CS453 - Automated Software Testing

Spring 2023

Analysis on Mutation Based Fault Localization

Team 5

Project Information:

Project GitHub Repository: https://github.com/topcue/mbfl-muse

Account information of each team member:

- **Hyungjoon Yoon** is associated with the github id ***topcue***
- **Sanghyun Park** is associated with the github id ***hy38***
- **Banseok Woo** is associated with the github id ***BanseokWoo***
- **Hyeonseok Lee** is associated with the github id ***hslee1225***
- **Somin Kim** is associated with the github id ***thals1214***

Contents

1. Introduction

In the area of debugging and fault localization, there are several techniques available, such as SBFL (Spectrum-Based Fault Localization) [1] and MBFL (Mutation-Based Fault Localization) [2], each with its own strengths and limitations. Our project aims to overcome these limitations by trying to combine the key features of SBFL and MBFL, and thereby optimizing the process of fault localization and potentially improving overall performance.

To achieve this, we conduct a thorough analysis of SBFL(i.e., Ochiai [3], Jaccard [4], and Op2 [5]) and MBFL (i.e., MUSE [2]) techniques in both simple C hand-crafted examples and Java projects using Defects4J [6]. By exploring these techniques in different programming languages, and real life scenarios, we aim to gain a comprehensive understanding of their effectiveness and applicability in various software testing domains.

This document will first detail the identified problem, and discuss how it may be improved. We then propose our methodology, and then evaluate based on our approach. The report concludes with our observations and possible improvements.

2. Related Work

Fault Localization

Fault localization is the process of identifying the specific location in a software that is responsible for a fault or bug. This process aids programmers to fix the bug by only carefully looking at the suspicious lines of code thus relieving the burden of debugging. There are various methods and techniques used for fault localization, such as IRFL [7], SBFL, MBFL. Also, delta debugging [8] similarly behaves to achieve the goal of fault localization.

Mutation Testing

Mutation testing is a method that modifies a program's source code in small ways in order to test the quality of test suites. It is used to determine whether or not the test cases are sufficient to detect the small changes made in the software. The mutants are generated by altering the specific points of a program, creating copies of the original code but with slight modifications (e.g., changing arithmetic operator, altering a logical condition). Once these mutants are generated, the test suite is run against each of them. If the tests fail, the mutant is said to be "killed", meaning that the test suite was effective in detecting the simulated fault. If the tests pass, it means the test suite could not detect the mutation, and the mutant is said to have "survived".

Mutation-based Fault Localization(MBFL)

MBFL, as a method, is based on the concept of mutation testing. It inserts small faults or 'mutations' into a software program's source code and runs test cases on these modified programs, known as 'mutants.' However, in the context of fault localization, MBFL doesn't aim to evaluate the quality of the test suite, but rather uses these mutants and the results of the test execution to provide information about the likely location of faults in the program.

Spectrum-based Fault Localization(SBFL)

SBFL, on the other hand, is a technique that leverages program spectra, an execution profile of program runs, to locate faults in the software. The SBFL technique does not require knowledge of the program's intended behavior or its specifications.

Instead, SBFL is based on the idea that if a particular statement in the code is frequently executed during failing tests, and less frequently during successful tests, then it's likely that the statement is faulty. Therefore, SBFL ranks program statements based on their suspiciousness level, which is calculated from their execution pattern in successful and failing tests. Techniques such as Tarantula, Ochiai, and Jaccard, among others, provide different formulas for calculating this suspiciousness level.

3. Methodology

3.1 Reproduction of MBFL and SBFL results

In our approach, we primarily focused on reproducing Mutation-Based Fault Localization (MBFL) and Spectrum-Based Fault Localization (SBFL) results on basic target C files. This process consisted of multiple stages, each having its specific purpose and role in achieving our objective.

In the initial stage, our goal was to mutate the target C files systematically. To accomplish this task, we employed MULL [9], a mutation testing tool optimized for C and C++ languages. MULL's capabilities allowed us to systematically introduce minor faults, or "mutations," into the target code. These mutations were in the form of slight changes to the program’s source code that ideally wouldn't affect its overall functionality but could potentially uncover underlying faults when subject to test cases. Using MULL streamlined this phase, ensuring we could generate a comprehensive set of mutants from the original C files quickly and efficiently.

