on the test outcomes. Table 2 presents the result using Metallaxis and Table 3 presents the result using MUSE. Column 3 in both tables presents the mutants that Metallaxis and MUSE use to check the impact of each code element. For the sake of simplicity, we generate at most two mutants for each statement, resulting in a total of 11 mutants. In Table 2, Column 4

[1]For the tied elements, we show the lowest applicable ranking.

Table 2. Fault localization using Metallaxis

	Statements	Mutants	Impacts on Tests			$\lvert T_f^{(m)}(e)\rvert$	$\lvert T_p^{(m)}(e)\rvert$	Susp	Rank
			TC1	TC2	TC3				
	`BankAcnt(String a){`								
s1	`account=a;`	`m1:account="Hello"`				0	0	0.00	9
s2	`saving=100;`	`m2:saving=50`	X	X	X	2	1	0.67	6
s3	`bank="ABank";}`	`m3:bank="A"`				0	0	0.00	9
	`double getBalance(){`								
s4	`return saving; }`	`m4:return 0`	X	X	X	2	1	0.67	6
	`double withdraw(double v){`								
s5	`if(saving>=v) {`	`m5:if(saving<v) {`	X		X	2	0	1.00	4
		`m6:if(saving==v){`	X		X	2	0	1.00	
s6	`saving = saving-v;`	`m7:saving=saving+v`	X		X	2	0	1.00	4
s7	`return v;`	`m8:return 0;`	X		X	2	0	1.00	4
	`}else{`								
s8	`return 0;} }`	`m9:return v`				0	0	0.00	9
	`void deposit (double v){`								
s9	`saving = saving-v; }`	`m10:saving=saving+v`	X		X	2	0	1.00	4
		`m11:saving=saving/v`	X	X	X	2	1	0.67	
			F	P	F				

Table 3. Fault localization using MUSE

	Statements	Mutants	Impacts on Tests			$\lvert T_f^{(m)}(e)\rvert$	$\lvert T_p^{(m)}(e)\rvert$	Susp	Rank
			TC1	TC2	TC3				
	`BankAcnt(String a){`								
s1	`account=a;`	`m1:account="Hello"`				0	0	0	7
s2	`saving=100;`	`m2:saving=50`		P→F		0	1	-0.83	9
s3	`bank="ABank";}`	`m3:bank="A"`				0	0	0	7
	`double getBalance(){`								
s4	`return saving; }`	`m4:return 0`		P→F		0	1	-0.83	9
	`double withdraw (double v) {`								
s5	`if(saving>=v) {`	`m5:if(saving<v) {`	F→P			1	0	0.5	2
		`m6:if(saving==v){`	F→P			1	0		
s6	`saving = saving-v;`	`m7:saving=saving+v`	F→P			1	0	0.5	2
s7	`return v;`	`m8:return 0;`				0	0	0	7
	`}else{`								
s8	`return 0;} }`	`m9:return v`				0	0	0	7
	`void deposit (double v){`								
s9	`saving = saving-v; }`	`m10:saving=saving+v`	F→P		F→P	2	0	0.085	3
		`m11:saving=saving/v`		P→F		0	1		
			F	P	F				

presents the mutants impacting the three tests. We use "X" to denote that the mutant impacts the corresponding test, i.e., the test failure messages changed after mutation. Then we can apply the SBI formula to Metallaxis to calculate the suspiciousness values of all mutants on each element and assign the maximum value to corresponding element. Based on Metallaxis, the suspiciousness value of statement `s9` can be calculated as 1.00 since mutant `m10` has highest value among mutants `m10` and `m11` of this statement. Finally, according to suspiciousness values of all statements, statement `s9` can be ranked 4th by Metallaxis, outperforming the corresponding traditional spectrum-based technique.

MUSE. In Table 3 , Column 4 presents the test outcome changes for each test on each mutant. "F→P" denotes that an originally failed test now passes on the mutant, while "P→F" denotes that an originally passed test now fails on the mutant. Note that the blank cells denote the cases where the test outcomes do not change before and after the mutation. In total, there are 5 tests changed from failed to passed, i.e., $f2p$ =5. and 3 tests changed from passed to failed, i.e., $p2f$ =3. Therefore, MUSE calculates α as $\frac{f2p}{|T_f|} * \frac{|T_p|}{p2f} = 5/2 * 1/3 = 0.83$. Using this α value, suspiciousness values of different statements can be calculated based on Equation 2. For example, the suspiciousness value of statement `s9` is $1/2 * ((2/2 - 0.83 * 0) + (0 - 0.83 * 1/1)) = 0.085$, where $|T_f^{(m_{10})}(\texttt{s9})| = 2$ and

```
java.lang.NumberFormatException: For input string: "80000000"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:495)
        at java.lang.Integer.valueOf(Integer.java:556)
        at java.lang.Integer.decode(Integer.java:984)
        at org.apache.commons.lang3.math.NumberUtils.createInteger(NumberUtils.java:684)
        at org.apache.commons.lang3.math.NumberUtils.createNumber(NumberUtils.java:474)
        at org.apache.commons.lang3.math.NumberUtilsTest.TestLang747(NumberUtilsTest.java:256)
```

Fig. 2. JUnit failure message example

$|T_p^{(m_{10})}(\texttt{s9})|= 0$ for mutant `m10`, while $|T_f^{(m_{11})}(\texttt{s9})|= 0$ and $|T_p^{(m_{11})}(\texttt{s9})|= 1$ for mutant `m11`. Based on the impacts of each code element's mutant(s) on each test, MUSE ranks `s5` and `s6` above the faulty statement `s9` since their mutants only change failed tests to passed ones while `s9`'s mutant may change passed tests into failed ones. However, the rank of `s9` indicates that MUSE still outperforms the traditional SBI technique.

3 APPROACH

While traditional mutation-based fault localization simply transforms program source code via mutation, in this section, we introduce how we further transform test outputs/messages and test code to achieve more effective fault localization. Section 3.1 presents why and how we record different types of test failure outputs/messages in order to investigate their impacts on fault localization. More specifically, we transform the test output/message information to distill 4 different types of test failure messages. Section 3.2 further investigates different mutation-based fault localization techniques not only at the test level but also at the assertion level. More specifically, at the assertion level, we further transform the test code to capture the detailed execution information of each assertion in each test. Then the impact information of each mutant on each assertion can be utilized to compute more precise fault localization information (i.e., simply using the existing test-level techniques by treating each assertion as a test). Finally, we investigate how to improve mutation-based fault localization by incorporating all the above traced mutation information via the Learning-to-Rank algorithm [Liu 2009] (Section 3.3).

3.1 Test Output/Message Transformation

When a test fails, certain failure output or message will be thrown. The most popular one is the outcome of this test, i.e., whether this test passes or fails. For a failed test, however, the reasons that cause it to fail may be different even though this test is always marked as "failed" in different failure cases. First, a test can fail due to different types of exceptions, eg. `ArrayIndexOutOfBoundsException` (indicating that an array has been accessed with an illegal index) or `NullPointerException` (indicating that the accessed object is `NULL`) in Java. Second, although the cause of failure may be the same type of exception, the detailed message of this exception may not be same. For example, the failure messages of `ArrayIndexOutOfBoundsException` may be different when accessing the array with different illegal indexes. Besides the exception information above, the detailed stack trace[2] information also matters for a test failure. The test failing with the same exception type and message may not have exactly the same stack trace information due to different calling contexts. To illustrate, Figure 2 shows the complete JUnit test failure information for bug Lang-1 of the Defects4J benchmark [Just et al. 2014]. In this example, test `TestLang747` is failing since an exception has been thrown. The type of the exception is `NumberFormatException` and the corresponding exception message is "`For input string:80000000`", indicating that incorrect number format

[2] A stack trace, which is also called stack backtrace, is a list of the method calls on the current stack when an exception is thrown during the execution of a program.

of string `80000000` incurs the failure. The detailed stack trace information is further printed after the exception message to help with fault diagnosis.

Metallaxis [Papadakis and Le Traon 2015] evaluates the effectiveness of fault localization by investigating changes of all available failure messages, but did not consider the impact of different types of failure messages. MUSE [Moon et al. 2014] uses the changes of pass/fail outcome of each test for fault localization, but ignores more detailed failure information. Therefore, Metallaxis may be too sensitive to test failure message changes while MUSE may be too insensitive. In this paper, we further extend both Metallaxis and MUSE by recording 4 different types of failure outputs/messages for fault localization as following: (1) Type1: pass/fail information, (2) Type2: exception type information, (3) Type3: exception type and message, and (4) Type4 : exception type, message and the full stack trace of the exception. Intuitively, Type4 provides more detailed information than other types so that it may be most effective for fault localization. However, comparing Table 2 with Table 3 in Section 2.3, Metallaxis with more detailed failure messages (i.e., Type4) may actually be inferior to MUSE with less detailed failure message (i.e., Type1). The reason is that Metallaxis has 6 statements (3 for MUSE) with mutants impacting failed tests due to the sensitive failure messages, making many statements share similar suspiciousness values with the actual buggy statement. Therefore, we transform the test outputs/messages to consider all the 4 different types of failure messages to investigate their effectiveness.

3.2 Test Code Transformation

Various programming languages and unit testing frameworks support assertions to verify the correctness of program execution, e.g., the JUnit testing framework[3] for Java. For one test execution, if any assertion of the test is not satisfied, the test will throw an assertion exception and abort the test execution. For example, in Java, class `java.lang.AssertionError` is used to indicate assertion violation exceptions. If an assertion fails, an exception of type `AssertionError` will be thrown from the assertion statement without being caught, incurring early execution termination. In this case, the following assertions are not executed and the detailed outcome of each assertion will not be available. In order to investigate the effectiveness of mutation-based fault localization at the assertion level, we further propose to transform the test code to catch all the assertion violation exceptions that may be thrown to force the test code to verify following assertion outcomes in case of early assertion violation. In addition, we also transform the test code to record the detailed value(s) checked by each assertion to detect any assertion-level impact information. For example, if any mutant causes an assertion to check a different value, we treat that mutant as impacting the assertion.

Note that after catching all the assertion exceptions, there still may be faults that are detected by other types of exceptions rather than assertions, e.g., `NullPointerException`. Therefore, we also record the different types of failure messages of the transformed tests automatically via implementing a runtime listener for test outcome events, since the transformed tests may fail due to other exceptions than assertions. For the uniformness and conciseness, we actually treat this kind of exceptions of the transformed tests as a default assertion (i.e., `a0`). That is, for any test with n assertions, we will record the checked value of each assertion as well as the transformed test outcome (default assertion), resulting in $n + 1$ assertion outcomes that can capture the detailed test execution information. Note that for the default assertion, we also have four different types of failure messages to configure (shown in Section 3.1). In the assertion-level mutation-based fault localization, the prior mutation-based fault localization techniques can be directly applied, and the only difference is that now each assertion is treated as a test for fault localization.

