Assessing and Comparing Mutation-based Fault Localization Techniques

Thierry Titcheu Chekam
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
thierry.titcheu-chekam@uni.lu

Mike Papadakis
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
yves.letraon@uni.lu

ABSTRACT

Recent research demonstrated that mutation-based fault localization techniques are relatively accurate and practical. However, these methods have never been compared and have only been assessed with simple hand-seeded faults. Therefore, their actual practicality is questionable when it comes to real-wold faults. To deal with this limitation we asses and compare the two main mutation-based fault localization methods, named Metallaxis and MUSE, on a set of real-world programs and faults. Our results based on three typical evaluation metrics indicate that mutation-based fault localization methods are relatively accurate and provide relevant information to developers. Overall, our result indicate that Metallaxis and MUSE require 18% and 37% of the program statements to find the sought faults. Additionally, both methods locate 50% and 80% of the studied faults when developers inspect 10 and 25 statements.

Keywords

Fault Localization; Mutation Analysis; Debugging; Real Faults

1. INTRODUCTION

Software debugging is an essential software development activity. A typical debugging scenario consists of the following three phases: fault localization, fault comprehension and fault repair. Fault localization refers to the process when developers try to identify statements that they suspect that are problematic. Fault comprehension refers to the process when developers try to understand the actual problem with their code. Once they understand it they try to fix it. Each one of these phases is painful, costly and thus, research is dedicated to (semi)automate them.

Fault locazation is the basic step for the above scenario since both fault comprehension and repair are strongly depended on it. Thus, it is natural to expect that more accurate fault localizaton leads to faster fault comprehension and repair. In view of this, spectrum-based fault localization methods have been developed. These methods approximate the likelihood that program statements are faulty given the execution profile of failed and passing test cases. Then, they rank all the program statements in a decreasing likelihood order in an attempt to guide developers.

Spectrum-based fault localization techniques have been studied with the use of different spectra types, i.e., statements, branches data-flows, and different likelihood estimation formulas. However despite the efforts of the community, they are relatively inaccurate [29]. This is partly due to the fact that coverage is not correlated with test effectiveness, which results in the so-called coincidental correctness [17], and partly due to the small number of test cases that are typically available. Therefore, it is quite hard, if not impossible, to relate spectra elements with actual faults. Another problem with spectrum-based fault localization is that the ordered statement list contains many statements that are unrelated to the sought fault [29].

The present paper studies the mutation-based fault localization techniques [26], which aim at addressing the above-mentioned limitations of the spectrum-based techniques. The underlying idea of these techniques is to gain information from mutants, which have been shown to correlate with faults [4], and provide guidance regarding the identified faults. Despite the promising results of these methods, today there is no evidence on how well these methods perform in real-world cases, i.e., in localizing real-world faults. This form the primary objective of the this paper.

In literature, two mutation-based fault localization techniques, named as Metallaxis [26] and MUSE [18], have been proposed. Metallaxis realises the idea that mutants are coupled with faults (killing mutants results in finding faults) and localizes them based on the mutants that couple with the fault according to the test suite. MUSE is inspired by the automated fault repair tools and localizes faults based on mutants that turn failing executions into passing ones. Given the differences of the two approaches, a natural question to ask is which one is more effective and why. Since, no comparison between them has been attempted a secondary objective of this paper is to compare these techniques and provide insights on when and why one approach performs better than the other.

For evaluation we use CoREBench, a benchmark of complex real-world faults and show that they provide relatively accurate results. In particular our result indicate that developers using Metallaxis and MUSE successfully locate the studied faults by investigating only 18% and 37% of the executable program statements. The results also indicate that Metallaxis was two times less costly than MUSE.

Our study also reveal that Metallaxis can locate approximately 50% or 80% of the studied faults by investigating only 10 or 25 statements, respectively. This is particularly important finding since it ensures that developers will not experience a drop-off on their interest for the fault localization [29].

Another major advantage offered by these techniques is that the most suspicious statements are related to the actual faulty program statements. Our result indicate that the average relevance of the top 10 statements is 28.16% and 25.88% for Metallaxis and Muse, meaning that on average 3 out of the first 10 statements can lead to the sought fault. This was a key requirement set by the user study of Parin and Orso [29].

Overall, the contributions of the present paper can be summarised on the following points:

- We evaluate the accuracy of mutation-based fault localisation methods on real-world programs and faults.
- We compare and analyse the performance of the mutation-based fault localisation methods.
- We perform a qualitative analysis of the studied approaches.

The rest of the paper is organized as follows: Sections 2 and 3 respectively detail the studied approaches and the design of the conducted study. Sections 4 and 5 are then analysing the results of the study, in a quantitative and qualitative manner, respectively. Then, Section 6 and Section 7 discusses threats to validity and related work. Finally, Section 8 concludes the paper.

2. BACKGROUND

2.1 Mutation Analysis

Mutation is a software analysis technique that alters (mutates) the syntax of the program under analysis with the intention to produce programs that have small semantic differences from the original one [20]. The produced programs are called mutants and are constructed by making simple syntactic transformations on the original program. The transformation rules that are used to produce the mutants are called mutant operators. Mutation has been demonstrated to be quite powerful, mainly due to its ability to force and expose the different behaviours of the program under analysis. A test analysis scenario involves observing the behaviour of the mutants when exercised by test cases. In case a mutant exhibit different behaviour from the original program it is called killed, while it is called live in the opposite case.

Mutation analysis differs from the coverage-based techniques because it forces the candidate test cases not only to execute specific program locations but also to exercise the sensitivity of these locations to trigger potential errors to the observable output. This requirement of mutation makes it particularly powerful with respect to testing and analysis. Mutants have been shown to be quite effective in mimicking the behaviour of real faults [4] and in performing thorough testing that reveal more faults than other test criteria [5]. The present paper realises the above attributes of mutation, in the context of fault localization, and ultimately examines the ability of mutants to effectively point out the location of real faults.

$$Suspiciousness(e) = \frac{failed(e)}{\sqrt{totfailed * (failed(e) + passed(e))}}$$

Figure 1: The Ochiai formula. In the formula: e represents a mutant, totfailed - the total number of test cases that fail, failed(e) - the number of test cases that kill the mutant e and fail and passed(e) - the number of test cases that kill mutant e and pass.

2.2 Mutation-based fault localization

Fault localization techniques collect dynamic information of the program under analysis and try to associate the executed statements with the experienced failures. In other words they try to identify program statements that are most likely to be faulty. Thus, for every program statement they estimate and assign a value that represents the probability that it is faulty. This value is called suspiciousness value. To find the faults, users have to inspect the program statements according to their suspiciousness. Thus, fault localization methods produce a priority list of statements that the users have to follow. Statements position in this list is called rank and it is used as a basis of comparison between fault localization methods.