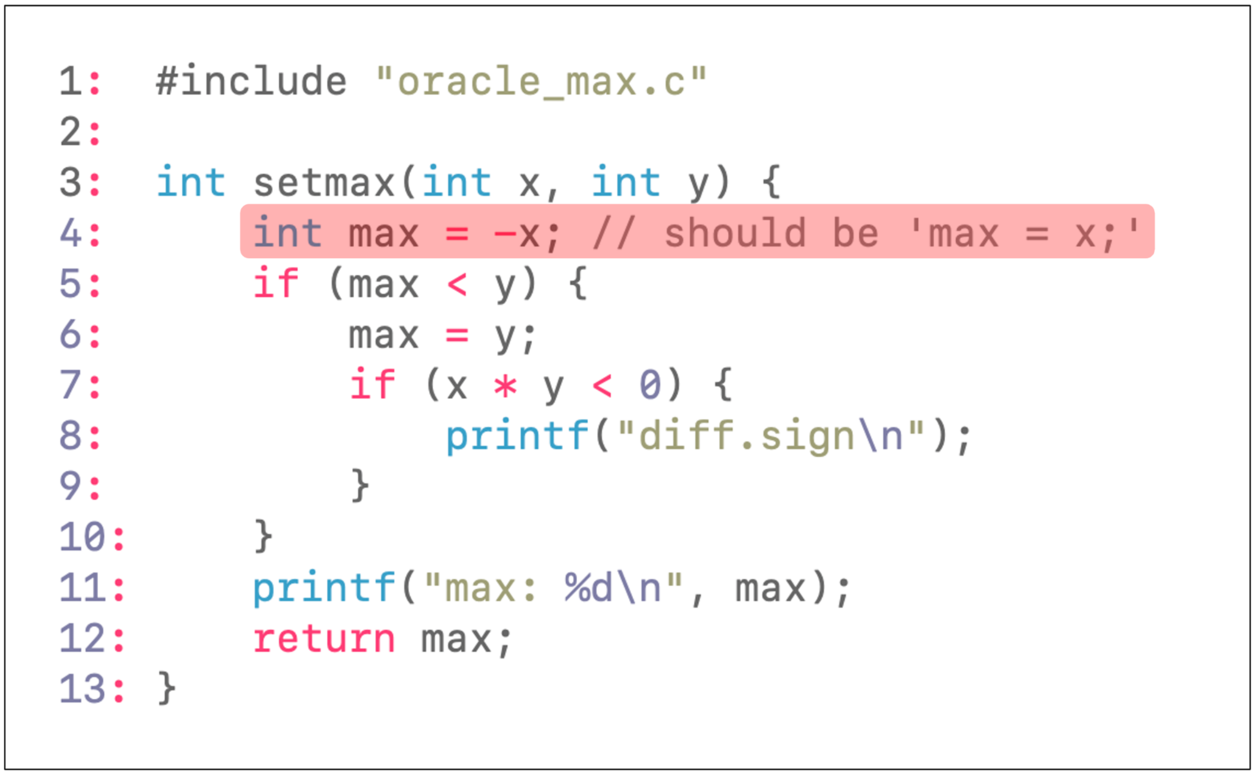
Once we had a range of mutated C files, the next stage of our methodology involved the use of a specialized Python script. This script served two crucial roles in our project. First, it utilized gcov, a test coverage program, to track the code coverage of our original, non-mutated C file. By doing so, we obtained a comprehensive view of how much of the original code was exercised by the test cases. This was particularly essential when applying the SBFL, where the execution profile of program runs is key in identifying potential faults.

The second role of our Python script was to monitor and record the pass/fail results for each mutant. When the test suite was run against each mutant, the script would log whether the tests passed or failed. From these results, we calculated the MBFL and SBFL scores for each line of code, leveraging the pass/fail results from the mutants and the code coverage information.

Through this two-step process, we managed to reproduce the MBFL and SBFL results on our basic target C files. The combination of MULL's mutation capabilities and our custom Python script's tracking and recording functions proved to be an effective method for investigating and analyzing fault localization.