[3] http://junit.org/junit4/

Table 4. Fault localization using MUSE on assertion-level

	Statements	Mutants	Impacts on Assertions					$\lvert T_f^{(m)}(e)\rvert$	$\lvert T_p^{(m)}(e)\rvert$	Susp	Rank
			TC1-a1	TC1-a2	TC2-a1	TC3-a1	TC3-a2				
	BankAcnt(String a){										
s1	account=a;	m1:account="Hello"						0	0	0	7
s2	saving=100;	m2:saving=50			P→F	P→F		0	2	-0.67	9
s3	bank="ABank";}	m3:bank="A"						0	0	0	7
	double getBalance(){										
s4	return saving; }	m4:return 0	F→P	P→F	P→F	P→F		1	3	-0.5	8
	double withdraw (double v) {										
s5	if(saving>=v) {	m5:if(saving<v) { m6:if(saving==v){	F→P F→P			P→F P→F		1 1	1 1	0.17	2
s6	saving = saving-v;	m7:saving=saving+v	F→P			P→F		1	1	0.083	3
s7	return v;	m8:return 0;						0	0	0	7
	}else{										
s8	return 0;} }	m9:return v						0	0	0	7
	void deposit (double v){										
s9	saving = saving-v; }	m10:saving=saving+v m11:saving=saving/v	F→P 		 P→F		F→P 	2 0	0 1	0.33	1
			F	P	P	P	F				

To illustrate how assertion-level mutation information improves fault localization, we split the 3 original tests in Figure 1 based on the number of assertions that they have. Shown in Table 4, the original `TC1` result is split into two assertion-level results: `TC1-a1` and `TC1-a2` (Note that for the ease of illustration, we do not show the default assertions). Then we apply MUSE to the assertion-level results to perform fault localization (the result is similar for Metallaxis). The result shows that MUSE at the assertion level can rank the buggy statement as 1st due to detailed assertion-level information, outperforming the traditional mutation-based techniques.

3.3 Learning-to-Rank Fault Localization

Learning-to-Rank is a supervised machine learning technique for solving ranking problems in the field of information retrieval [Liu 2009]. There are two phases of Learning-to-Rank : (1) the learning phase and (2) the ranking phase. The training data for Learning-to-Rank consists of queries and documents and their ground-truth relevance degree. Then, in the *learning* phase, Learning-to-Rank takes the specific attributes of documents and queries as different features, e.g., cosine similarity and proximity value. A ranking model can then be learned in learning phase to predict the relevance labels for new queries and documents with computed features. The ranking model is usually an optimal combination of weights for different features. In the *ranking* phase, test data including new queries and documents are passed into the ranking model built in the learning phase. Finally, the ranking model can return a ranked list of documents for the given queries for further analysis.

There are three major categories of approaches in Learning-to-Rank : (1) pointwise approaches, (2) pairwise approaches, and (3) listwise approaches. In the *pointwise* approaches, for a given query, each document in the training data has its own label. In the *pairwise* approaches, each pair of two documents will be computed a label based to their ordering for the given query. In the *listwise* approaches, the order of a list of documents will be considered for prediction. Note that in the fault localization problem, we only care about the capability of distinguishing faulty elements from correct ones. Besides ranking faulty elements over correct ones, there is no further relationships among faulty or correct elements. Therefore, the pairwise approaches are the most suitable for Learning-to-Rank fault localization.

Recently, Learning-to-Rank has been applied to improve the effectiveness of spectrum-based fault localization [B Le et al. 2016; Xuan and Monperrus 2014]. In this work, we further apply Learning-to-Rank to incorporate various mutation information to further improve fault localization. The basic idea of Learning-to-Rank for fault localization is to combine various different fault localization techniques (as different features) by learning a weight for each feature via machine learning. For each code element e, there can be n suspiciousness values computed by n studied fault

localization techniques, including both spectrum-based techniques and mutation-based techniques based on different mutation information (shown in Section 3.1 and Section 3.2). We assume that each suspiciousness value of e is $Susp_i(e)$ ($i = 1, 2, ...n$). According to certain Learning-to-Rank algorithm, the weight for each feature of e can be learned as $Weight_i(e)$ ($i = 1, 2, ...n$). Then new combined suspiciousness value of element e can be calculated as:

$$Susp_{Comb}(e) = \sum_{i \in 1,2,..n} Weight_i(e) * Susp_i(e) \quad (3)$$

Then the code elements can be ranked according to new suspiciousness values. Assume e^+ and e^- denote any pair of faulty and correct code elements, the loss function can be defined as the number of incorrectly ranked pairs:

$$Loss = \sum_{\langle e^+, e^- \rangle} \| Susp_{Comb}(e^+) \leq Susp_{Comb}(e^-) \| \quad (4)$$

In the learning phase, the training set consists of code elements from historical faulty programs/versions which include at least one failed test. Each element has several features that are the suspiciousness values calculated by various spectrum-based and muation-based fault localization techniques.

Table 5. Sample of traning data

	Label	PID	$Susp_1$	$Susp_2$	$Susp_3$	...
Element1	$y^{(1,1)}$	1	$x_1^{(1,1)}$	$x_2^{(1,1)}$	$x_3^{(1,1)}$	...
Element2	$y^{(1,2)}$	1	$x_1^{(1,2)}$	$x_2^{(1,2)}$	$x_3^{(1,2)}$	...
Element1	$y^{(2,1)}$	2	$x_1^{(2,1)}$	$x_2^{(2,1)}$	$x_3^{(2,1)}$	...
Element2	$y^{(2,2)}$	2	$x_1^{(2,2)}$	$x_2^{(2,2)}$	$x_3^{(2,2)}$	...
...	...	...	...	...	...	...

Table 5 presents a sample of training data. In the table, Column 1 presents different code elements. Column 2 ("Label") represents if the element is faulty or not. It corresponds to the ground-truth reference label in information retrieval. Column 3 ("PID") represents the ID of faulty program/version. In our study, PID represents Bug ID of Defects4J. It corresponds to the query in information retrieval. For example, $y^{(i,j)} = 1$ means the jth element in ith program/version is faulty. Other columns present element suspiciousness values computed by different fault localization techniques. For example, $x_k^{(i,j)}$ presents the suspiciousness value of the jth element in ith program/version based on the kth fault localization technique. In the learning phase, a ranking model including weights of different suspiciousness values can be built. Then the model is used to predict new suspiciousness values of elements in test data in the ranking phase. Various pairwise Learning-to-Rank approaches can be applied for fault localization, e.g., RankSVM [Lee and Lin 2014], RankBoost [Freund et al. 2003], RankNet [Burges et al. 2005], FRank [Tsai et al. 2007], and LambdaRank [Burges et al. 2006].

4 EXPERIMENTAL SETUP

In this paper, we investigate the following research questions:

- **RQ1:** How does mutation-based fault localization perform in localizing real bugs?
- **RQ2:** How do different failure message types impact mutaton-based fault localization techniques?
- **RQ3:** How does the assertion level information further impact mutaton-based fault localization techniques?
- **RQ4:** How does the Learning-to-Rank fault localization incorporating various mutation information perform in localizing real bugs?

In *RQ1*, we compare two mutation-based fault localization techniques (i.e., Metallaxis and MUSE) against 34 widely used traditional spectrum-based fault localization formulae on real bugs. In *RQ2*, we evaluate the effectiveness of Metallaxis and MUSE with 4 test failure message types. In *RQ3*, we

compare the effectiveness of Metallaxis and MUSE at both the test and assertion levels. In *RQ4*, we investigate the effectiveness of Learning-to-Rank fault localization with different suspiciousness values computed by different techniques, including Metallaxis and MUSE with different failure message types at both the test and assertion levels, as well as traditional spectrum-based techniques.

4.1 Implementation and Tool Supports

We use the PIT mutation testing framework[4] to apply mutation testing to the studied subjects since PIT is the most robust and widely used mutation testing tool for Java projects [Denaro et al. 2015; Lu et al. 2016; Možucha and Rossi 2016; Musco et al. 2016; Zhang et al. 2016]. We made three main modifications to PIT (Version 1.1.5) to implement Metallaxis and MUSE: (1) following August et al. [Shi et al. 2014], we force PIT to execute each mutant against the remaining tests even some tests have killed the mutant since the original PIT aborts test execution for a mutant once it is killed; (2) we enable PIT to apply mutation testing on programs with failed tests since the original PIT aborts mutation testing if any of the original tests fails; (3) we enable PIT to further capture detailed test outputs/messages for each mutant. In original PIT, the method `onTestFailure()` of class `ErrorListener` is only used to record test pass/fail information. Besides the test pass/fail information, we further modify PIT to capture the exception types, exception messages, and stack traces by invoking methods `getClass()`, `getMessage()`, and `getStackTrace()` on the captured `java.lang.Throwable` objects, respectively. To fully evaluate the potential of mutation-based fault localization, we use all the 16 mutation operators of PIT as shown in Table 6. The detailed explanation for each mutation operator can be found on PIT homepage.

We further capture the detailed assertion-level information for each mutant at the byte-code level using the ASM byte-code manipulation framework[5] to avoid modifying the test source code. Furthermore, to avoid changing the physical test byte-code on disk, we use Java Agent[6] to apply on-the-fly test byte-code transformation. Our on-the-fly instrumentation captures all the invocations to the original JUnit assertion APIs, and replaces them with the invocations to our own shadow version of assertion APIs which record the detailed checked values within each assertion and also catch thrown assertion exceptions (shown in Section 3.2). Note that for arrays checked by assertions, in order to capture the detailed changes for array elements, we concatenate all the array element contents. For the non-primitive objects checked by assertions, in order to detect detailed changes to object fields or transitive fields, we use the XStream library[7] to serialize the entire object graph into XML strings. For each assertion encountered during each mutant execution, capturing the detailed checked values can be extremely space-consuming and time-consuming due to the extensive file IO in case of large arrays or object graphs. Therefore, we apply the Apache Commons Codec library[8] to compute the CheckSum of the checked value for each assertion. In this way, all the checked values uniformly have only 40-character CheckSum hash.

Table 6. PIT mutation operators

ID	Mutation Operator
M1	Constructor Call Mutator
M2	Increments Mutator
M3	Inline Constant Mutator
M4	Invert Negs Mutator
M5	Math Mutator
M6	Negate Conditionals Mutator
M7	Non-Void Method Call Mutator
M8	Remove Conditional Mutator
M9	Return Vals Mutator
M10	Void Method Call Mutator
M11	Remove Increments Mutator
M12	Member Variable Mutator
M13	Switch Mutator
M14	Argument Propagation Mutator
M15	Conditional Boundary Mutator
M16	Remove Switch Mutator

[4] http://pitest.org/
[5] http://asm.ow2.org/
[6] https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html
[7] http://x-stream.github.io/
[8] https://commons.apache.org/proper/commons-codec/