The underlying idea of fault localization is that entities covered by failing tests are more likely to be responsible for failures than entities covered mostly by passing tests. Mutation-based fault localization realises this idea with mutants. Thus, mutants killed by failing tests and rarely by passing tests are more likely to relate with failures than mutants killed mostly by passing tests. In this paper we are considering two fault localization methods named as Metallaxis [26] and MUSE [18].

Metallaxis assigns suspiciousness values to the employed mutants using the Ochiai similarity function (shown in figure 1). Then, based on the location of the mutants it assigns a suspiciousness value on the program statements (using the maximum value of the included mutants). Additional details regarding Metallaxis can be found in the work of Papadakis and Le Traon [26].

MUSE follows a different way in identifying suspicious statements. Instead of using the "traditional" way of judging whether mutants are killed or not (based on program outputs [3]), it only considers mutants as killed (or not) based on the passing and failing tests. Thus, MUSE only considers the cases that mutants turn a passing test case to a failing one and vice versa. Thus, it ignores the cases that both mutants and the original (faulty) program fail but in different ways, i.e., their outputs differ.

MUSE assigns suspiciousness values to the program statements using the formula of figure 2. The first term $\frac{|f_P(s) \cap p_m|}{|f_P|}$ represents the proportion of tests that were turned from fail-

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

Figure 2: The suspiciousness metric of MUSE. In the formula: mut(s) represents the number of mutants residing on the statement s, the term $\frac{|f_P(s) \cap p_m|}{|f_P|}$ - the proportion of tests that were turned from failing to passing, the term $\frac{|p_P(s) \cap f_m|}{|p_P|}$ - the proportion tests that were turned from passing to failing, the value a adjusts the average values of the two terms.

	Metallaxis												MUSE										
	Mutants	Statements	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	#Passed	#Failed	Suspiciousness	Rank	Statements	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	#P→F	# F→P	Suspiciousness	Rank
mid(**int** x, **int** y, **int** z){			3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	3, 1, 5						3, 3, 5	1, 2, 3	3, 2, 1	5, 5, 5	5, 3, 4	3, 1, 5				
int m;		*1*										*13*	*1*										*13*
m = z;	*M1. z → z+1*	*2*				√	√		*2*	*0*	*0*	*13*	*2*				P → F	P→F		*2*	*0*	*0*	*13*
	M2. z → z-1					√	√		*2*	*0*	*0*						P → F	P→F		*2*	*0*	*0*	
if (y < z)	*M3. y → y+1*	*3*		√			√		*2*	*0*	*0*	*13*	*3*		P→F			P→F		*2*	*0*	*0*	*13*
	M4. y → y-1								*0*	*0*	*0*									*0*	*0*	*0*	
	M5. z → z+1								*0*	*0*	*0*									*0*	*0*	*0*	
	M6. z → z-1			√			√		*2*	*0*	*0*				P→F			P→F		*2*	*0*	*0*	
if (x < y)	*M7. x → x+1*	*4*							*0*	*0*	*0*	*13*	*4*							*0*	*0*	*0*	*13*
	M8. x → x-1								*0*	*0*	*0*									*0*	*0*	*0*	
	M9. y → y+1								*0*	*0*	*0*									*0*	*0*	*0*	
	M10. y → y-1								*0*	*0*	*0*									*0*	*0*	*0*	
m = y;	*M11. y → y+1*	*5*		√					*1*	*0*	*0*	*13*	*5*		P→F					*1*	*0*	*0*	*13*
	M12. y → y-1			√					*1*	*0*	*0*				P→F					*1*	*0*	*0*	
else if (x < z)	*M13. x → x+1*	*6*							*0*	*0*	*0*	*13*	*6*							*0*	*0*	*0*	*13*
	M14. x → x-1								*0*	*0*	*0*									*0*	*0*	*0*	
	M15. z → z+1								*0*	*0*	*0*									*0*	*0*	*0*	
	M16. z → z-1								*0*	*0*	*0*									*0*	*0*	*0*	
m = y;	*M17. y → y+1*	*7*	√					√	*1*	*1*	*0.71*	*1*	*7*	P→F						*1*	*0*	*0*	*13*
//should be m = x;	*M18. y → y-1*		√					√	*1*	*1*	*0.71*			P→F						*1*	*0*	*0*	
else		*8*										*13*	*8*										*13*
if (x > y)	*M19. x → x+1*	*9*							*0*	*0*	*0*	*13*	*9*							*0*	*0*	*0*	*13*
	M20. x → x-1				√				*1*	*0*	*0*					P→F				*1*	*0*	*0*	
	M21. y → y+1				√				*1*	*0*	*0*					P→F				*1*	*0*	*0*	
	M22. y → y-1								*0*	*0*	*0*									*0*	*0*	*0*	
m = y;	*M23. y → y+1*	*10*			√				*1*	*0*	*0*	*13*	*10*			P→F				*1*	*0*	*0*	*13*
	M24. y → y-1				√				*1*	*0*	*0*					P→F				*1*	*0*	*0*	
else if (x > z)	*M25. x → x+1*	*11*							*0*	*0*	*0*	*13*	*11*							*0*	*0*	*0*	*13*
	M26. x → x-1								*0*	*0*	*0*									*0*	*0*	*0*	
	M27. z → z+1								*0*	*0*	*0*									*0*	*0*	*0*	
	M28. z → z-1								*0*	*0*	*0*									*0*	*0*	*0*	
m = x;	*M29. x → x+1*	*12*							*0*	*0*	*0*	*13*	*12*							*0*	*0*	*0*	*13*
	M30. x → x-1								*0*	*0*	*0*									*0*	*0*	*0*	
return m;	*M31. m → m+1*	*13*	√	√	√	√	√	√	*5*	*1*	*0.41*	*2*	*13*	P→F	P→F	P→F	P → F	P→F		*5*	*0*	*0*	*13*
	M32. m → m-1		√	√	√	√	√	√	*5*	*1*	*0.41*			P→F	P→F	P→F	P → F	P→F		*5*	*0*	*0*	
}			P	P	P	P	P	F						P	P	P	P	P	F				

Figure 3: Mutation-based fault localisation example when using Metallaxis and MUSE

ing to passing. Similarly, the second term $\frac{|p_P(s) \cap f_m|}{|p_P|}$ of the formula represents the proportion tests that were turned from passing to failing. The value a is used for balancing the two terms and hence it adjusts the average values of the terms. To compute the overall suspiciousness of a statement, the sum of the two terms is divided with the number of mutants residing on the statement. Further details regarding MUSE can be found in the work of Moon et al. [18].