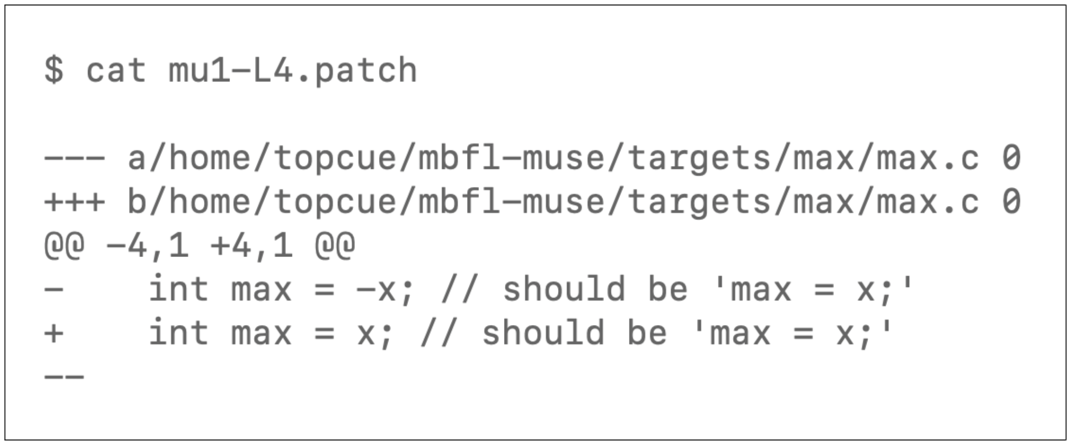
3.1.1 Case Study : max.c

The following is a case study for max.c file, which is one of our target c files, to better explain our implementation for MBFL analysis on C files.



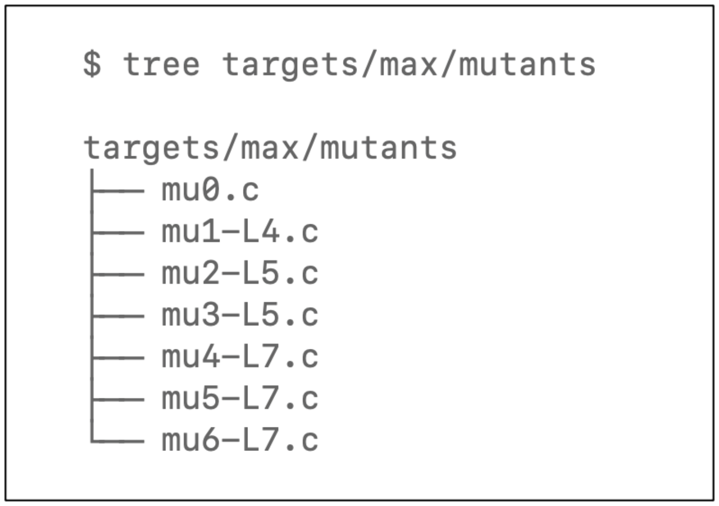
```
1:  #include "oracle_max.c"
2:
3:  int setmax(int x, int y) {
4:      int max = -x; // should be 'max = x;'
5:      if (max < y) {
6:          max = y;
7:          if (x * y < 0) {
8:              printf("diff.sign\n");
9:          }
10:     }
11:     printf("max: %d\n", max);
12:     return max;
13: }
```

Above is a max.c file that is the target for applying MBFL to. Line 4 is buggy, since it should be “max = x;”.



```
$ cat mu1-L4.patch

--- a/home/topcue/mbfl-muse/targets/max/max.c 0
+++ b/home/topcue/mbfl-muse/targets/max/max.c 0
@@ -4,1 +4,1 @@
-    int max = -x; // should be 'max = x;'
+    int max = x; // should be 'max = x;'
--
```

Here is an example patch that our implementation created. It is the fixing mutant that we are looking for. “mu1” means its mutant id is 1, “L4” means it is a mutant for line4.