Table 7. Studied spectrum-Based fault Localization formulae

Tech	Defn	Tech	Defn	Tech	Defn
Tarantula	$\frac{\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert}}{\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert}+\frac{\lvert T_p(e)\rvert}{\lvert T_p\rvert}}$	Ochiai	$\frac{\lvert T_f(e)\rvert}{\sqrt{(\lvert T_f(e)\rvert+\lvert T_p(e)\rvert)\lvert T_f\rvert}}$	Jaccard	$\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert+\lvert T_p(e)\rvert}$
Ample	$\lvert\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert}-\frac{\lvert T_p(e)\rvert}{\lvert T_p\rvert}\rvert$	RussellRao	$\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert+\lvert T_p\rvert}$	Hamann	$\frac{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert-\lvert T_p(e)\rvert-\lvert T_f(\bar{e})\rvert}{\lvert T_f\rvert+\lvert T_p\rvert}$
SørensenDice	$\frac{2\lvert T_f(e)\rvert}{2\lvert T_f(e)\rvert+\lvert T_p(e)\rvert+\lvert T_f(\bar{e})\rvert}$	Dice	$\frac{2\lvert T_f(e)\rvert}{\lvert T_f\rvert+\lvert T_p(e)\rvert}$	Kulczynski1	$\frac{\lvert T_f(e)\rvert}{\lvert T_f(\bar{e})\rvert+\lvert T_p(e)\rvert}$
Kulczynski2	$\frac{1}{2}(\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert}+\frac{\lvert T_f(e)\rvert}{\lvert T_f(e)\rvert+\lvert T_p(e)\rvert})$	SimpleMatching	$\frac{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert}{\lvert T_f\rvert+\lvert T_p\rvert}$	Sokal	$\frac{2\lvert T_f(e)\rvert+2\lvert T_p(\bar{e})\rvert}{2\lvert T_f(e)\rvert+2\lvert T_p(\bar{e})\rvert+\lvert T_f(\bar{e})\rvert+\lvert T_p(e)\rvert}$
M1	$\frac{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert}{\lvert T_f(\bar{e})\rvert+\lvert T_p(e)\rvert}$	M2	$\frac{\lvert T_f(e)\rvert}{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert+2\lvert T_f(\bar{e})\rvert+2\lvert T_p(e)\rvert}$	RogersTanimoto	$\frac{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert}{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert+2\lvert T_f(\bar{e})\rvert+2\lvert T_p(e)\rvert}$
Goodman	$\frac{2\lvert T_f(e)\rvert-\lvert T_f(\bar{e})\rvert-\lvert T_p(e)\rvert}{2\lvert T_f(e)\rvert+\lvert T_f(\bar{e})\rvert+\lvert T_p(e)\rvert}$	Hamming	$\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert$	Euclid	$\sqrt{\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert}$
Overlap	$\frac{\lvert T_f(e)\rvert}{min(\lvert T_f(e)\rvert,\lvert T_p(e)\rvert,\lvert T_f(\bar{e})\rvert)}$	Anderberg	$\frac{\lvert T_f(e)\rvert}{\lvert T_f(e)\rvert+2\lvert T_f(\bar{e})\rvert+2\lvert T_p(e)\rvert}$	Ochiai2	$\frac{\lvert T_f(e)\rvert\lvert T_p(\bar{e})\rvert}{\sqrt{(\lvert T_f(e)\rvert+\lvert T_p(e)\rvert)(\lvert T_f(\bar{e})\rvert+\lvert T_p(\bar{e})\rvert)(\lvert T_f(e)\rvert+\lvert T_p(\bar{e})\rvert)(\lvert T_f(\bar{e})\rvert+\lvert T_p(e)\rvert)}}$
Zoltar	$\frac{\lvert T_f(e)\rvert}{\lvert T_f\rvert+\lvert T_p(e)\rvert+\frac{10000\lvert T_f(\bar{e})\rvert\lvert T_p(e)\rvert}{\lvert T_f(e)\rvert}}$	Wong1	$\lvert T_f(e)\rvert$	Wong2	$\lvert T_f(e)\rvert-\lvert T_p(e)\rvert$
ER5c	$\begin{cases}0 & \text{if } \lvert T_f(e)\rvert<\lvert T_f\rvert \\ 1 & \text{if } \lvert T_f(e)\rvert=\lvert T_f\rvert\end{cases}$	GP02	$2(\lvert T_f(e)\rvert+\sqrt{\lvert T_p\rvert})+\sqrt{\lvert T_p(e)\rvert}$	GP03	$\sqrt{\lvert\lvert T_f(e)\rvert^2-\sqrt{\lvert T_p(e)\rvert}\rvert}$
GP13	$\lvert T_f(e)\rvert(1+\frac{1}{2\lvert T_p(e)\rvert+\lvert T_f(e)\rvert})$	GP19	$\lvert T_f(e)\rvert\sqrt{\lvert\lvert T_p(e)\rvert-\lvert T_f(e)\rvert+\lvert T_f\rvert-\lvert T_p\rvert\rvert}$	SBI	$\frac{\lvert T_f(e)\rvert}{\lvert T_f(e)\rvert+\lvert T_p(e)\rvert}$
DStar2	$\frac{\lvert T_f(e)\rvert^2}{\lvert T_p(e)\rvert+\lvert T_f(\bar{e})\rvert}$	Wong3	$\lvert T_f(e)\rvert-h, where\ h=\begin{cases}\lvert T_p(e)\rvert & \text{if } \lvert T_p(e)\rvert\le 2 \\ 2+0.1(\lvert T_p(e)\rvert-2) & \text{if } 2<\lvert T_p(e)\rvert\le 10 \\ 2.8+0.01(\lvert T_p(e)\rvert-10) & \text{if } \lvert T_p(e)\rvert>10\end{cases}$		
ER1a	$\begin{cases}-1 & \text{if } \lvert T_f(e)\rvert<\lvert T_f\rvert \\ \lvert T_p\rvert-\lvert T_p(e)\rvert & \text{if } \lvert T_f(e)\rvert=\lvert T_f\rvert\end{cases}$	ER1b	$\lvert T_f(e)\rvert-\frac{\lvert T_p(e)\rvert}{\lvert T_p\rvert+1}$		

Table 8. Subject statistics

ID	Program	#Faults	#Susp. Methods	#Tests	LoC
Lang	Commons Lang	65	674	2,245	22K
Time	Joda-Time	27	3,307	4,130	28K
Math	Commons Math	106	4,594	3,602	85K
Chart	JFreeChart	26	2,708	2,205	96K
Closure	Closure Compiler	133	104,007	7,927	90K
	Total	357	115,290	20,109	321K

For the Learning-to-Rank technique, we use both LIBSVM[9] (our default library), a widely-used library for support vector machines, and XGBoost[10], an widely-used optimized distributed gradient boosting library, to investigate their effectiveness. We use RankSVM with linear kernel (version 1.95) from LIBSVM with default settings, and LambdaRank from XGBoost with the `gbtree` booster and a popular setting: `max_depth = 60`, `num_round = 100`, `colsample_bytree=0.85`, and `eta =0.5` (we use the default values for all the other parameters). For each technique, we perform leave-one-out cross validation [B Le et al. 2016] not only across each of the five projects, but also across whole five projects. For the total n bugs, we separate them into two groups: one bug as the test data to predict its rank and other $n-1$ bugs as the training data to build the ranking model.

We implement 34 most widely used spectrum-based fault localization formulae in Java, e.g., Tarantula [Jones and Harrold 2005], SBI [Liblit et al. 2005], Jaccard [Abreu et al. 2007], Ochiai [Abreu et al. 2006], Ochiai2 [Naish et al. 2011], and Kulczynski2 [Naish et al. 2011], including all state-of-the-art formulae. They are shown in Table 7 and the notations $T_f(e)$,$T_p(e)$,$T_f(\bar{e})$, $T_p(\bar{e})$,T_f and T_p are shown in Section 2.1. These techniques all rely on coverage information of both passed/failed tests. We perform on-the-fly bytecode instrumentation using ASM and Java Agent to collect the required coverage information. Note that the studied spectrum-based formulae are used as baseline techniques as well as in implementing Metallaxis.

All our experiments were conducted on a Dell machine with Intel(R) Xeon(R) CPU E5-2697 v4@2.30GHz (18C) and 94GB RAM, running Ubuntu 14.04.5 LTS and Oracle Java 64-Bit Server version 1.8.0_77.

4.2 Subjects, Tests and Faults

In this work, we evaluate the studied fault localization techniques on real faults from the Defects4J benchmark[11]. We use all the five available projects from Defects4J in August 2016, which include total 357 real faults detected during their software development cycle. The detailed subject information used in this work are shown in Table 8. In the table, Column 1 presents the subject IDs that will be used in the remaining text. Column 2 presents the full names for the subjects. Column 3 presents the number of studied faulty versions for each subject. Column 4 presents the total number of suspicous methods (i.e., the methods executed by failed tests) of all faulty versions for each subject. Columns 5 and 6 present the LoC (i.e., Lines of Code) and test number information for the first version (i.e., the most recent and usually the largest version) of each subject in Defects4J. Note that in the paper, each unique bug ID is represented by the subject ID and the buggy version number, e.g., Lang-1 indicate the first buggy version of subject Lang.

4.3 Dependent Variables

Previous studies have demonstrated that statement-level fault localization may be too fine-grained and miss useful context information [Parnin and Orso 2011], while class-level fault localization is too coarse-grained and cannot help understand and fix the bug within a class [Kochhar et al. 2016; Wang et al. 2015]. Therefore, following recent work on fault localization [B Le et al. 2016; Dao et al. 2017; Le et al. 2015; Zhang et al. 2017], we also focus on method-level fault localization, i.e., localizing the faulty methods among all the source code methods. We use the following widely used dependent variables to measure the effectiveness of the studied fault localization techniques:
DV1: Recall at Top-N: This dependent variable measures the number of faults with at least one faulty element within Top-N in the ranked list. The hypothesis for this dependent variable is that once the first faulty element is found, it may become much easier to find the remaining faulty elements. This metric emphasizes earlier fault detection and has been widely used in fault localization work [Le et al. 2015; Saha et al. 2013; Zhou et al. 2012]. Note that a recent study reported that developers usually only inspect top-ranked program elements during fault localization, e.g., 73.58% developers only check Top-5 localized elements [Kochhar et al. 2016]. Therefore, following prior work, we use Top-N (N=1, 3, 5) in our experimental study.
DV2: Mean Average Rank (MAR): Following existing work [Moon et al. 2014; Zhang et al. 2013], we use the rank of the faulty methods to directly measure the developers' effort in identifying the actual faulty methods using the fault localization techniques. For the faults with multiple faulty elements, we compute the average ranking of the faulty elements. Then, for each project, MAR is simply the mean of the average rank for all its faults. This metric emphasizes precise localization for all faulty elements.
DV3: Mean First Rank (MFR): In practice, for faults with multiple faulty elements, the identification of the first faulty element can be crucial since the rest faulty elements may be directly localized after that. Therefore, for each project, we use MFR to compute the mean of the first relevant faulty element's rank for each fault. This metric emphasizes fast localization of the first relevant faulty element to ease debugging.

5 RESULT ANALYSIS

In this section, we first present the detailed study results for evaluating the effectiveness of the original Metallaxis and MUSE on real bugs (Section 5.1). Then, we present the results of our

[9] https://www.csie.ntu.edu.tw/~cjlin/libsvm/
[10] https://github.com/dmlc/xgboost
[11] https://github.com/rjust/defects4j