2.3 Example

Figure 3 presents a working example of how the studied fault localization methods work. The example presents an implementation of a function that takes three integers as input and returns the median value. Assuming that there is a fault on the statement 7 and that the developer discovered it by using 6 test cases, 5 passing and 1 failing. For the shake of the example we will demonstrate the method by using only one mutant operator that adds or subtracts the value of 1 on every program statement that it is relevant.

Metallaxis works by executing the mutants with the 6 test cases and recording the number of passing and failing tests that kill (causing the program to provide different output) each one of the mutants (marked with '$\surd$'). Then, for every mutant it computes its suspiciousness value using the Ochiai formula 1. Finally the statements are assigned with the maximum value of mutant suspiciousness based on which they are ranked.

MUSE works by measuring the number of tests that are turned from passing to failing and from failing to passing (marked as 'P->F' and 'F->P'). Then, for every statement a suspiciousness value is assigned according to the formula presented in 2. Finally the statements are ranked according to their suspiciousness.

In the example of Figure 3 Metallaxis successfully ranked the faulty statement at the top of its list. This happened since the mutants M17 and M18 were only killed by the test case that failed, i.e., Test6. Unfortunately, since mutants M17 and M18 did not turned the execution of Test6 to passing, MUSE failed to locate the fault. It is noted that Metallaxis considers the contribution of the mutants by observing the output differences between the original and the mutant programs while MUSE by observing the differences between the expected output with that of the original and mutant programs.

3. STUDY DESIGN

3.1 Research Questions

The research questions investigated in our empirical study are the following:

RQ1: How Metallaxis and MUSE compare with the "optimal" fault localization?

RQ2: What are the accuracy differences between Metallaxis and MUSE?

RQ3: How relevant are the top ranked statements with the sought faults?

The first research question aims at evaluating the relative accuracy of the mutation-based fault localization techniques, i.e., Metallaxis and MUSE, in suggesting the locations, as line number in the source code, when compared to an 'optimal" fault localization. In the past, the two techniques have only been evaluated on hand-seeded faults [26, 18] with remarkably good accuracy. However, this might not be the case for the real faults that tend to be more complex than the hand-seeded ones.

The second research question aims at directly comparing the accuracy of Metallaxis and MUSE in recommending the suspicious faulty locations of the studied software faults. The focus here is on the accuracy differences between the two techniques.

The third research question investigates whether statements with the highest suspiciousness values are related to the sought faults. In other words, we seek to investigate whether developers can identify what went wrong by inspecting those statements. In practice localizing a fault requires locating the source code lines that are related and can easily lead to the faulty statement. In fact, some faults are easily understandood after reviewing several other lines related to the faulty ones [29].

3.2 Subjects

To conduct the experiment we used the bugs and patches from GNU Coreutils application suite[1], GNU Findutils[2] and GNU Grep[3] included in the CoREBench suite[4] [6].

Faults and Programs. CoREBench contains 22, 15 and 15 pairs of {*bug-introduction, bug-fixing*} versions of the Coreutils, Findutils and Grep. We consider bugs whose bug-fixes were made in the source file containing the C language `main` function; such selection result in a total of **30 bugs**. The executable source code lines (SLOC) changed during bug-fixing commit are considered as bug-fixes and chosen to be *Fault Locations*. For each bug (B), the newest version of the corresponding program preceding the bug-fixing commit is chosen as subject program (P).

Test Suites. For each bug B with corresponding faulty program P, the developer test suite included in P's repository is used in the experiments. We augment such test suite with the regression test cases added by B's bug-fixing commit and the regression tests shipped with CoREBench to test for B. We found that most of the test cases included in P's test suite are large script files containing multiple smaller test cases. This is a common phenomenon in practice which hinders the effectiveness of fault localization [34].

[1]http://www.gnu.org/software/coreutils/
[2]http://www.gnu.org/software/findutils/
[3]http://www.gnu.org/software/grep/
[4]http://www.comp.nus.edu.sg/ release/corebench/

Table 1: Used subjects and faults.

	ID	Prog.	# of SLOC	Source File	#Tests		
					Pass	Fail	All
Coreutils	1	rm	1	rm.c	70	2	72
	2	od	12	od.c	451	3	454
	3	cut	3	cut.c	108	1	109
	4	tail	6	tail.c	89	1	90
	5	tail	4	tail.c	84	5	89
	6	cut	1	cut.c	105	1	106
	7	seq	2	seq.c	60	1	61
	8	seq	9	seq.c	52	1	53
	9	seq	2	seq.c	45	5	50
	11	cut	4	cut.c	81	1	82
	12	cut	1	cut.c	77	2	79
	13	ls	3	ls.c	155	1	156
	14	ls	1	ls.c	154	1	155
	15	du	1	du.c	58	2	60
	16	tail	2	tail.c	82	1	83
	17	cut	2	cut.c	68	3	71
	18	seq	1	seq.c	34	1	35
	19	seq	19	seq.c	30	1	31
	20	seq	16	seq.c	23	7	30
	21	cut	3	cut.c	180	2	182
	22	expr	1	expr.c	20	2	22
Findutils	27	find	3	ftsfind.c	565	1	566
	32	find	2	ftsfind.c	324	1	325
	33	find	2	ftsfind.c	324	1	325
	35	find	2	ftsfind.c	296	1	297
	36	find	24	find.c	24	1	25
	37	find	65	find.c	21	1	22
Grep	46	grep	4	main.c	1361	1	1362
	47	grep	4	main.c	1110	1	1111
	48	grep	27	main.c	1364	1	1365

Therefore, we follow the suggestions of Xuan and Monperrus [34] and manually splitted such tests in order to have finer granularity and therefore more accuracy in the results.

Table 1 records information about the used subjects. In this table, the first column records the CoREBench ID for the bug. The column *Prog.* the name of the buggy program, the column *#Faulty SLOC* the number of executable lines of code changed in the bug-fixing commit[5]. The column *Source File* records the source file containing the definition of the C language *main* function of the corresponding buggy program. The columns *Pass* and *Fail* respectively show, for a subject, the number of tests that pass and fail on the buggy program due to the corresponding fault. The column *All* records the number of tests cases for each subject.

3.3 Mutation Operators

To perform the mutation analysis we used the mutation testing tool that was developed and used in the studies of Henard et al. [11]. The tool supports the following mutant operators: Arithmetic (AOR), Logical Connector Replacement (LCR), Relational (ROR), Unary Operator Mutation (UOM), Arithmetic assignment mutation (OAAA), Bitwise operator mutation (OBBN), Logical context negation (OCNG), Statement Deletion (SSDL) and Integer Constant replacement (CRCR).