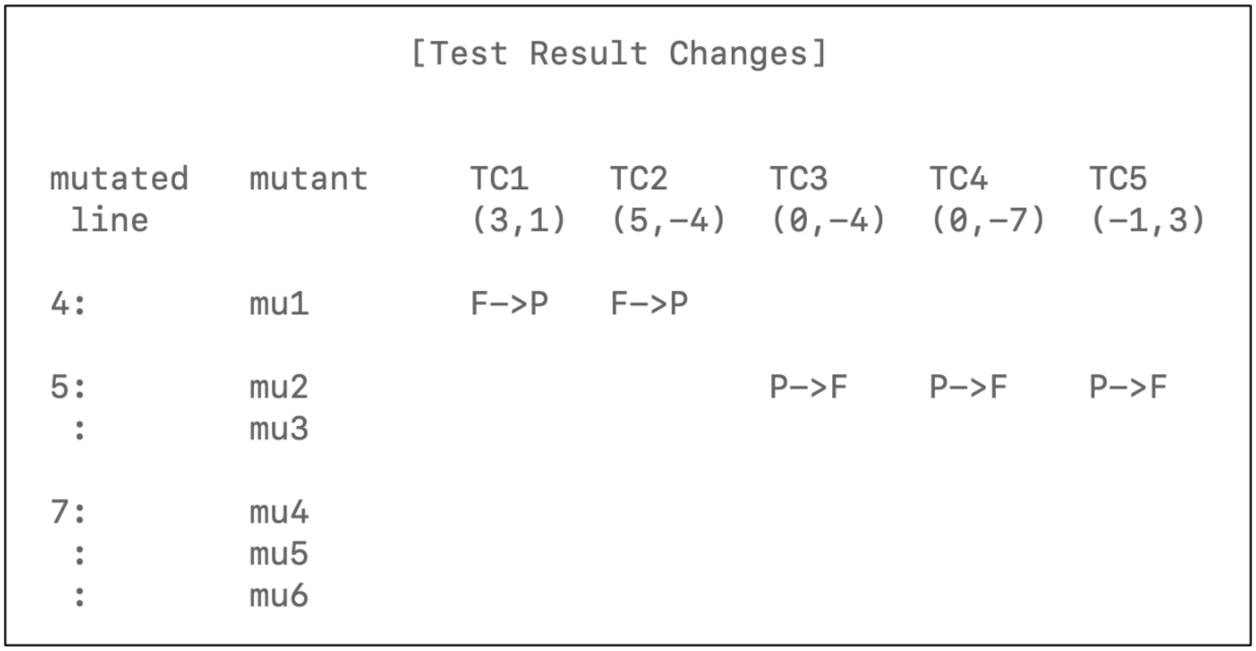


```
$ tree targets/max/mutants

targets/max/mutants
├── mu0.c
├── mu1-L4.c
├── mu2-L5.c
├── mu3-L5.c
├── mu4-L7.c
├── mu5-L7.c
└── mu6-L7.c
```

Here, all the created mutants are listed for max.c file.

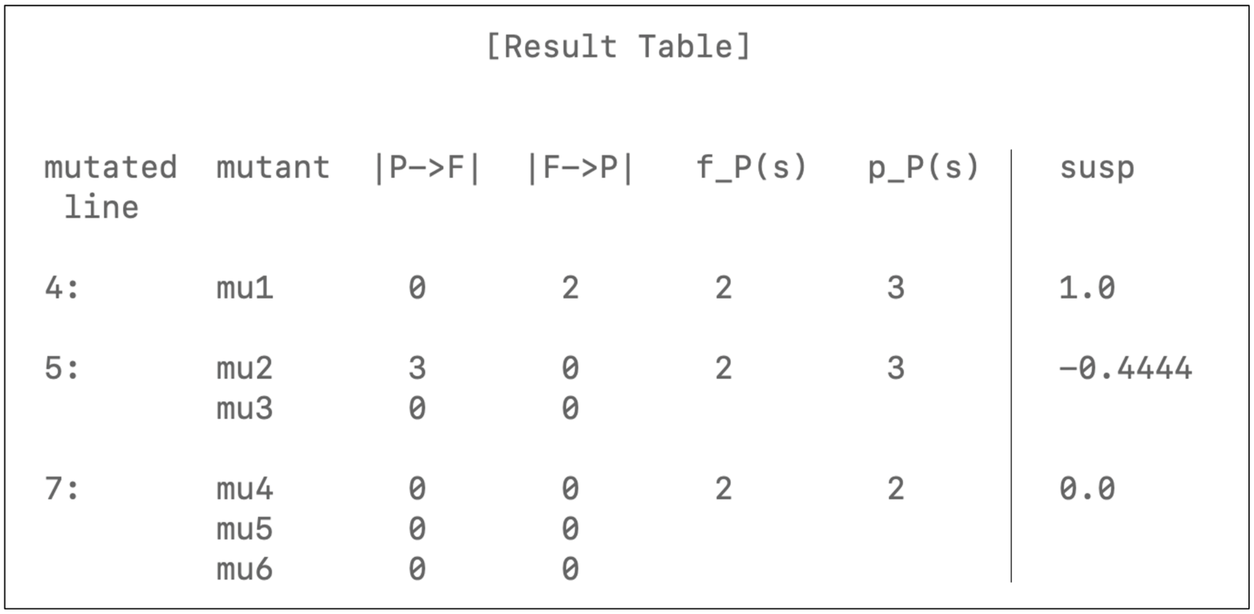
max.c	mu4-L7.c	mu5-L7.c	mu6-L7.c
`if (x * y < 0) {`	`if (x * y >= 0) {`	`if (x * y <= 0) {`	`if (x / y < 0) {`