Table 9. Effectiveness of existing mutation-based fault localization techniques

Subjects	Techniques	Top-1	Top-3	Top-5	MFR	MAR	Subjects	Techniques	Top-1	Top-3	Top-5	MFR	MAR
	Ochiai	6	14	15	9.00	9.51		Ochiai	6	11	13	15.96	18.87
	Kulczynski2	6	15	16	9.48	10.88		Kulczynski2	7	13	14	21.23	24.95
	Zoltar	6	15	16	8.80	19.36		Zoltar	7	13	14	21.27	23.80
	Ochiai2	6	14	15	8.92	9.42		Ochiai2	6	11	13	16.46	18.99
Chart	M2	5	13	14	34.68	35.09	Time	M2	8	13	14	22.54	25.60
	Me-Ochiai	7	15	17	12.68	13.31		Me-Ochiai	7	12	15	12.35	14.82
	Me-Kulczynski2	7	15	17	13.04	13.66		Me-Kulczynski2	6	11	15	16.58	19.35
	Me-Zoltar	7	15	17	13.04	13.66		Me-Zoltar	6	11	14	17.15	19.36
	Me-Ochiai2	7	15	17	12.64	13.27		Me-Ochiai2	7	12	15	12.35	14.74
	Me-M2	7	15	16	21.76	22.34		Me-M2	7	13	16	17.69	20.16
	Muse	4	12	12	23.04	23.84		Muse	5	6	7	87.81	89.96
	Ochiai	23	52	62	9.73	11.72		Ochiai	24	44	50	4.63	5.01
	Kulczynski2	21	50	59	9.85	12.07		Kulczynski2	25	44	50	4.69	5.19
	Zoltar	21	49	58	10.13	12.11		Zoltar	25	44	50	4.52	5.01
	Ochiai2	23	52	62	9.72	11.70		Ochiai2	24	44	50	4.60	4.98
Math	M2	22	49	56	10.14	12.46	Lang	M2	25	44	50	4.82	5.19
	Me-Ochiai	20	50	70	7.84	9.41		Me-Ochiai	32	51	56	2.84	3.15
	Me-Kulczynski2	19	50	70	7.71	9.18		Me-Kulczynski2	32	50	56	2.85	3.17
	Me-Zoltar	19	50	70	7.71	9.11		Me-Zoltar	32	50	56	2.74	3.05
	Me-Ochiai2	20	50	70	7.84	9.41		Me-Ochiai2	32	51	56	2.79	3.10
	Me-M2	20	47	68	8.30	10.12		Me-M2	32	51	57	2.87	3.30
	Muse	17	45	55	27.54	28.44		Muse	22	38	46	6.00	6.87
	Ochiai	14	30	38	90.28	102.28		Ochiai	73	151	178	39.56	44.99
	Kulczynski2	17	31	41	88.51	103.11		Kulczynski2	76	153	180	39.36	45.99
	Zoltar	17	31	41	87.65	102.06		Zoltar	76	152	179	39.05	46.1
	Ochiai2	14	30	38	94.57	106.14		Ochiai2	73	151	178	41.2	46.43
Closure	M2	17	32	40	86.19	103.14	**Overall**	M2	77	151	174	40.51	47.91
	Me-Ochiai	20	52	69	21.72	26.14		Me-Ochiai	86	180	227	12.86	15.28
	Me-Kulczynski2	20	52	70	25.69	32.25		Me-Kulczynski2	84	178	228	14.66	17.88
	Me-Zoltar	20	52	71	27.24	33.43		Me-Zoltar	84	178	228	15.27	18.28
	Me-Ochiai2	20	52	69	21.72	26.14		Me-Ochiai2	86	180	227	12.84	15.26
	Me-M2	24	55	70	25.46	36.62		Me-M2	90	181	227	15.46	20.51
	Muse	26	53	63	215.89	237.86		Muse	74	154	183	98.78	107.69

extensions to traditional mutation-based fault localization by considering different test failure message types as well as the assertion-level information (Section 5.2 and Section 5.3). Finally, we present the experimental results for our TraPT Learning-to-Rank technique combining various dimensions of mutation information (Section 5.4).

5.1 RQ1: Mutation-based Fault Localization on Real Bugs

Quantitative Analysis. Table 9 presents the main fault localization results of Metallaxis (with Type4 test message) and MUSE (with Type1 test message) using all the mutants generated by PIT. Note that we also select 5 most effective spectrum-based formulae as the baseline from all the 34 studied formulae to implement the Metallaxis technique. In the table, Columns "Subjects" and "Techniques" present the corresponding subjects and studied techniques, while the other columns present the studied metrics, including Top-N (N=1, 3, 5), MFR, and MAR. Note that, since Metallaxis utilizes different spectrum-based formulae to calculate the suspiciousness values, we use Me-"Spectrum" to represent Metallaxis utilizing the corresponding spectrum-based formulae in the table. From the table, we have two observations. First, Metallaxis is able to outperform spectrum-based fault localization on most subjects in terms of the most studied metrics. For example, in total, Ochiai (one of the most effective spectrum-based fault localization techniques) can only localize 73 faulty methods within Top-1, while Metallaxis with Ochiai is able to localize 86 faulty methods within Top-1. Furthermore, MAR of Metallaxis with Ochiai is 15.28, 66.04% more precise than that Ochiai (44.99)! To our knowledge, this is the first work demonstrating the effectiveness of mutation-based fault localization on a large number of real bugs. Second, MUSE tends to be less effective than even the traditional spectrum-based fault localization on all the studied subjects due

```
public static synchronized GJChronology getInstance(DateTimeZone zone,
                                    ReadableInstant gregorianCutover, int minDaysInFirstWeek) {
     zone = DateTimeUtils.getZone(zone);
     Instant cutoverInstant;
     if (gregorianCutover == null) {
       cutoverInstant = DEFAULT_CUTOVER;
     } else {
       cutoverInstant = gregorianCutover.toInstant();
   +++ LocalDate cutoverDate = new LocalDate(cutoverInstant.getMillis(),
                                  GregorianChronology.getInstance(zone));
   +++ if (cutoverDate.getYear() <= 0) {
   +++     throw new IllegalArgumentException("Cutover too early. Must be on or after 0001-01-01.");
   +++ }
     }...
}
```

Fig. 4. Buggy method of Time-6

to the insensitive failure message type used (i.e., the pass/fail information). Finally, the effectiveness of mutation-based fault localization techniques may not be stable, e.g., Metallaxis is not uniformly more effective than spectrum-based fault localization, while MUSE is also not uniformly less effective. For Chart, although Metallaxis is better than Ochiai in terms of Top-N metrics, it is inferior in terms of MFR and MAR, indicating that Metallaxis may perform extremely poorly in some cases; for Closure, although MUSE performs poorly in MFR/MAR, it is able to localize the most number (i.e., 26) of bugs within Top-1.

> **Finding 1:** Overall, Metallaxis can greatly outperform corresponding spectrum-based fault localization techniques (e.g., 66.04% more precise in MAR when using the Ochiai formula), while MUSE tends to be even less effective than spectrum-based fault localization. Also, Metallaxis and MUSE are both unstable, e.g., MUSE may even outperform Metallaxis on some bugs.

Qualitative Analysis. The above quantitative analysis shows that even though the overall result of Metallaxis is promising, mutation-based fault localization is not better than spectrum-based fault localization all the time. We further qualitatively investigate why mutation-based is not better than spectrum-based fault localization and have found the following potential reasons:

Absent Mutants. In some cases, buggy methods may not generate any mutant so that their suspiciousness values cannot be calculated at all. For example, shown in Figure 3, for bug Time-22, the buggy method `Period(long duration)` in class `org.joda.time.Period` has only one statement, `super(duration, null, null)`, which simply invokes the initializer method of its super class; the corresponding bug fix changes the statement to invoke another super class initializer `super(duration)`. For the PIT mutation testing tool that we used, there is no mutation operator to apply for this buggy method. Although PIT has a mutation operator to remove method invocations, the resulting mutant is not valid for execution since the super class initializer method must be invoked in the buggy initializer method according to the Java specification. For such cases, more mutation operators (e.g., replacing method API invocations) need to be designed for mutation-based fault localization. For this work, in order to rank the methods without mutants, we set their suspiciousness value as 0.

```
public Period(long duration) {
  --- super(duration, null, null);
  +++ super(duration);
}
```

Fig. 3. Buggy method of Time-22

Stubborn Tests. Metallaxis and MUSE calculate suspiciousness values based on if there are mutants impacting test executions. In some cases, however, the tests can be quite *stubborn*, making the

```
public void test_cutoverPreZero() {
  ...
  try {
      GJChronology.getInstance(
        DateTimeZone.UTC, cutover);
      fail();
  } catch (IllegalArgumentException ex) {
      // expected
  }
}
```

Fig. 5. Failed test of Time-6

```
junit.framework.AssertionFailedError
    at junit.framework.Assert.fail(Assert.java:55)
    at junit.framework.Assert.fail(Assert.java:64)
    at junit.framework.TestCase.fail(TestCase.java:235)
    at org.joda.time.chrono.TestGJDate.test_cutoverPre(...)
```

Fig. 6. Failure message of Time-6

```
public PiePlotState initialise(Graphics2D g2,
                Rectangle2D plotArea,PiePlot plot,
                Integer index, PlotRenderingInfo info) {
     PiePlotState state = new PiePlotState(info);
     state.setPassesRequired(2);
+++ if (this.dataset != null) {
         state.setTotal(DatasetUtilities.
             calculatePieDatasetTotal(plot.getDataset()));
+++ }
     state.setLatestAngle(plot.getStartAngle());
     return state;
}
```

Fig. 7. Buggy method of Chart-15

```
public void testDrawWithNullDataset() {
        boolean success = false;
        try {...
            success = true;
        }
        catch (Exception e) {
            success = false;
        }
        assertTrue(success);
}
```

Fig. 8. Failed test of Chart-15

```
protected double doSolve() {
    // prepare arrays with the first points
    final double[] x = new double[maximalOrder + 1];
    final double[] y = new double[maximalOrder + 1];
    ...
    // target for the next evaluation point
    double targetY;
    if (agingA >= MAXIMAL_AGING) {
   ---  targetY = -REDUCTION_FACTOR * yB;
   +++  final int p = agingA - MAXIMAL_AGING;
   +++  final double weightA = (1 << p) - 1;
   +++  final double weightB = p + 1;
   +++  targetY = (weightA * yA - weightB *
           REDUCTION_FACTOR*yB)/(weightA+weightB);}
    }...
}
```

Fig. 9. Buggy method of Math-40

```
public void testIssue716() {
  BracketingNthOrderBrentSolver solver =
          new BracketingNthOrderBrentSolver
               (1.0e-12, 1.0e-10, 1.0e-22, 5);
  ...
  double result = solver.solve(100, sharpTurn,
    -0.9999999,30,15,AllowedSolution.RIGHT_SIDE);
  Assert.assertEquals(0, sharpTurn.value(result),
         solver.getFunctionValueAccuracy());
  Assert.assertTrue(sharpTurn.value(result)>=0);
  Assert.assertEquals(-0.5, result, 1.0e-10);
```

Fig. 10. Failed test of Math-40

generated mutants unable to change test executions. For example, shown in Figure 4, bug Time-6 has a buggy method `getInstance(...)`, which was fixed via adding some additional code logics. Note that the buggy method has a number of mutants due to the large method body. However, shown in Figure 5, the corresponding failed test `TestGJDate.test_cutoverPreZero()` only has one special JUnit assertion `fail()`, which has no parameters and is used to make sure an exception is thrown when executed. Therefore, the test always produces the same failure message (shown in Figure 6), making it hard for the mutants to change its failure message no matter what changes the mutants make. Another example is from bug Chart-15 (shown in Figure 7), whose fixed version adds a conditional statement to perform `null` checks before a method invocation. As illustrated in Figure 8, the failed test for this bug has only one assertion, `assertTrue(success)`, and checks that no exception is thrown. The assertion carries only boolean variable as its parameter so that output message also cannot be changed unless the exception can be directly muted by

mutants (which can be hard). In these cases, the mutation-based techniques are not able to change the output messages of failed tests, making the buggy methods ranked similarly with other correct methods. This reason actually motivates us to investigate multiple assertions inside tests to capture more information.

Insensitive/Sensitive Failure Messages. MUSE considers pass/fail information of tests and Metallaxis considers detailed changes in failure messages. The fact that the results of Metallaxis are much better than those of MUSE shows that the detailed failure messages can be more informative than the insensitive pass/fail information, indicating that insensitive information can make mutation-based fault localization perform poorly for some cases. Meanwhile, the detailed failure messages may also be too sensitive sometimes because they include stack traces of failed tests which may even change for different runs of the same failed tests (e.g., due to the non-determinism in real code, such as randomness or concurrency). For example, as shown in Figure 9, bug Math-40 was fixed via adding more computing logics for variable `targetY`. Its corresponding failed test is shown in Figure 10. When localizing this bug, Metallaxis with Ochiai ranks this buggy method as 7th, while MUSE with more insensitive test information can rank it as 1st. The reason is that the detailed test failure message of the failed test can be easily changed by many mutants of bug-free methods, making Metallaxis rank certain bug-free methods even higher than the actualy buggy method. To illustrate, various mutants can cause the originally failed tests to throw different types of exceptions (e.g., `TooManyEvaluationsException` and `NoBracketingException`) or exceptions with different messages before the JUnit assertion invocations. In contrast, only mutants of the actually buggy method can make the originally failed test pass, making MUSE rank the buggy method as the highest. This finding actually motivates us to empirically study the impacts of different types of failure messages on mutation-based fault localization.