3.4 Experiment Design

3.4.1 Procedure

For each bug B with corresponding faulty program P, the expriment procedure follows:

1. Construct line coverage matrix. The line coverage ma-

[5]Code insertion is represented by change in both the lines preceding and following the insertion point

trix has one row for each test case and one column for each line of code in the preprocessed source file. Entry (i, j) shows whether test i covered line j.

2. Generate mutants and remove duplicates. First order mutation is applied on the statements of the source file presented in Table 1 in order to generate mutants of the buggy program. Mutation testing tools tend to generate many duplicate mutants. Since these can be numerous, approximately 20% [25], they can artificially increasing the number of mutants and potentially influencing the fault localization results. Thus, we eliminated them using the TCE approach[25]. The number of mutants generated and remaining after duplicates removal for each subject is shown in Table 2 where the column *ID* shows the CoREBench ID for the bug. The column *Gen* shows the number of mutants generated by the mutation tool. The columns *Dup.* and *No-Dup.* respectively show the number of duplicate mutants and the number remaining after duplicates removal (non-duplicates). The columns *Live* shows the number of non-duplicates mutants equivalent to the original program with respect to the tests suite (dormant mutants). Finally, the column *Killed (MS)* shows the mutants whose test result differ from the original program for at least one test case (non-dormant mutants) and the respective mutation score. In the remaining part of this paper, we use the word mutants to refer non-dormant mutants.

3. Generate a result matrix. All the tests cases are both executed with the program P prior mutation and with each one of the mutants. The resulting matrix has one row for each mutant and one column for each test case. Entry (i, j) shows whether test i passed or failed on mutant j.

4. Execute both Metallaxis and MUSE. We implement the methods in Python programming language. In our implementation, the techniques use the data from the line coverage matrix, results matrix and the mutants information to generate a ranking of the SLOC according the suspiciousness of being faulty.

5. Gather line number of faulty SLOC. Regarding RQs 1-4, locations of each faulty line are obtained by manually reviewing the source code changes ("diff") from the bug-fix commit report accessible through CoREBench webpage. Note that the locations of the changed SLOC are obtained after source code pre-processing with a compiler, thus a single SLOC change in a bug-fixing commit may result in several faulty lines gathered. This may happen when the SLOC's changes appear in a C language macro. Changes where a new function definition is added are not considered, the reason being that such a change is not representative of the fault locations.

Several SLOCs may have the same suspiciousness value. Simply sorting the SLOCs by suspiciousness value would give different ranking for SLOC with same suspiciousness value. In this experiment, for such situation, we use an approach that is commonly used in literature [**?**, 14, 13]; were all SLOC having same suspiciousness value are ranked together at the upper of their ranks (*Upper-Rank*). We extend this with regard to the situation where more than one faulty SLOC are among the SLOC with same suspiciousness value v as following: all SLOC with suspiciousness value v are ranked at the rank they would have using *Upper-Rank* when only one faulty SLOC has suspiciousness value v. Altogether, suppose we have n SLOC having suspiciousness value v among which $k, 0 \leqslant k \leqslant n$ are faulty and $n - k$ non-faulty; the lower rank i and upper rank $j = i + n - 1$. All these n SLOC having suspiciousness value v are ranked at rank $r = j - k + 1$. The reason of this choice follow from the assumption that locating any faulty SLOC is equivalent to localize the fault, assumption made by some metrics used in this study.

Table 2: Mutants used.

ID	# of Mutants				
	Gen.	Dup.	No-Dup.	Live	Killed (MS)
1	1531	1203	328	177	151 (10%)
2	5785	2496	3289	1778	1511 (26%)
3	2749	1490	1259	963	296 (11%)
4	6413	2687	3726	1956	1770 (28%)
5	6399	2682	3717	1970	1747 (27%)
6	2692	1492	1200	368	832 (31%)
7	2530	1349	1181	355	826 (33%)
8	2364	1200	1164	379	785 (33%)
9	2322	1189	1133	390	743 (32%)
11	2512	1329	1183	372	811 (32%)
12	2659	1580	1079	388	691 (26%)
13	12262	4611	7651	4808	2843 (23%)
14	12238	4569	7669	4806	2863 (23%)
15	2435	1382	1053	518	535 (22%)
16	5472	2232	3240	709	2531 (46%)
17	2405	1262	1143	370	773 (32%)
18	1620	927	693	209	484 (30%)
19	1409	793	616	219	397 (28%)
20	1369	773	596	210	386 (28%)
21	2108	1003	1105	373	732 (35%)
22	1729	556	1173	602	571 (33%)
27	1428	594	834	531	303 (21%)
32	1362	567	795	556	239 (18%)
33	1353	563	790	550	240 (18%)
35	970	412	558	352	206 (21%)
36	2570	897	1673	1058	615 (24%)
37	2665	923	1742	1132	610 (23%)
46	5450	1767	3683	1752	1931 (35%)
47	5240	1755	3485	1894	1591 (30%)
48	5459	1770	3689	1741	1948 (36%)

3.4.2 *Metrics*

To assess the effectiveness of the studied fault localization techniques we adopt the following three metrics, which are common in literature, e.g., [15]:

- **Top N:** Measures the number of faults found by an FL technique when considering the *top-N* results in its ranking [30, 36, 15]. For a fault affecting several lines of code, if one of the faulty lines is found in the *top-N* results is considered localized.

- **Percentage Score (PS):** Measures the percentage of SLOC that are not checked by a programmer reviewing the SLOC in decreasing suspiciousness order until she reaches any faulty line. this "*score*" measure was proposed in the empirical evaluation of the tarantula automatic FL system by Jones et al. [14]. PS is calculated as:

$$PS = \frac{\text{\# of ranked SLOCs} - rank}{\text{\# of ranked SLOCs}},$$

were *rank* denote the the minimum of the ranks of all faulty SLOCs. The mean of PSs (MPS) across all bugs among the subjects is also used as metric in this study.

- **Mean Average precision (MAP):** Measures the mean of average precision for an FL technique with all the bugs considered in the study [15, 16]. An average precision is computed for each bug, as following [15]:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{\text{number of faulty lines}}$$

In the above formula, k is a rank in the returned ranked SLOCs, M is the number of ranked SLOCs, $pos(k)$ is a flag showing whether the k^{th} SLOC is faulty or not, respectively taking values 1 and 0. $P(k)$ is the precision considering a given *top-k* SLOC, and computed with the formula:

$$P(k) = \frac{\text{\# faulty SLOC in the top-k}}{k}$$

Note that when several SLOCs have the same suspiciousness value, we may have $P(k) > 1$.