As shown here, even one single line, (line 7 in this case) can be mutated in different ways.



```
                    [Test Result Changes]


mutated    mutant      TC1     TC2     TC3     TC4     TC5
 line                  (3,1)   (5,-4)  (0,-4)  (0,-7)  (-1,3)

4:         mu1         F->P    F->P

5:         mu2                         P->F    P->F    P->F
 :         mu3

7:         mu4
 :         mu5
 :         mu6
```

Here is shown how each mutant on the target file changes each test result(F->P or P->F). Silent cases(F->F or P->P) are not printed here.



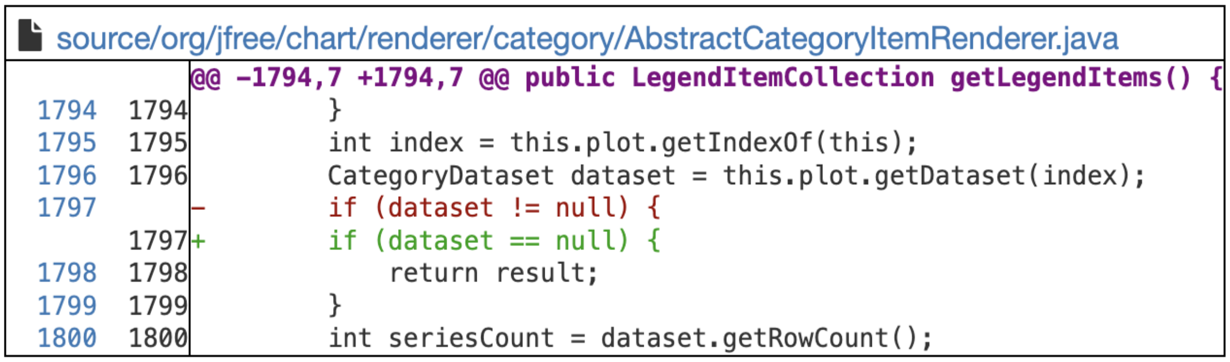
```
                        [Result Table]


mutated   mutant   |P->F|   |F->P|   f_P(s)   p_P(s)  |  susp
 line                                                 |
                                                      |
4:        mu1        0        2        2        3     |  1.0
                                                      |
5:        mu2        3        0        2        3     |  -0.4444
          mu3        0        0                       |
                                                      |
7:        mu4        0        0        2        2     |  0.0
          mu5        0        0                       |
          mu6        0        0                       |
```

Here, each line's suspiciousness scores are calculated using the following formula from MUSE paper. We can see that line 4, which is in fact the buggy line, is correctly chosen as most suspicious.
(|P->F|, |F->P|, f_P(s), p_P(s) mean pass to fail count, fail to pass count, number of tests that cover given line and fail and number of tests that cover given line and pass, respectively)
(Details can be found in the MUSE paper)