Finding 2: Absent mutants, stubborn tests, and insensitive/sensitive failure messages are the main reasons leading to unstable or poor mutation-based fault localization.

Mutation Overhead. Mutation testing has been widely recognized as one of the most expensive testing methodologies due to the execution of a large number of mutants on the tests [Jia and Harman 2011]. Therefore, we also investigate the mutation testing cost for fault localization. Table 10 shows the mutation testing cost by PIT for the first version (i.e., the latest and usually largest version) of each subject. In the table, Row 2 represents the number of threads used for each subject since PIT supports thread configuration for parallel mutation testing. Row 3 presents the mutation testing time, while Row 4 presents the number of all generated mutants using all mutation operators of PIT. From the table, although mutation testing only takes around 1 hour for the three smaller subjects, it can cost over 8 hours for larger subjects. Although such mutation cost can be easily alleviated by increasing thread number, using more powerful machines/clusters, or running overnight, it is still interesting to explore safe ways to reduce mutation cost. In the literature, researchers have proposed to use *selective mutation testing* [Offutt et al. 1993; Zhang et al. 2013] to randomly select a subset of mutants for fault localization [Papadakis and Le Traon 2014, 2015]. However, as shown in prior work and also demonstrated in our above qualitative study, unselecting an important mutant that can change the outcomes of the originally failed tests can greatly impact the fault localization effectiveness.

Table 10. PIT mutation testing results

Subject	Chart-1	Time-1	Lang-1	Math-1	Closure-1
Threads	2	2	2	2	9
Time Cost	1h 16m	1h 19m	1h 8m	8h 35m	18h 23m
All Mutants	58,419	30,368	29,012	98,871	98,932
Suspicious Mutants	682	268	70	233	28,250

Table 11. Muse: all mutants vs. suspicious mutants

Subjects		Top-1	Top-3	Top-5	MFR	MAR
Chart	All Mutants	4	12	12	23.04	23.84
	Suspicious Mutants	4	12	12	23.00	23.80
Time	All Mutants	5	6	7	87.81	89.96
	Suspicious Mutants	5	6	7	87.81	89.96
Math	All Mutants	17	45	55	27.54	28.44
	Suspicious Mutants	17	45	55	27.53	28.43
Lang	All Mutants	22	38	46	6.00	6.87
	Suspicious Mutants	22	38	46	5.94	6.81
Closure	All Mutants	26	53	63	215.89	237.86
	Suspicious Mutants	26	53	63	215.89	237.86
Overall	All Mutants	74	154	183	98.78	107.69
	Suspicious Mutants	74	154	183	98.77	107.68

Fortunately, not all the mutants are important for fault localization; instead, only the mutants that can potentially change the failed test outcomes are important for fault localization. Actually, only the mutants occurring on the statements executed by failed tests (denoted as *suspicious mutants* in this paper) can potentially change the failed test outcomes, since the mutated statements of other mutants cannot even be executed by the failed tests. Prior work on fault localization has realized that Metallaxis has exactly the same results using either all mutants or only the suspicious mutants on many formulae (including all the 5 most effective spectrum-based formulae shown in Table 9) [Pearson et al. 2017]. The reason is that the Metallaxis formulae (e.g., Equation 1) only considers the mutant with highest suspiciousness value for each ranked suspicious method, while the non-suspicious mutants have suspiciousness value 0 and cannot impact the results for many spectrum-based formulae. On the contrary, the MUSE formula (Equation 2) involves a weight parameter α which is computed based on the total number of changed failed/passed tests for all mutants, and also can have negative suspiciousness values. Therefore, it is not clear how MUSE performs with only the suspicious mutants. To check the impact of using simply the suspicious mutants for MUSE, we modify the MUSE α computation by simply using the total number of changed failed/passed tests for the suspicious mutants, and also compute the method suspiciousness based on the suspiciousness values of only the suspicious mutants occurring in each method. Table 11 shows that the results of MUSE using all mutants and only the suspicious mutants are almost exactly same. For example, the overall MAR of using all mutants and suspicious mutants is 107.69 and 107.68 respectively. To our knowledge, this is the first study demonstrating that MUSE can also use only the suspicious mutants without losing much accuracy. Therefore, in our following experiments, we use only the suspicious mutants for both MUSE and Metallaxis. Shown in the last row of Table 10, this optimization can reduce the mutation cost significantly (e.g., reduce the number of executed mutants by 71.4% to 99.8%) with almost no accuracy lost.

> **Finding 3:** Mutation testing can cost hours for large-scale systems. However, using only the *suspicious mutants* can largely reduce the mutation testing cost with almost no accuracy lost for both Metallaxis and MUSE.

5.2 RQ2:Impacts of Different Failure Message Types

Tables 12 and 13 present the fault localization results for the two studied mutation-based fault localization techniques with 4 different types of failure messages. Note that due to the space limit, we select the best spectrum-based formulae Ochiai to implement Metallaxis (the other formulae show similar pattern). In the tables, Column 1 lists the studied subjects, Columns 2-5 present the MFR/MAR values for Metallaxis with different types of failure messages, and Columns 6-9 present the MFR/MAR values for MUSE with different failure message types. For example, “Me-Type1” and “Muse-Type1” represent that Metallaxis and MUSE with the first failure message type. For each technique on each subject, we show their average MAR/MFR values using four different failure message types, as well the BestRank value for each failure message type (i.e., the number

Table 12. MFR of different failure types at the test level

	Metallaxis								MUSE							
	Me-Type1		Me-Type2		Me-Type3		Me-Type4		Muse-Type1		Muse-Type2		Muse-Type3		Muse-Type4	
Subjects	BestRank	MFR	BestRank	MFR	BestRank	MFR	BestRank	MFR	BestRank	MFR	BestRank	MFR	BestRank	MFR	BestRank	MFR
Chart	14	22.84	18	12.92	20	12	18	12.68	14	23	17	20.2	19	18.2	18	19
Time	9	87.62	20	12.77	13	12.31	13	12.35	9	87.81	20	15.38	15	15.81	14	15.5
Math	51	27.61	66	7.06	60	8.16	57	7.84	49	27.53	62	7.38	56	8.51	54	8.14
Lang	29	5.82	48	3.11	47	2.74	44	2.84	28	5.94	47	3.16	46	2.81	45	2.87
Closure	81	215.02	76	48.46	49	32.82	48	21.72	72	215.89	82	51.73	52	34.87	53	20.98
Overall	184	98.42	228	22.79	189	17.06	180	12.86	172	98.77	228	24.84	188	18.66	184	13.36

Table 13. MAR of different failure types at the test level

	Metallaxis								MUSE							
	Me-Type1		Me-Type2		Me-Type3		Me-Type4		Muse-Type1		Muse-Type2		Muse-Type3		Muse-Type4	
Subjects	BestRank	MAR	BestRank	MAR	BestRank	MAR	BestRank	MAR	BestRank	MAR	BestRank	MAR	BestRank	MAR	BestRank	MAR
Chart	12	23.64	18	13.67	20	12.76	18	13.31	12	23.8	16	20.8	18	18.87	19	19.6
Time	8	89.87	21	14.58	13	14.76	12	14.82	8	89.96	19	16.91	15	17.98	15	17.61
Math	47	28.51	66	8.96	62	9.76	59	9.41	44	28.43	62	9.21	59	10.03	56	9.67
Lang	26	6.69	48	3.44	46	3.06	43	3.15	25	6.81	48	3.57	46	3.2	44	3.27
Closure	75	237.04	77	57.82	44	38.13	46	26.14	66	237.86	82	66	47	45.2	54	30.62
Overall	168	107.36	230	27.13	185	19.83	178	15.28	155	107.68	227	30.99	185	23.28	188	17.72

of bugs on which the type is the best among all four types[12]). For example, the BestRank value of "Me-Type1" for project Chart is 12 in terms of MAR. It means that there are 12 Chart buggy versions where MAR of "Me-Type1" is the lowest compared with other failure message types of Metallaxis (i.e., "Me-Type2", "Me-Type3" and "Me-Type4 "). To further illustrate the detailed effectiveness of different types, we also take Metallaxis as example and show the detailed rank of the first localized bug for each studied version in Figure 11.

The experimental data shows that for both Metallaxis and MUSE, the first failure message type is much worse than other types on average (e.g., much higher MAR/MFR values and much lower BestRank values). This is because the first failure message type simply traces pass/fail outcome, while a failed test usually cannot be easily changed into passing by mutation testing. When a failed test still fails after mutation, its exception type, exception message, or stack traces may actually change. This is why we consider more detailed failure message types.

The experimental results also show that the more detailed Type4 test information does not necessarily always perform better. Actually, both Type2 and Type3 information may outperform Type4 for the two studied mutation-based fault localization techniques. For example, for Metallaxis, Type2 and Type3 have 230 and 185 BestRank values in MAR, respectively, while Type4 's BestRank is only 178. The results are also similar for MUSE and MFR. The reason is that detailed failure messages may be too sensitive. Actually, sometimes different runs of the same tests on the same program version may produce different detailed failure messages. For example, for bug Math-79, the rank of a buggy method by Metallaxis is 9th for Type4 , but 6th for both Type2 and Type3.

To investigate whether the 4 failure types have statistically different impacts on fault localization, we further applied the One-way Repeated Measures ANOVA analysis (also denoted as rANOVA) [Girden 1992; Huck and McLean 1975; von Ende 2001] on the fault localization results using the 4 different failure types to investigate their statistical differences. Note that we applied rANOVA instead of the standard ANOVA because we were investigating the impact of the factor (i.e., the failure message type in this work) on the same set of subjects (i.e., bugs in this work). Table 14 presents the rANOVA analysis results on MFR and MAR values of Metallaxis and MUSE across all the studied bugs from five subject systems. All the statistical tests were computed using the R

Table 14. rANOVA analysis for failure message types

		Df	Sum Sq	Mean Sq	F-value	Pr(>F)
Metallaxis-MFR	FailureMsgType	3	1723119	574373	33.22	<2e-16***
	Residuals	1041	17998584	17290		
Metallaxis-MAR	FailureMsgType	3	1982700	660900	37.51	<2e-16***
	Residuals	1041	18340352	17618		
Muse-MFR	FailureMsgType	3	1682884	560961	33.45	<2e-16***
	Residuals	1041	17455831	16768		
Muse-MAR	FailureMsgType	3	1855371	618457	36.59	<2e-16***
	Residuals	1041	17596261	16903		

Significance codes p<0.001 (***) p<0.01 (**) p<0.05 (*)

[12] Note that we increase the BestRank values for all the tied best types.

Table 15. Test/assert level mutation for Me-Type4

Subjects	Test		Assert		Test		Assert	
	BestRank	MAR	BestRank	MAR	BestRank	MFR	BestRank	MFR
Chart	18	13.31	16	19.81	17	12.68	19	19.04
Time	17	14.82	16	25.14	16	12.35	18	19.23
Math	63	9.41	75	15.82	61	7.84	80	14.74
Lang	45	3.15	52	3.83	45	2.84	54	2.98
Overall	143	8.72	159	13.97	139	7.51	171	12.41

Table 16. Test/assert level mutation for Muse-Type4

Subjects	Test		Assert		Test		Assert	
	BestRank	MAR	BestRank	MAR	BestRank	MFR	BestRank	MFR
Chart	19	19.6	15	21.13	18	19	18	20.44
Time	17	17.61	18	24.97	15	15.5	20	19.31
Math	65	9.67	71	16.04	62	8.14	77	14.94
Lang	46	3.27	52	3.93	46	2.87	54	3.08
Overall	147	9.94	156	14.24	141	8.77	169	12.71

language[13]. From the table, we observe that the rANOVA p-value using different failure message types is <2e-06 in terms of both MAR and MFR metrics for both Metallaxis and MUSE, rejecting the null hypothesis at the level of 0.05 (p-value«0.05). The analysis results demonstrate that the 4 different failure message types have significantly different effectiveness for fault localization and can potentially be combined together for better fault localization performance.