In this experiment and regarding RQ1 we compare with the hypothetical "optimal" fault technique which is: for each fault, *all faulty lines are ranked with the highest suspiciousness.*

3.4.3 Utilized Tools

In the present study, we employed the mutation testing tool developed and used in the studies of Henard et al. [11]. The tool is built on top of the program matching and transformation framework called Coccinelle [22]. It operates at the source code level by matching pattern instances described in a dedicated language called semantic patch language. GNU Gcov[6] is used to generate the line coverage matrix. We encoutered some issues while using Gcov on our subjects: several bugs lead *Segmentation Fault* or *Infinite Loop* when executing regression tests, which cause the regression test execution to be "killed". Therefore, Gcov is not able to flush the coverage data to the disk. This scenario could create loss of information about the fault location in MUSE. In order to tackle such problem, we executed the regression test case with the project debugger GNU GDB[7]. Once the faulty program is killed, we manually call Gcov's *__gcov_flush()* function in GDB to flush the coverage data obtained before P was "killed".

4. QUANTITATIVE ANALYSIS

4.1 Practicality of Mutation-based FL

To answer RQ1 we assess the accuracy of Metallaxis and MUSE. We use the metrics presented in Section 3.4.2 to evaluate them with regard to the hypothetical "*optimal*" technique. Tables 3 and 4 record the performance results of Metallaxis, MUSE and that of the *optimal* technique in terms of the used metrics, i.e., Top N, MPS, MAP, AP and PS. Statistics regarding the faults that are hard and easy to localize are presented in Table 5.

[6]Gcov is a test coverage program part of GNU GCC
[7]https://www.gnu.org/software/gdb/

Table 3: Accuracy Assesment.

Metrics	Metallaxis	*Optimal*	MUSE
Top 1	3	30	4
Top 5	9	30	8
Top 10	15	30	14
Top 15	17	30	15
Top 20	19	30	16
Top 25	21	30	16
Top 30	24	30	18
Top 35	24	30	19
MPS	0.819	0.994	0.628
MAP	0.162	1.000	0.160

Table 3 presents a summary of the Top N metric value for each one of the FL technique. Each cell represents the value of the corresponding metric (row) for the corresponding technique (column). The results show that among all the faults, around 16% of them are ranked at the first position and 50% among the top 10 by both Metallaxis and MUSE. This shows the practicallity of both techniques to localize real faults, despite the high complexity of the studied faults, as demonstrated in the CoREBench authors [6]. Similarly, the MPS values are high and indicate the a developer following their faulty SLOC localization ranking would not need to check 82% of the code when using Metallaxis and 63% of the code when using MUSE. Both Metallaxis and MUSE have a very low MAP value, which again indicates that both techniques are relatively accurate.

Faults Easy and Hard to Localize. Table 5 shows for both Metallaxis (left side) and MUSE (right side), the subjects whose faults are easily localized (top end of the table) and those hard to localize (bottom end of the table). For each technique, the rank of the top ranked faulty SLOCs is shown. We consider a fault F easy to localize by a technique T when F is ranked by T among the top 10 and have high PS and AP values (relative to the subject analysed). Simillarily, F is hard to localize by T when F rank is greater than 10 and very low PS and AP values. Table 4 shows AP and PS values for each Fault. From both Tables 4 and 5, we observe that (1) Faults having both high AP ($\geqslant 0.1$) and high PS ($\geqslant 0.7$) are ranked to the high (top 10). And (2) fault having low AP ($\leqslant 0.99$) and high PS ($\geqslant 0.7$) are ranked after the top 10 but within the top 50. (3) The remaing have relatively low AP and PS.

We consider for this study the fauls of type (1) from the previous paragraph, as *easy to localize*, those of type (3) *hard to lolocalize*, and those of type (3) as *average*. Roughly 50% of the 30 faults are easily found, respectively 13% and 37% hard to localize by Metallaxis and MUSE.

4.2 Metallaxis vs MuSE

In this section, we present the results of our comparison in an attempt to answer RQ2.

4.2.1 Comparison using Different Metrics

Figure 4 visualises the results of the comparison regarding the Top N metric. From these data we can observe that when $N \leqslant 5$, MUSE localize 1 fault more than Metallaxis, when $4 \geqslant N \geqslant 11$ the two technique perform the same, although the $top - k$ localized faults are not same for both techniques as shown in Table 5. As N grows, the number of faults localized by Metallaxis increase faster than MUSE's,

Table 4: AP and PS values for the considered faults.

	ID	Prog.	AP		PS	
			Meta.	MUSE	Meta.	MUSE
l_1	1	rm	**1.000**	**1.000**	**0.984**	**0.984**
l_2	15	du	**1.000**	0.167	**0.995**	0.969
l_3	35	find	**1.000**	**1.000**	**0.984**	**0.984**
l_4	11	cut	0.312	**0.612**	0.991	**0.995**
l_5	6	cut	**0.200**	0.005	**0.977**	0.033
l_6	46	grep	**0.198**	0.119	**0.987**	0.981
l_7	22	expr	0.167	**0.500**	0.959	**0.986**
l_8	5	tail	**0.159**	0.037	**0.993**	0.987
l_9	7	seq	**0.156**	0.127	**0.971**	0.949
l_{10}	47	grep	**0.125**	**0.125**	**0.991**	**0.991**
l_{11}	27	find	0.075	**0.151**	0.867	**0.933**
l_{12}	14	ls	0.053	**0.091**	0.979	**0.988**
l_{13}	12	cut	**0.050**	0.030	**0.970**	0.955
l_{14}	9	seq	**0.050**	0.003	**0.939**	0.037
l_{15}	33	find	**0.050**	0.017	**0.865**	0.595
l_{16}	8	seq	**0.049**	0.004	**0.871**	0.064
l_{17}	19	seq	0.041	**0.111**	0.969	**0.990**
l_{18}	20	seq	0.037	**0.307**	0.870	**0.967**
l_{19}	18	seq	0.037	**0.333**	0.763	**0.974**
l_{20}	21	cut	**0.035**	0.005	**0.899**	0.040
l_{21}	36	find	**0.016**	0.008	**0.952**	0.899
l_{22}	3	cut	0.014	**0.020**	0.489	**0.707**
l_{23}	17	cut	**0.014**	0.003	**0.818**	0.039
l_{24}	48	grep	**0.012**	0.001	**0.952**	0.085
l_{25}	4	tail	**0.008**	0.002	**0.948**	0.199
l_{26}	32	find	0.007	**0.007**	0.000	**0.041**
l_{27}	2	od	**0.004**	0.003	**0.705**	0.363
l_{28}	16	tail	0.001	**0.001**	0.000	**0.214**
l_{29}	37	find	**0.001**	0.001	**0.874**	0.806
l_{30}	13	ls	0.000	**0.000**	0.000	**0.080**

Table 5: Faults Easy and Hard to localize.