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

3.2 MBFL and SBFL analysis with Defects4J mutation testing

Having achieved a successful implementation of both MBFL and SBFL methodologies on our basic target C files, we explored the application of these techniques on real-world defective Java programs. For this, we leveraged the Defects4J dataset [6], a rich resource offering real-world bugs extracted from open-source projects, with each bug's faulty and fixed versions available. A key example that became the focus of our study is the ”Chart-1” data, which is a buggy version of “Chart” project accessible via the Defects4J command-line interface.



source/org/jfree/chart/renderer/category/AbstractCategoryItemRenderer.java

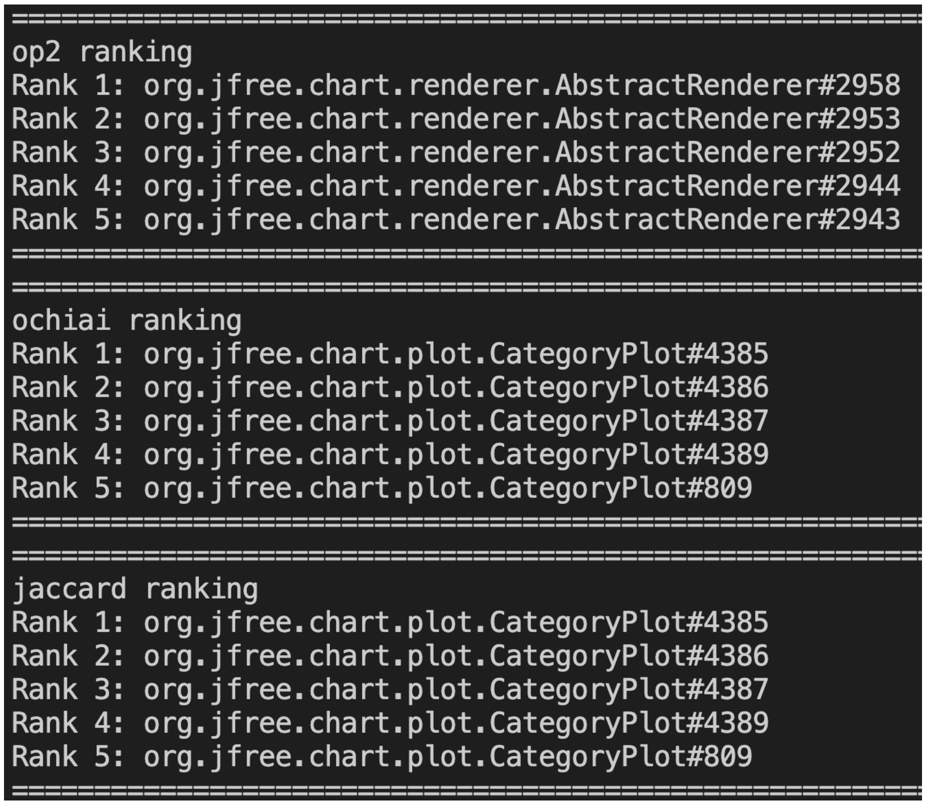
```
@@ -1794,7 +1794,7 @@ public LegendItemCollection getLegendItems() {
         }
         int index = this.plot.getIndexOf(this);
         CategoryDataset dataset = this.plot.getDataset(index);
-        if (dataset != null) {
+        if (dataset == null) {
             return result;
         }
         int seriesCount = dataset.getRowCount();
```

Figure : The developer patch for Chart-1

The reason we chose Chart 1b as the first target for our experiment is that, as can be seen in the figure, fixing the bugs seemed like it could be caught with a simple fix by the mutation operator (ROR). Therefore, we anticipated that we could obtain meaningful results from the Mutation Based Fault Localization Experiment. This was derived from the observation that MUSE gives weight to the mutant as the most suspicious when a partial fix (changes the failing test to a passing test) occurs.