Finally, the experimental results show that despite the different effectiveness of different failure message types, each failure message type has its own superiority. Although MAR of Type1 is worse than other types, the BestRank results show that there are certain buggy versions for which Type1 outperforms other types of failure messages. For example, for the bug Chart-6, "Me-Type1" can rank the buggy method as 2nd, outperforming other 3 types that rank the buggy method as 4th, 16th, and 16th, respectively. When we consider Type1 of this special version, we observe that very few mutants (including the mutants of the buggy method) impact the pass/fail information of tests so that it is easy to differentiate the buggy method from other methods. However, when considering other types, we found that mutants of more methods make the failure messages change; thus, the buggy method and other methods share the similar suspiciousness values and cannot be easily distinguished. In this case, the rank of buggy method for Type1 is better than other types. Actually, each type has non-trivial BestRank values in terms of MAR/MFR for both studied techniques.

Finding 4: Four different failure message types have significant impacts on the fault localization results for both MUSE and Metallaxis. In addition, each failure message type has its own superiority, further motivating our Learning-to-Rank solution to combine mutation information of all four different types.

5.3 RQ3:Impacts of Detailed Assertion Information

In this RQ, we further investigate the impacts of the detailed assertion-level mutation information. As shown in Section 3.2, the default assertion also has four different failure message types for the assertion-level information. Since the results for different failure message types follow a similar pattern, in this RQ, we show the results using the Type4 failure messages as the representative. Tables 15 and 16 present the results for Me-Type4 with Ochiai and MUSE, respectively. Note that since Closure is a JavaScript compiler and does not have standard JUnit tests (e.g., often do not directly contain assertions or contain self-defined assertions), our current implementation is not able to trace assertion-level information for Closure. From the overall results we can observe that MAR and MFR of the test level are better than those of the assertion level for both Metallaxis and MUSE. For example, MAR is 8.72 at the test level while 13.97 at the assertion level for Metallaxis. However, the assertion-level information is not always inferior to the test-level information. For example, the BestRank values at the test level are lower than those at the assertion level in terms of both MAR and MFR. In total, for MAR, the assertion level performs the best on 159 buggy versions while the test level performs the best on only 143 buggy versions for Metallaxis. To illustrate, for bug Chart-20, "Me-Type4 " with "Ochiai" at the test level can rank the buggy method as 1st while the assertion level can only rank it as 8th. The failed test of this buggy version is `test1808376()` which includes 6 assertions in it. We assume that each assertion is denoted as `a-n`, where `n` is

[13] https://www.r-project.org/about.html

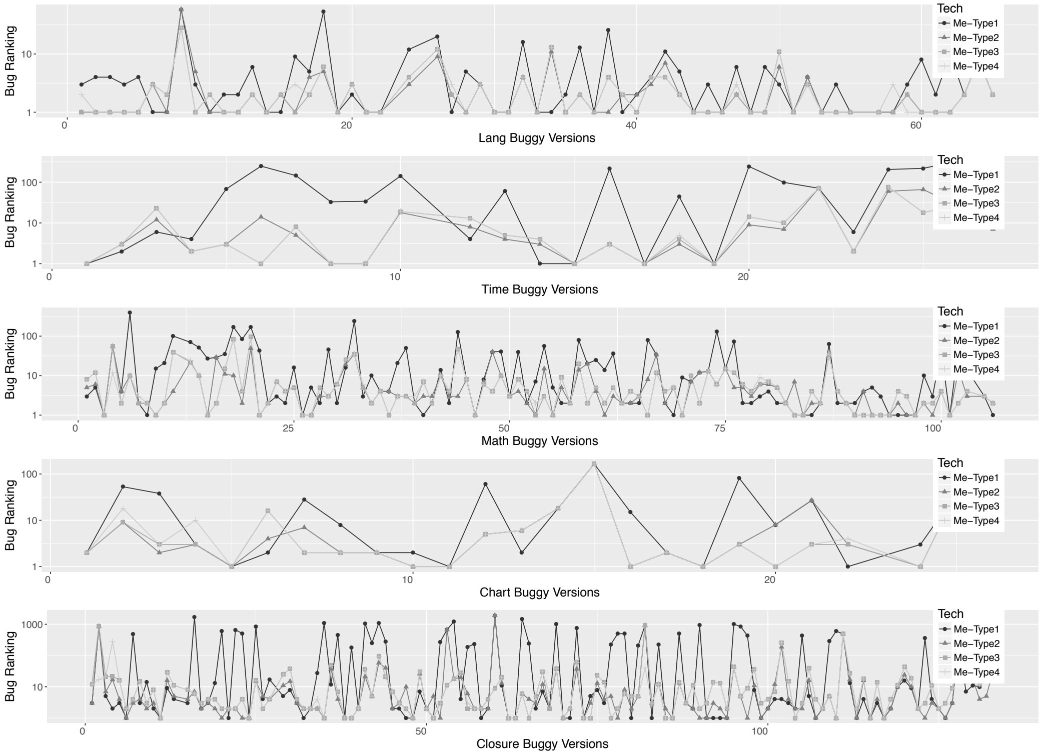

Fig. 11. The impacts of different failure message types on localizing the first bugs using Metallaxis

0,1,2,...,6 (0 denotes the default assertion). Without mutation, the original failed assertions are `a-3` and `a-4` at the assertion level. The results of mutation testing show that `a-3` and `a-4` still fail and are not impacted by mutations; furthermore, a passed assertion `a-1` is now impacted to fail due to mutations. Therefore, the assertion-level information ranks the buggy method quite low. On the contrary, at the test level, the failed test originally failed due to `a3` but now fails due to `a-1` in case of mutations and the failure message changes since `a-1` and `a-3` throw different exception messages. Therefore, the test level is able to detect that the buggy method can impact failed tests, and thus ranks it high. In summary, there is one failed test which has a changed value at the test level, but there is no changed-value failed tests at the assertion level. In this case, the rank of buggy method is better at the test level than at the assertion level. However, for bug Math-62, the rank of buggy method is 1st at the assertion level, but 5th at the test level since the detailed assertions capture the impact information in a finer-grained level.

To investigate whether mutation information at different levels (i.e., test or assertion level) has statistically different impacts on fault localization, we further applied the One-way Repeated Measures ANOVA analysis (i.e., rANOVA) on the fault localization results using the two different mutation information levels to investigate their statistical differences. Table 17 presents the rANOVA analysis results on MFR and MAR values of Metallaxis and MUSE

Table 17. rANOVA analysis for test/assertion levels

		Df	Sum Sq	Mean Sq	F-value	Pr(>F)
Metallaxis-MFR	MutationLevel	1	2613	2613.4	4.873	0.0283*
	Residuals	216	115843	536.3		
Metallaxis-MAR	MutationLevel	1	2991	2990.9	8.534	0.00385**
	Residuals	216	75697	350.4		
Muse-MFR	MutationLevel	1	1993	1992.9	4.553	0.034*
	Residuals	216	94540	437.7		
Muse-MAR	MutationLevel	1	1658	1658	4.964	0.0269*
	Residuals	216	72141	334		

Significance codes p<0.001 (***) p<0.01 (**) p<0.05 (*)

Table 18. TraPT with test-level mutation information

Subjects	Techniques	XGBoost					LIBSVM				
		Top-1	Top-3	Top-5	MFR	MAR	Top-1	Top-3	Top-5	MFR	MAR
Lang	Ochiai	24	44	50	4.63	5.01	24	44	50	4.63	5.01
	Me-Ochiai	32	51	56	2.84	3.15	32	51	56	2.84	3.15
	MULTRIC	22	42	49	5.55	5.84	23	42	49	5.50	5.82
	TraPT$_T$	40	55	59	2.37	2.72	40	57	59	2.65	2.93
Time	Ochiai	6	11	13	15.96	18.87	6	11	13	15.96	18.87
	Me-Ochiai	7	12	15	12.35	14.82	7	12	15	12.35	14.82
	MULTRIC	4	10	11	23.31	26.82	6	13	13	24.58	27.31
	TraPT$_T$	9	16	18	13.96	16.74	6	12	16	12.88	14.18
Math	Ochiai	23	52	62	9.73	11.72	23	52	62	9.73	11.72
	Me-Ochiai	20	50	70	7.84	9.41	20	50	70	7.84	9.41
	MULTRIC	22	46	58	14.34	16.32	19	48	58	10.41	12.64
	TraPT$_T$	35	71	81	5.88	7.50	32	66	79	5.58	7.43
Chart	Ochiai	6	14	15	9.00	9.51	6	14	15	9.00	9.51
	Me-Ochiai	7	15	17	12.68	13.31	7	15	17	12.68	13.31
	MULTRIC	5	12	15	25.96	27.00	7	15	16	8.08	8.85
	TraPT$_T$	9	15	19	6.76	7.26	10	15	16	5.20	5.86
Closure	Ochiai	14	30	38	90.28	102.28	14	30	38	90.28	102.28
	Me-Ochiai	20	52	69	21.72	26.14	20	52	69	21.72	26.14
	MULTRIC	9	20	26	127.12	141.93	17	31	41	87.34	100.85
	TraPT$_T$	41	72	82	11.91	17.92	54	85	94	13.95	18.97
Overall	Ochiai	73	151	178	39.56	44.99	73	151	178	39.56	44.99
	Me-Ochiai	86	180	227	12.86	15.28	86	180	227	12.86	15.28
	MULTRIC	62	130	159	56.73	63.29	72	149	177	39.39	45.46
	TraPT$_T$	134	229	259	8.19	11.24	142	235	264	8.73	11.37

across all the bugs from the four subject systems with assertion level mutation information. From the table, we observe that the rANOVA p-value using different levels of mutation information is always below 0.05 in terms of both MAR and MFR metrics for both Metallaxis and MUSE, rejecting the null hypothesis at the level of 0.05.

Finding 5: The analysis results demonstrate that the test-level and assertion-level mutation information also have significantly different effectiveness on fault localization and can be applied in tandem to further boost fault localization.

Combining both RQ2 and RQ3, the mutation information using different failure message types for both tests and assertions can all provide useful hints for fault localization. This finding motivates us to combine all the available mutation information to further improve the effectiveness of fault localization via Learning-to-Rank .

5.4 RQ4: Learning-to-Rank with Both Spectrum and Mutation Information

TraPT with Test Level Information. We first use suspiciousness values from 34 spectrum-based formulae to implement the traditional Learning-to-Rank technique, MULTRIC [Xuan and Monperrus 2014], which is purely based on different spectrum-based techniques. Then we introduce suspiciousness values from mutation-based fault localization using different failure message types at the test level and 34 spectrum-based fault localization, totaling 174 features[14], to implement our Learning-to-Rank technique, denoted as TraPT$_T$. Table 18 presents the main comparison results for our TraPT$_T$ technique with the overall most effective spectrum-based and mutation-based techniques as well as MULTRIC. The experimental result show that our Learning-to-Rank technique performs surprisingly well using both the LIBSVM and XGBoost Learning-to-Rank libraries. For example, overall, our TraPT$_T$ using XGBoost is able to localize 134 bugs within Top-1, which outperforms the most effective spectrum-based technique, the most effective mutation-based technique, and MULTRIC by 83.56%, 55.81%, and 116.13%, respectively. In terms of MFR/MAR,

[14]That is, 34 spectrum formulae, 34 spectrum formulae * 4 different failure message types for Metallaxis, and 4 different failure message types for MUSE.