	ID	Prog.	Meta.	ID	Prog.	MUSE
l_1	1	rm	1	1	rm	1
l_2	15	du	1	11	cut	1
l_3	35	find	1	19	seq	1
l_4	11	cut	2	35	find	1
l_5	19	seq	3	22	expr	2
l_6	5	tail	4	18	seq	3
l_7	47	grep	4	20	seq	3
l_8	6	cut	5	47	grep	4
l_9	7	seq	5	15	du	6
l_{10}	12	cut	6	27	find	6
l_{11}	22	expr	6	5	tail	7
l_{12}	46	grep	7	7	seq	9
l_{13}	36	find	9	12	cut	9
l_{14}	9	seq	10	46	grep	10
l_{15}	33	find	10	14	ls	11
l_{16}	20	seq	12	36	find	19
l_{17}	27	find	12	3	cut	27
l_{18}	14	ls	19	33	find	30
l_{19}	21	cut	20	37	find	34
l_{20}	8	seq	22	32	find	70
l_{21}	37	find	22	9	seq	157
l_{22}	48	grep	26	8	seq	160
l_{23}	18	seq	27	2	od	186
l_{24}	4	tail	28	21	cut	191
l_{25}	17	cut	37	17	cut	195
l_{26}	3	cut	47	6	cut	206
l_{27}	32	find	73	4	tail	431
l_{28}	2	od	86	16	tail	447
l_{29}	16	tail	569	48	grep	497
l_{30}	13	ls	905	13	ls	833

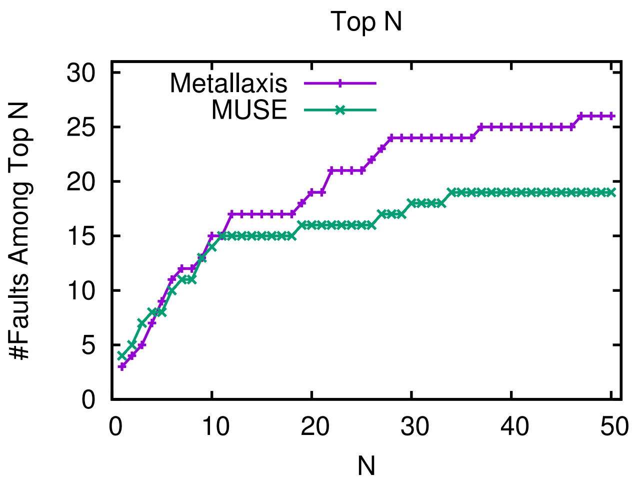

Figure 4: Top N, Metallaxis vs MUSE

showing that Metallaxis is more stable than MUSE. Furthermore, referring to Table 3, Metallaxis have both higher MAP and MPS values than MUSE.

4.2.2 Comparison Per-Fault Ranking Difference

Table 6 records a side by side rank that is given by each one of the considered faults and techniques. The table also records the difference between the rank given by Metallaxis and that of MUSE (rank difference). Over the 30 faults, 11 faults (*common*) are ranked in the top 10 both by Metallaxis and MUSE (middle section of Table 6: l_8 and $l_{10} \sim l_{19}$), 3 faults (*MUSE-Metallaxis*) ranked in the top 10 by MUSE and not Metallaxis (upper section of Table 6: l_3, l_5 and l_7), and finally 4 faults (*Metallaxis-MUSE*) are ranked in top 10 by Metallaxis and not MUSE (lower section of Table 6: l_{20}, l_{22}, l_{25} and l_{28}). The absolute difference in the rankings is not greater than 5 for all *common* faults, but greater than 5 for both *Metallaxis-MUSE* and *MUSE-Metallaxis*. In the next section, we investigate *common*, *MUSE-Metallaxis* and *Metallaxis-MUSE* faults.

5. QUALITATIVE ANALYSIS: RELEVANCE OF THE RANKED SLOCS

To further understand our results, we manually investigate the produced rankings with respect to the studied faults.

To answer RQ3, we analysed the relevance of the faulty SLOC with the top ranked statements. We investigate the first 10 SLOCs from each ranking according to the following procedure: (1) Manually locate each SLOC and the faulty SLOCs in the source code. (2) Guided by the faulty statements and dependancy analysis we assigned a relevance score () $\sim$ 100) to the SLOC. Such a relevance score depend on how fast a developer analysing the SLOC may end up to the faulty SLOC. We strictly considered dependencies and tried to quantify how easily a developer that starts from the provided statements and navigates through the dependencies can find the fault.

The results of the relevance analysis are recorded in Table 7. Note: Only faulty lines have a score of 100; a score of 0 do not necessarily mean that the SLOC is unrelated with

Table 6: Per-Fault Ranking Difference.

	ID	Prog.	Highest Fault Rank		
			Metallaxis	MUSE	Diff
l_1	16	tail	569	447	122
l_2	13	ls	905	833	72
l_3	18	seq	27	3	24
l_4	3	cut	47	27	20
l_5	20	seq	12	3	9
l_6	14	ls	19	11	8
l_7	27	find	12	6	6
l_8	22	expr	6	2	4
l_9	32	find	73	70	3
l_{10}	19	seq	3	1	2
l_{11}	11	cut	2	1	1
l_{12}	1	rm	1	1	0
l_{13}	35	find	1	1	0
l_{14}	47	grep	4	4	0
l_{15}	5	tail	4	7	-3
l_{16}	12	cut	6	9	-3
l_{17}	46	grep	7	10	-3
l_{18}	7	seq	5	9	-4
l_{19}	15	du	1	6	-5
l_{20}	36	find	9	19	-10
l_{21}	37	find	22	34	-12
l_{22}	33	find	10	30	-20
l_{23}	2	od	86	186	-100
l_{24}	8	seq	22	160	-138
l_{25}	9	seq	10	157	-147
l_{26}	17	cut	37	195	-158
l_{27}	21	cut	20	191	-171
l_{28}	6	cut	5	206	-201
l_{29}	4	tail	28	431	-403
l_{30}	48	grep	26	497	-471

the fault, but instead, means that it wouldn't quickly guide a developer to the faulty SLOCs according to our judgement. In situations where the fault is hard to localize and all SLOCs are given the same rank by the technique, we randomly choose to analyse 10 SLOCs.

We observe that at least one relevant SLOC can be ranked high by the techniques for 80% of the faults. Furthermore, The average relevance score of the top 10 results can be as high as $\sim 70/100$ for several faults. This results show that mutation-based FL techniques have the potential to help developer localize faults. They also point out a future direction of research that is to produce groups of statements that are related, as pointed out by Parnin and Orsro [29].