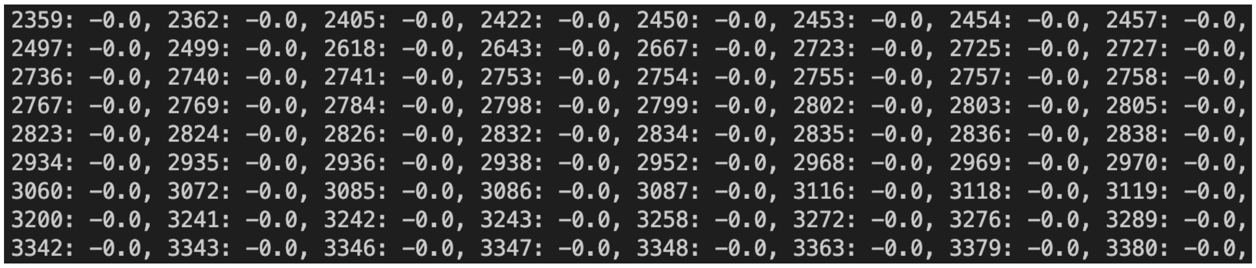
We performed MBFL and SBFL analysis on target Java programs, by examining each program's coverage matrix and kill-matrix.

From the coverage matrix, we calculated the SBFL scores for every statement of code.



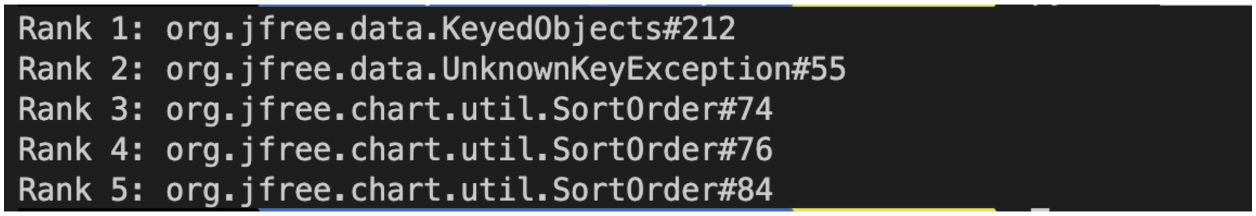
```
===========================================================
op2 ranking
Rank 1: org.jfree.chart.renderer.AbstractRenderer#2958
Rank 2: org.jfree.chart.renderer.AbstractRenderer#2953
Rank 3: org.jfree.chart.renderer.AbstractRenderer#2952
Rank 4: org.jfree.chart.renderer.AbstractRenderer#2944
Rank 5: org.jfree.chart.renderer.AbstractRenderer#2943
===========================================================
===========================================================
ochiai ranking
Rank 1: org.jfree.chart.plot.CategoryPlot#4385
Rank 2: org.jfree.chart.plot.CategoryPlot#4386
Rank 3: org.jfree.chart.plot.CategoryPlot#4387
Rank 4: org.jfree.chart.plot.CategoryPlot#4389
Rank 5: org.jfree.chart.plot.CategoryPlot#809
===========================================================
===========================================================
jaccard ranking
Rank 1: org.jfree.chart.plot.CategoryPlot#4385
Rank 2: org.jfree.chart.plot.CategoryPlot#4386
Rank 3: org.jfree.chart.plot.CategoryPlot#4387
Rank 4: org.jfree.chart.plot.CategoryPlot#4389
Rank 5: org.jfree.chart.plot.CategoryPlot#809
===========================================================
```

Concurrently, we analyzed the kill-matrix, obtained through mutation testing on the Java programs. As in our preliminary methodology with C files, this matrix was generated by noting which test cases 'killed' the mutants - the program versions with introduced minor alterations or "mutations". By exploiting both coverage matrix and the kill-matrix, we calculated the MBFL scores, providing a more nuanced interpretation of potential fault locations within the program.



```
2359: -0.0, 2362: -0.0, 2405: -0.0, 2422: -0.0, 2450: -0.0, 2453: -0.0, 2454: -0.0, 2457: -0.0,
2497: -0.0, 2499: -0.0, 2618: -0.0, 2643: -0.0, 2667: -0.0, 2723: -0.0, 2725: -0.0, 2727: -0.0,
2736: -0.0, 2740: -0.0, 2741: -0.0, 2753: -0.0, 2754: -0.0, 2755: -0.0, 2757: -0.0, 2758: -0.0,
2767: -0.0, 2769: -0.0, 2784: -0.0, 2798: -0.0, 2799: -0.0, 2802: -0.0, 2803: -0.0, 2805: -0.0,
2823: -0.0, 2824: -0.0, 2826: -0.0, 2832: -0.0, 2834: -0.0, 2835: -0.0, 2836: -0.0, 2838: -0.0,
2934: -0.0, 2935: -0.0, 2936: -0.0, 2938: -0.0, 2952: -0.0, 2968: -0.0, 2969: -0.0, 2970: -0.0,
3060: -0.0, 3072: -0.0, 3085: -0.0, 3086: -0.0, 3087: -0.0, 3116: -0.0, 3118: -0.0, 3119: -0.0,
3200: -0.0, 3241: -0.0, 3242: -0.0, 3243: -0.0, 3258: -0.0, 3272: -0.0, 3276: -0.0, 3289: -0.0,
3342: -0.0, 3343: -0.0, 3346: -0.0, 3347: -0.0, 3348: -0.0, 3363: -0.0, 3379: -0.0, 3380: -0.0,
```