Table 19. p-value of Wilcoxon tests between TraPT$_T$ and other techniques

Subjects	MAR Me-Ochiai	MAR Ochiai	MAR MULTRIC	MFR Me-Ochiai	MFR Ochiai	MFR MULTRIC
Chart	❍(0.1264)	❍(0.5524)	❍(0.4951)	❍(0.2249)	❍(0.2997)	❍(0.4158)
Time	❍(0.4437)	❍(0.394)	❍(0.0723)	❍(0.6723)	❍(0.5086)	❍(0.0512)
Lang	✓(0.0039)	✓(0.0003)	✓(1.551e-05)	✓(0.0056)	✓(0.0005)	✓(2.986e-05)
Math	✓(0.0005)	✓(9.989e-05)	✓(3.085e-06)	✓(1.800e-05)	✓(5.443e-05)	✓(1.797e-06)
Closure	✓(1.515e-05)	✓(2.536e-12)	✓(1.214e-12)	✓(1.124e-07)	✓(9.761e-14)	✓(5.666e-14)
Overall	✓(6.283e-11)	✓(7.550e-19)	✓(3.656e-22)	✓(1.632e-13)	✓(2.172e-20)	✓(4.505e-24)

Table 20. TraPT including both test and assertion level information

Subjects	Techniques	XGBoost Top-1	XGBoost Top-3	XGBoost Top-5	XGBoost MFR	XGBoost MAR	LIBSVM Top-1	LIBSVM Top-3	LIBSVM Top-5	LIBSVM MFR	LIBSVM MAR
Lang	TraPT$_T$	40	55	59	2.37	2.72	40	57	59	2.65	2.93
	TraPT$_{TA}$	41	57	59	2.23	2.63	45	57	60	1.84	2.39
	TraPT$_{TA}$(Cross Proj)	42	58	60	1.84	2.11	41	59	59	1.90	2.29
Time	TraPT$_T$	9	16	18	13.96	16.74	6	12	16	12.88	14.18
	TraPT$_{TA}$	7	16	18	12.46	14.85	9	14	16	14.00	15.80
	TraPT$_{TA}$(Cross Proj)	6	15	17	9.96	11.26	7	15	17	12.04	13.44
Math	TraPT$_T$	35	71	81	5.88	7.50	32	66	79	5.58	7.43
	TraPT$_{TA}$	46	78	87	5.15	7.22	37	69	85	5.34	6.95
	TraPT$_{TA}$(Cross Proj)	50	74	83	4.80	6.19	45	74	84	4.58	6.01
Chart	TraPT$_T$	9	15	19	6.76	7.26	10	15	16	5.20	5.86
	TraPT$_{TA}$	8	16	19	6.84	7.33	10	16	17	7.68	8.18
	TraPT$_{TA}$(Cross Proj)	9	14	15	6.04	6.52	10	16	20	6.72	7.35
Overall	TraPT$_T$	93	157	177	5.94	7.21	88	150	170	5.57	6.77
	TraPT$_{TA}$	102	167	183	5.39	6.84	101	156	178	5.65	6.85
	TraPT$_{TA}$(Cross Proj)	107	161	175	4.71	5.67	103	164	180	4.95	5.99

TraPT$_T$ is also able to outperform existing techniques by at least 36.31%/26.44%. The results using the default LIBSVM seem even better – TraPT$_T$ with LIBSVM localizes 142 bugs within Top-1, outperforming Ochiai, Me-Ochiai, and MULTRIC with LIBSVM by 94.52%, 65.12%, and 97.22%, respectively. The experimental results indicate that mutation information incorporating different failure message types together with spectrum information can significantly boost the effectiveness of fault localization.

To further investigate statistical differences between our TraPT$_T$ with other techniques, we apply Wilcoxon Signed-Rank test[15] [Wilcoxon 1945] at the significance level of 0.05 between TraPT$_T$ with MULTRIC, Ochiai and Me-Ochiai separately. Note that we only show the results using our default LIBSVM Learning-to-Rank library, and the results using XGBoost follow the same trend. Shown in Table 19, ✓ represents that there is statistical difference and TraPT$_T$ performs significantly better than other techniques; ✗ represents that there is statistical difference and TraPT$_T$ performs significantly worse than other techniques; ❍ represents that there is no statistical difference between techniques. In addition, the values inside brackets represent the corresponding p-values. From the analysis results, we can observe that TraPT$_T$ performs significantly better than all the other techniques on the majority subjects, and never performs significantly worse than any other technique. Note that for the two subjects with no statistical difference, TraPT$_T$ can also outperform the existing techniques in the majority cases, and the absence of statistical difference is due to the small number of bugs for those two subjects (e.g., 26 bugs for Chart and 27 for Time). Furthermore, shown in Row "Overall", TraPT$_T$ performs significantly better than all other techniques across all the used subjects.

Finding 6: TraPT$_T$ with test level information can significantly outperform state-of-the-art fault localization techniques, e.g., TraPT$_T$ via LIBSVM can outperform Ochiai, Me-Ochiai, and MULTRIC in terms of Top-1 bugs by 94.52%, 65.12%, and 97.22%, respectively.

[15] We use Wilcoxon Signed-Rank test because it does not have the assumption that the data should follow normal distribution.

TraPT with Both Test and Assertion Level Information. According to the analysis of RQ3, mutation information at the assertion level also plays an important role. Therefore, we further add 140 features computed from assertion-level mutation-based techniques (i.e., 34 spectrum formulae * 4 different failure message types for Metallaxis, and 4 different failure message types for MUSE) to TraPT$_T$ to form the more advanced TraPT$_{TA}$. Table 20 presents the results of TraPT$_{TA}$ on the 4 subjects with assertion-level information (TraPT$_T$ is also included for comparison). The experimental results show that TraPT$_{TA}$ further boosts TraPT$_T$ significantly. For example, TraPT$_{TA}$ with XGBoost/LIBSVM localizes 102/101 bugs within Top-1, 9.68%/14.77% more than TraPT$_T$. However, we also observe that the assertion-level information cannot help much with the overall fault localization results (e.g., in terms of MAR/MFR) for LIBSVM.

Finding 7: TraPT with both test and assertion level information can further greatly improve the number of top-ranked bugs (e.g., 14.77% more Top-1 bugs than TraPT$_T$ via LIBSVM), but cannot help much with the overall fault localization results for LIBSVM.

Cross-project Prediction. So far, our cross validation setting has been following prior Learning-to-Rank fault localization work [B Le et al. 2016; Xuan and Monperrus 2014], i.e., performing cross validation for each of the studied projects [B Le et al. 2016]. To investigate whether bug data of other projects can further boost TraPT's effectiveness, we further extend our experimental setting to perform cross validation across different projects. That is, when performing fault localization on one buggy version of one subject, we perform leave-one-out cross validation by using all the other buggy versions from the same project and other projects for training. The "TraPT$_{TA}$(Cross Proj)" technique in Table 20 represents this new setting. According to the experimental results, bug data from other projects can further help boost TraPT$_{TA}$. For example, the cross-project validation can further improve TraPT$_{TA}$ with XGBoost by 12.62% and 17.11% in terms of MFR and MAR, respectively. We also notice that cross-project validation only improves the Top-N values slightly, indicating that bug data from other projects can help with the overall results, but cannot help much for the bugs already localized precisely.

Finding 8: Cross-project training can further improve the overall fault localization effectiveness (e.g., 10+% further improvement over within-project TraPT$_{TA}$ via XGBoost in MFR/MAR), but has limited help in terms of the number of top-ranked bugs.

Learning Overhead. Tables 21 and 22 present the time cost of Learning-to-Rank via LIBSVM and XGBoost on the first version (i.e., the latest version) of each studied subject. Shown in Column "TraPT$_T$", the training time of "TraPT$_T$" ranges from 1.09 seconds to 25.89 seconds using LIBSVM, while ranging from 21.41 seconds to 71.99 seconds using XGBoost; the prediction time of "TraPT$_T$" ranges from 0.08 seconds to 127.89 seconds using LIBSVM, while ranging from 0.05 seconds to 0.16 seconds using XGBoost. These results show that both LIBSVM and XGBoost can be quite efficient for TraPT$_T$ in practice. In addition, due to the different machine learning algorithms used, LIBSVM tends to be faster in training while XGBoost is much faster for prediction. For example, for Closure, the size of the generated classification model

Table 21. Training and prediction time of via LIBSVM

Subjects	TraPT$_T$		TraPT$_{TA}$		TraPT$_{TA}$ (Cross Proj)	
	Train	Prediction	Train	Prediction	Train	Prediction
Chart-1	1.09s	0.44s	1.85s	0.43s	7.34s	2.68s
Time-1	1.42s	0.30s	2.47s	0.45s	6.68s	1.86s
Lang-1	1.39s	0.08s	0.80s	0.21s	7.41s	1.35s
Math-1	2.24s	0.30s	4.17s	0.79s	7.28s	1.24s
Closure-1	25.89s	127.89s	✗	✗	✗	✗

Table 22. Training and prediction time via XGBoost

Subjects	TraPT$_T$		TraPT$_{TA}$		TraPT$_{TA}$ (Cross Proj)	
	Train	Prediction	Train	Prediction	Train	Prediction
Chart-1	24.10s	0.16s	21.92s	0.06s	65.07s	0.09s
Time-1	21.41s	0.05s	21.94s	0.06s	63.33s	0.05s
Lang-1	27.74s	0.06s	21.04s	0.05s	64.89s	0.04s
Math-1	58.72s	0.05s	44.60s	0.05s	64.64s	0.29s
Closure-1	71.99s	0.08s	✗	✗	✗	✗

is 181M for LIBSVM, while being only 255K for XGBoost (because XGBoost uses a parallel tree boosting and is designed to be highly efficient, flexible and portable), making the prediction for XGBoost much faster. Shown in Column "TraPT$_{TA}$", the training and prediction time for TraPT$_{TA}$ using both LIBSVM and XGBoost is similar or slightly longer than that for TraPT$_T$, indicating that adding more attributes does not degrade the performance much. Shown in Column "TraPT$_{TA}$ (Cross Proj)", the training time and prediction time of TraPT$_{TA}$ tend to be longer in the cross-project scenario, but is still quite practical, e.g., less than 70 seconds for all the subjects except Closure (Note that TraPT$_{TA}$ is not applicable to Closure).

Finding 9: The Learning-to-Rank process for both TraPT$_T$ and TraPT$_{TA}$ can be quite lightweight in practice, e.g., costing less than 3 minutes for even the largest studied subject. Also, our results demonstrate that while LIBSVM is faster for training, XGBoost is much faster in prediction, providing practical guidelines for practitioners.

5.5 Threats to Validity

Threats to internal validity are mainly concerned with the uncontrolled factors that may also be responsible for the results. In this work, the main threat to internal validity is the potential faults in the implementation of our own techniques as well as the reimplementation of existing techniques. To reduce this threat, we built our techniques on top of state-of-the-art tools and frameworks, such as ASM byte-code manipulation framework, Java Agent, PIT, XStream, Commons Codec, LIBSVM, and XGBoost. We also reimplemented existing techniques/tools strictly following their original work. Furthermore, we carefully review all our code and experimental scripts to ensure their correctness. However, there is still a risk of introducing subjectivity during the code review, thus introducing potential implementation or experimentation flaws.