Finally, it is noted that the results presented here might be subjective (to some extend). Though, our goal was to give some hints regarding their relevance (with the studied faults) and not fully compare them. Furthermore, since we strictly considered dependencies all the statements with relevance higher than 0 are indeed related to the faults.

6. THREATS TO VALIDITY

Threats related to construct validity concern whether the experimental setup and metrics reflect real-world situation. In this study, a first thread to construct validity is due to the metrics used to measure that faults are localized. Following the recommendations made by literature [15], we employed three metrics, i.e., Top N, MAP and MPS. The second threat to validity regards the definition of *Faulty Line*. To reduce this threat, following existing studies [15], we used the executable lines of code changed after bug-fixing commit on the buggy programs. However, it is possible that a fault can be fixed by making changes many different locations. In order to further reduce this threat, more analysis is required on the faulty programs to gather more information about the root cause of the fault and therefore accurately evaluate the ranking of each fault localization technique. The major threats to internal validity are related to uncontrolled factors that may affect our results. In this study, the major threats to internal validity are the tools we used and our implementation of the FL-techniques, data collection and analysis scripts. For this matter we employ tool that have previously been used in the academic software testing community. We did our best to resolve all errors from the tools that we implemented. The major concern of external validity are whether the results of the present study can generalize to any other setting. The present study considered real faults selected by CoREBench's authors [6] from widely used programs, and the test case written by the programs' developers. The diversity and complexity of the faults as stated by CoREBench's authors is a leading factor contributing to the generalization of the present study on other C language programs. Whether this study is applicable on program written in other languages is unclear.

7. RELATED WORK

In literature there is extensive research on both fault localization and mutation analysis. Despite this, only recently researchers attempted to combine them. The present section provides a brief description of the most relevant fault localization 7.1, mutation-based fault localization 7.2 and mutation analysis methods 7.3.

7.1 Spectrum-based fault localization

Spectrum-based fault localization has been widely studied. Perhaps the most popular and known technique is the Tarantula [14]. It uses program statements and a similarity function that relates program statements with faulty executions. A widely known improvement of this technique was the use of Ochiai formula, which was demonstrated to be significantly more accurate [1]. Later fault localization methods used other spectra types like program branches and du-pairs [31]. All these approaches were unified by Santelices et al. [31] who also showed that there is no certain type of spectra that can be stated as the most effective. Other researchers attempted to use different suspicious formulas in order to improve effectiveness. Wong et al. [32] experimented with several heuristics and demonstrated that they could be better than Tarantula. Xie et al. [33] developed a theoretical framework and showed that some formulas are at least as good as others.

All the above techniques rely on coverage spectra which have been shown to be ineffective compared to mutation-based approaches. For instance, MUSE was shown to be 25 times more precise than statement-based fault localization. Papadakis and Le Traon [28] also showed that Metallaxis outperforms all the different spectra types, i.e., statements, branches and du-pairs.

7.2 Mutation-based fault localization

Research on mutation-based fault localization started by the work of Papadakis and Le Traon [27, 26] on Metallaxis. This work was then extended to reduce its application cost, using mutant-reduction techniques such as mutant sampling

Table 7: Relevance of the top 10 ranked statements. M indicates the average relevance.

	ID	Top 10 Lines Relevance Scores 0 ∼ 100										M	ID	Top 10 Lines Relevance Scores 0 ∼ 100										M
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**			**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	
Me	1	100	0	99	0	0	0	0	0	0	0	20	17	0	0	0	85	85	85	0	0	0	0	26
MU		100	95	99	99	0	0	0	0	0	55	45		0	0	0	0	0	0	0	0	0	0	0
Me	2	0	0	0	0	0	0	0	0	0	0	0	18	95	90	99	60	0	0	0	0	0	70	42
MU		0	0	0	0	0	0	0	0	0	0	0		90	99	100	95	30	90	70	70	99	50	80
Me	3	0	0	0	0	0	95	97	0	99	100	40	19	100	96	0	100	0	0	97	95	0	0	49
MU		0	0	0	0	5	99	0	5	99	5	22		100	100	0	0	0	95	20	0	20	2	34
Me	4	0	0	0	0	0	0	0	0	0	0	0	20	0	0	50	0	99	0	0	5	0	100	26
MU		0	0	0	0	0	0	0	0	0	0	0		0	0	100	100	100	100	100	99	100	99	80
Me	5	0	0	0	100	100	80	57	0	0	0	34	21	5	5	0	0	0	0	0	0	0	0	1
MU		0	0	0	0	99	98	100	0	0	0	30		100	80	75	60	60	20	15	0	0	0	41
Me	6	0	0	0	0	100	0	99	99	99	0	40	22	75	76	60	60	60	100	0	0	0	0	44
MU		0	0	0	0	0	0	0	0	0	0	0		0	100	0	5	0	0	0	0	0	0	11
Me	7	0	0	100	0	0	100	0	0	0	0	20	27	0	20	20	20	20	50	50	0	0	100	28
MU		0	0	0	0	0	100	0	0	99	0	20		100	20	20	20	20	0	100	50	50	70	45
Me	8	0	0	50	0	85	90	90	95	99	40	55	32	90	50	10	0	0	0	65	65	50	0	33
MU		99	0	50	85	90	85	90	95	90	0	69		0	65	0	0	50	0	50	50	0	65	28
Me	9	0	0	0	0	99	0	100	99	99	99	50	33	50	0	90	50	0	0	0	0	0	100	29
MU		0	95	0	99	0	95	95	99	0	0	49		0	50	0	0	0	0	0	0	90	0	14
Me	11	0	100	100	95	0	0	0	0	0	0	30	35	100	100	0	0	10	0	0	0	95	0	31
MU		100	100	0	0	0	0	100	0	0	0	30		100	100	0	15	0	35	0	90	95	97	54
Me	12	0	0	100	99	99	0	0	0	0	0	30	36	100	0	0	100	0	0	0	0	0	0	20
MU		0	0	96	0	0	0	99	0	100	99	40		3	0	90	0	0	3	0	0	0	0	10
Me	13	0	0	0	0	0	80	0	0	0	0	8	37	0	0	0	0	0	0	100	0	0	0	10
MU		0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0	10	1
Me	14	0	80	99	90	99	100	0	0	0	95	57	46	0	0	0	0	0	100	100	100	0	99	40
MU		0	80	90	90	100	0	0	0	0	0	36		0	0	0	0	0	0	0	0	0	100	10
Me	15	100	99	40	68	40	95	95	90	87	0	72	47	0	0	0	0	0	0	0	0	15	0	2
MU		40	99	95	0	0	100	0	0	0	0	34		0	0	0	0	0	0	0	0	0	0	0
Me	16	0	0	0	0	0	0	0	0	0	0	0	48	0	0	0	0	0	0	0	87	70	0	16
MU		0	0	0	0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0

and selective mutation [28]. The method was evaluated using the Proteum/FL tool [23] on hand-seeded faults and it was shown to be significantly more effective than the main spectrum-based fault localization techniques [28]. A follow up work and probably the most recent one that attempted to improve Metallaxis was MUSE [18]. However, as already explained MUSE was evaluated on hand-seeded faults and was not compared with Metallaxis. Additionally, since it observes whether a mutant is turns a test case from passing to failing and vice versa, it is limited to mutants fixing the bugs, while it completely ignores the killed mutants by failing test cases. These issues motivated the present paper, which compares the effectiveness of Metallaxis and MUSE on real-world settings.

Other mutation-based fault localization methods are due to Zhang et al. [35] who used mutants to locate faults on evolving programs. Thus, the work of Zhang et al. was the first that aimed at ranking suspicious program edits by combining the information provided by program changes and the behaviour of mutants. Another similar work of this types is due to Murtaza et al. [19] who used the impact of mutants on test execution in an attempt to produce traces that are similar to the (unknown) faulty ones. These traces can then be used to highly functions that are related to a given field failure. These two methods, although related, have a different goal than in our paper (Zhang et al. aims at program evolution and Murtaza et al. aims at field failures). In the present paper we aim at comparing and assessing the MUSE and Metallaxis on real-world settings given only the program with its passing and failing test cases.

7.3 Mutation Analysis

Mutation analysis was suggested as a way to solicit high quality test cases [21]. Then, researchers extended to support many software engineering tasks such as suggesting test oracles [9], reducing test suites [10], and debugging activities like fault localization [26] and bug fixing [7]. The idea has also been applied on several types of models, e.g., feature models [12], combinatorial interaction testing models [24] and behavioral models [8, 2].

Mutation is a popular technique mainly because it can support experimentation [4] and can subsume most of the other software testing techniques [3]. Recently, many robust and fast mutation testing tools have been built and integrated with software development tools. Thus, making it easy to use. One of the main problems of mutation is the equivalent mutants. Despite the recent success in automating this process [25], the problem remains. Luckily, equivalent mutants are not killed and thus, they do not pose any problems on fault localization.

8. CONCLUSION

This paper empirically investigates a relatively new direction of research, the mutation-based fault localization. The paper provides promising results indicating that by inspecting only 10 statements developers can successfully locate approximately 50% of the studied faults. The number of located faults increases to 80% for developers inspecting 25 statements. Overall, Metallaxis and MUSE can successfully locate the studied faults by respectively investigating the 18% and 37% of the executable statements of the programs under analysis. Therefore, indicating that Metallaxis significantly outperforms MUSE.

Another important finding of our results is the relation of the top-ranked statements with the faults under analysis. Our results show that at least one relevant statement exist in the top 10 statements for 80% of the studied faults. Furthermore, the average relevance of the top 10 statements of Metallaxis and MUSE is 28.16% and 25.88%, respectively. This is important since it ensures that developers will maintain their interest when using fault localization [29].

Overall, our results indicate that mutation-based fault localization achieves to successfully highlight faulty statements. It also points out several closely related, to the sought faults, statements providing evidence that the produced rankings can be of practical value. Also, our results indicate that Metallaxis produces twice more accurate results than MUSE.

9. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 89–98, 2007.

[2] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran. Killing strategies for model-based mutation testing. *Softw. Test., Verif. Reliab.*, 25(8):716–748, 2015.

[3] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[4] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.*, 32(8):608–624, 2006.

[5] R. Baker and I. Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2013.

[6] M. Böhme and A. Roychoudhury. Corebench: studying complexity of regression errors. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 105–115, 2014.

[7] V. Debroy and W. E. Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45–60, 2014.

[8] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens, and P. Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 655–666, 2016.

[9] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Trans. Software Eng.*, 38(2):278–292, 2012.

[10] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 361–372, 2014.

[11] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon. Comparing white-box and black-box test prioritization. In *ICSE*, 2016.

[12] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Assessing software product line testing via model-based mutation: An application to similarity testing. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013 Workshops Proceedings, Luxembourg, Luxembourg, March 18-22, 2013*, pages 188–197, 2013.

[13] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 167–178, New York, NY, USA, 2008. ACM.

[14] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[15] T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 579–590, New York, NY, USA, 2015. ACM.

[16] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[17] W. Masri and R. A. Assi. Prevalence of coincidental correctness and mitigation of its impact on fault localization. *ACM Trans. Softw. Eng. Methodol.*, 23(1):8:1–8:28, 2014.

[18] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, pages 153–162, Washington, DC, USA, 2014. IEEE Computer Society.

[19] S. S. Murtaza, A. Hamou-Lhadj, N. H. Madhavji, and M. Gittens. An empirical study on the use of mutant traces for diagnosis of faults in deployed systems. *Journal of Systems and Software*, 90:29–44, 2014.

[20] A. J. Offutt and J. H. Hayes. A semantic model of program faults. In *ISSTA*, pages 195–200, 1996.

[21] J. Offutt. A mutation carol: Past, present and future. *Information & Software Technology*, 53(10):1098–1107, 2011.

[22] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 247–260, 2008.

[23] M. Papadakis, M. E. Delamaro, and Y. L. Traon. Proteum/fl: A tool for localizing faults using mutation analysis. In *13th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2013, Eindhoven, Netherlands, September 22-23, 2013*, pages 94–99, 2013.

[24] M. Papadakis, C. Henard, and Y. L. Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 1–10, 2014.

[25] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 936–946, Piscataway, NJ, USA, 2015. IEEE Press.

[26] M. Papadakis and Y. Le Traon. Metallaxis-fl: Mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5-7):605–628, Aug. 2015.

[27] M. Papadakis and Y. L. Traon. Using mutants to locate "unknown" faults. In *Fifth IEEE International Conference on Software Testing, Verification and*

Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012, pages 691–700, 2012.

[28] M. Papadakis and Y. L. Traon. Effective fault localization via mutation analysis: a selective mutation approach. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pages 1293–1300, 2014.

[29] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 199–209, 2011.

[30] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013.

[31] R. A. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 56–66, 2009.

[32] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Trans. Reliability*, 63(1):290–308, 2014.

[33] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31, 2013.

[34] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 52–63, 2014.

[35] L. Zhang, L. Zhang, and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 765–784, 2013.

[36] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.