Threats to construct validity are mainly concerned with whether the measurements used in our study reflect real-world situations. To reduce this threat, we use various widely used metrics to measure the effectiveness of fault localization, e.g., the Top-N metric, the rank of the all faulty elements, as well as the rank of the first faulty element for each fault. Furthermore, we also compare all the studied techniques in the same experimental settings. To further reduce this threat, we plan to perform user studies to evaluate the actual user debugging efforts in localizing the real faults when using different techniques. Furthermore, we also plan to investigate the effectiveness of the proposed techniques in helping with automated program repair [Le Goues et al. 2012; Long and Rinard 2015; Nguyen et al. 2013; Xiong et al. 2017].

Threats to external validity are mainly concerned with whether the findings in our study are generalizable for other experimental settings. The subjects, tests, and faults used in this work may also introduce threats to external validity. To reduce these threats, we use the five subjects from the widely used Defects4J benchmark suite. Furthermore, we use the tests and real faults accumulated during the real software development cycle. However, they may still not be representative of all the available subjects, tests, and faults. To further reduce these threats, we plan to evaluate on more real-world projects with different sizes and application domains.

6 RELATED WORK

In this section, we discuss our related work in fault localization. Note that although there are also a huge amount of work on information-retrieval based fault localization [Le et al. 2015; Saha et al. 2013; Zhou et al. 2012], they do not utilize any test execution information, and only rely on well-formed bug reports (which may not always be available [Wang et al. 2015]) for static fault localization. Therefore, we mainly discuss about the most closely related work in spectrum and mutation based fault localization.

Spectrum-based Fault Localization. To date, a number of basic formulae for spectrum-based fault localization have been proposed. For example, Jones et al. firstly proposed Tarantula [Jones and Harrold 2005] to rank statements by distinguishing the executions of passed and failed tests. The SBI [Liblit et al. 2005] formula was originally introduced by Liblit et al. to compute the suspiciousness of program predicates, and has been applied to general spectrum-based fault localization. Abreu et al. then introduced Ochiai [Abreu et al. 2006] and Jaccard [Abreu et al. 2007] for spectrum-based fault localization. More recently, Naish et al. [Naish et al. 2011] proposed Kulczynski2 and Ochiai2 to improve previous formulae (e.g., Ochiai) by additionally considering the influence of non-executed or passed tests. All the above formulae have been widely used in spectrum-based fault localization.

Besides investigating different suspiciousness computation formulae, researchers have also considered other dimensions for improving spectrum-based fault localization. For example, Santelices et al. [Santelices et al. 2009] combined three types of coverage information (i.e., statements, branches, and data dependencies) to build a more effective fault localization framework. Baah et al. [Baah et al. 2011] proposed a novel causal-inference technique to reduce the confounding bias of dynamic data and control dependencies in order to improve the fault localization. Gong et al. [Gong et al. 2012] proposed to order unlabeled tests based on diversity maximization speedup to help reduce the expensive test oracle efforts during fault localization. Lucia et al. [Lucia et al. 2014] proposed to treat fault localization as a measurement of the relationship between the execution of program elements and test failures, and empirically studied the effectiveness of various existing association measures from the literature on fault localization. Gopinath et al. [Gopinath et al. 2012] proposed to apply spectrum-based localization with specification-based analysis in tandem to localize faults more accurately. Daniel et al. [Daniel et al. 2014] investigated an improved technique to clone the execution profiles of fail tests beyond balanced test suite to improve the performance of spectrum-based fault localization. However, a common limitation for those spectrum-based fault localization techniques is that they only focus on the coverage information without considering the impact of the code elements to the program correctness and test outcomes, and thus can have limited effectiveness in practice [Parnin and Orso 2011].

Mutation-based Fault Localization. Mutation-based fault localization [Moon et al. 2014; Papadakis and Le Traon 2012, 2015; Zhang et al. 2013] was proposed to improve spectrum-based fault localization by considering the actual impacts of code elements using mutation testing [Jia and Harman 2011; Offutt et al. 1993]. The main idea of mutation-based fault localization is to inject mutation faults to each code element to simulate its impact on test outcomes. Papadakis et al. [Papadakis and Le Traon 2012] was the first to use mutation testing results to replace the coverage information used by spectrum-based fault localization, and demonstrated that mutation testing information can be more effective than the widely used statement coverage information on fault localization. Meanwhile, Zhang et al. [Zhang et al. 2013] firstly used mutation testing results to simulate the impact of program edits during software evolution, and proposed FIFL, a framework for localizing failure-inducing program edits. Later on, Papadakis et al. [Papadakis and Le Traon 2014, 2015] further studied to reduce the cost of mutation-based fault localization via mutant sampling. Moon et al. [Moon et al. 2014] also proposed a new mutation-based fault localization formulae, MUSE, based on the different impacts of mutating correct and faulty statements. Different from spectrum-based fault localization, mutation-based fault localization focuses on mutating source code to investigate the impacts of each code element to help with more precise fault localization. Recently, Pearson et al. [Pearson et al. 2017] performed an extensive study to compare the effectiveness of mutation-based fault localization on artificial and real bugs at the statement level. They found that while mutation-based fault localization performs well on artifitial bugs, it cannot even outperform spectrum-based fault localization on real bugs. We think the reason to be that the number of mutants can be quite small or frequently zero for each statement,

making mutation-based fault localization unable to compute the suspiciousness values for many statements, and thus perform poorly at the statement level. In contrast, in this work, we perform an extensive study of mutation-based fault localization for localizing real bugs at the method level, and firstly demonstrate that mutation-based fault localization can be much more effective than state-of-the-art spectrum-based techniques but sometimes unstable. Then, we further studied the impacts of different failure message types and mutation information levels on mutation-based fault localization. Finally, our study results motivate us to improve mutation-based fault localization by transforming programs and tests in tandem via Learning-to-Rank algorithms.

7 CONCLUSION

Fault localization is essential for both manual debugging as well as automated program repair. In this paper, we first present an extensive study on the effectiveness of mutation-based fault localization on real bugs of modern real-world programs. Our study results confirm the effectiveness of mutation-based fault localization, and also reveal various guidelines to further improve mutation-based fault localization. Based on the learnt guidelines, we propose TraPT, the first (Learning-to-Rank) approach that transforms both programs and tests in tandem to achieve precise fault localization. More specifically, TraPT further transforms test outputs/messages and test code to record detailed test execution information while transforming the source code via mutation testing to check the detailed impacts of each code element. Furthermore, we also empirically studied our TraPT with state-of-the-art fault localization techniques on 357 real faults from Defects4J. Our experimental results show that TraPT with the default setting of LIBSVM is able to outperform state-of-the-art mutation-based and spectrum-based fault localization by 65.12% and 94.52% in localizing Top-1 bugs, indicating a promising future for investigating fault localization via transforming both source code and tests.

ACKNOWLEDGEMENTS

This work is supported in part by NSF Grant No. CCF-1566589, UT Dallas faculty start-up fund, and Google Faculty Research Award. We also thank the anonymous reviewers for the valuable comments that help improve the paper significantly during the two-stage review process.

REFERENCES

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*. 39–46.

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 89–98.

Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 49–60.

Tien-Duy B Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 177–188.

George K Baah, Andy Podgurski, and Mary Jean Harrold. 2011. Mitigating the confounding effects of program dependences for effective fault localization. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 146–156.

Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*. ACM, 89–96.

Christopher JC Burges, Robert Ragno, and Quoc Viet Le. 2006. Learning to rank with nonsmooth cost functions. In *NIPS*, Vol. 6. 193–200.

Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 2, 3 (2011), 27.

Patrick Daniel, Kwan Yong Sim, and Soonuk Seol. 2014. Improving Spectrum-based Fault-localization through Spectra Cloning for Fail Test Cases Beyond Balanced Test Suite. *Contemporary Engineering Sciences* 7 (2014), 677–682.

Tung Dao, Lingming Zhang, and Na Meng. 2017. How does execution information help with information-retrieval based bug localization?. In *Proceedings of the 25th International Conference on Program Comprehension.* 241–250.

Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *Computer* 4 (1978), 34–41.

Giovanni Denaro, Alessandro Margara, Mauro Pezze, and Mattia Vivanti. 2015. Dynamic data flow testing of object oriented systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 947–958.

Görschwin Fey, Stefan Staber, Roderick Bloem, and Rolf Drechsler. 2008. Automatic fault localization for property checking. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27, 6 (2008), 1138–1149.

Yoav Freund, Raj Iyer, Robert E Schapire, and Yoram Singer. 2003. An efficient boosting algorithm for combining preferences. *Journal of machine learning research* 4, Nov (2003), 933–969.

Ellen R Girden. 1992. *ANOVA: Repeated measures.* Number 84. Sage.

Liang Gong, Daniel Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Diversity maximization speedup for fault localization. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on.* IEEE, 30–39.

Divya Gopinath, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2012. Improving the effectiveness of spectra-based fault localization using specifications. In *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on.* IEEE, 40–49.

Andreas Griesmayer, Stefan Staber, and Roderick Bloem. 2007. Automated fault localization for C programs. *Electronic Notes in Theoretical Computer Science* 174, 4 (2007), 95–111.

Richard G Hamlet. 1977. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on* 4 (1977), 279–290.

Schuyler W Huck and Robert A McLean. 1975. Using a repeated measures ANOVA to analyze the data from a pretest-posttest design: A potentially confusing task. *Psychological Bulletin* 82, 4 (1975), 511.

Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on* 37, 5 (2011), 649–678.

James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering.* ACM, 273–282.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA).* San Jose, CA, USA, 437–440.

Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* ACM, 165–176.

Tien-Duy B Le, Richard J Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 579–590.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on.* 3–13.

Ching-Pei Lee and Chih-Jen Lin. 2014. Large-scale linear ranksvm. *Neural computation* 26, 4 (2014), 781–817.

Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, Vol. 40. 15–26.

Tie-Yan Liu. 2009. Learning to rank for information retrieval. *Foundations and Trends® in Information Retrieval* 3, 3 (2009), 225–331.

Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* 166–178.

Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the 38th International Conference on Software Engineering.* 535–546.

Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. 2014. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process* 26, 2 (2014), 172–219.

Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on.* IEEE, 153–162.

Jakub Možucha and Bruno Rossi. 2016. Is Mutation Testing Ready to Be Adopted Industry-Wide?. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17.* Springer, 217–232.

Vincenzo Musco, Martin Monperrus, and Philippe Preux. 2016. A large-scale study of call graph-based impact prediction using mutation testing. *Software Quality Journal* (2016), 1–30.

Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 11.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. 772–781.

A Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An experimental evaluation of selective mutation. In *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 100–107.

Mike Papadakis and Yves Le Traon. 2012. Using mutants to locate" unknown" faults. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 691–700.

Mike Papadakis and Yves Le Traon. 2014. Effective fault localization via mutation analysis: A selective mutation approach. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 1293–1300.

Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 199–209.

Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*. 609–620.

Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 345–355.

Raul Santelices, James A Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 56–66.

August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 246–256.

Ming-Feng Tsai, Tie-Yan Liu, Tao Qin, Hsin-Hsi Chen, and Wei-Ying Ma. 2007. FRank: a ranking method with fidelity loss. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 383–390.

CARL N von Ende. 2001. Repeated-measures analysis. *Design and analysis of ecological experiments. Oxford University Press, Oxford* (2001), 134–157.

Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. An In-Depth Study of IR-Based Fault Localization Techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*. To appear.

Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics bulletin* 1, 6 (1945), 80–83.

W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective fault localization using code coverage. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, Vol. 1. IEEE, 449–456.

Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*. 416–426.

Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 191–200.

Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 342–353.

Lingming Zhang, Milos Gligoric, Darko Marinov, and Sarfraz Khurshid. 2013. Operator-based and random mutant selection: Better together. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 92–102.

Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *OOPSLA*. 765–784.

Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using PageRank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 261–272.

Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Software Engineering (ICSE), 2012 34th International Conference on*. 14–24.