Figure: In most cases, the suspicious score based on MUSE resulted in '0'.



```
Rank 1: org.jfree.data.KeyedObjects#212
Rank 2: org.jfree.data.UnknownKeyException#55
Rank 3: org.jfree.chart.util.SortOrder#74
Rank 4: org.jfree.chart.util.SortOrder#76
Rank 5: org.jfree.chart.util.SortOrder#84
```

Figure: Top 5 ranked statement based on the suspicious score of MUSE

MUSE improves the accuracy of fault localization when (Partial)Fix occurs, but it is not as effective if no (Partial)Fix occurs. Therefore, as a result, incorrect statements shown above were included in the top 5, and most statements had a same suspicious score.

By applying these techniques to real-world Java programs, we aimed to evaluate their effectiveness and performance in a complex and practical setting. Our approach provided valuable insights into the applicability of SBFL and MBFL techniques for software testing in realistic scenarios, going beyond the simpler C files we initially studied.

4. Evaluation

4.1 Environmental Setup

4.1.1 Target C Programs

There are a total of 6 target c programs we ran MBFL analysis on.

File Name	Description
`max.c`	Program that, given two integer inputs, outputs the maximum value of those.
`getQuotient.c`	Program that, given two integer inputs, outputs the quotient of those
`sha256.c`	Hash function for the SHA256 algorithm
`quicksort.c`	Program that performs a quicksort on a given integer array
`stack.c`	Program that implements a stack data structure with functions to create, check, push, and pop
`bfs.c`	Program that, given a tree structure, performs a breadth-first search and outputs visited nodes

4.2.2 Target Java Program

We also tested SBFL and MBFL on the Chart-1 of the Defects4J Dataset. We analyzed the results driven from developer’s tests through coverage and mutation testing. We created a kill matrix for each statement from analyzing mutants.log and the coverage matrix.

4.2 Results

4.2.1 Target C Programs

PUT	num of mutants	buggy line	suspiciousness rank top3			detect bug	false alarm
			1st	2nd	3rd		
max	6	4	**4**	5	7	O	X
getQuotient	3	8	**8**	7	-	O	X
sha256	62	110	**110**	-	109, 111	O	X
quicksort	12	21	13	14	11	X	O
stack	5	29	-	-	-	X	X
bfs	20	58	-	-	-	X	X

Above table presents the results on each target C file. “Num or mutants” and “buggy line” mean how many mutants are generated and which line is the intended buggy line.

Then, top 3 candidates according to MBFL suspiciousness score are listed. detect bug” checks whether MBFL could correctly rank the buggy line as the top candidate, and “false alarm” means MBFL successfully finished but its prediction is wrong.

Based on this result, we can categorize each case into the following three categories.

- **`max, getQuotient, sha256`** : These programs are the ones that our MBFL analysis was able to correctly locate the faulty line
- **`quicksort`** : For this program, MBFL analysis worked but it created a false alarm(MBFL ranks wrong line as most suspicious)
- **`stack, bfs`** : For these programs, MULL did not create appropriate mutant (the mutant that would fix the buggy code back into our intended correct code), which led to lack of fixing mutants(F->P, mutants that turn Fail to Pass). Therefore, our MBFL

analysis failed calculation, but we expect it to correctly locate the faulty region, provided appropriate mutant is created.

4.2.2 Target Java Programs

Method	Top1	Top2	Top3
MUSE	org.jfree.data.KeyedObjects	org.jfree.data.UnknownKeyException	org.jfree.chart.util.SortOrder
Op2	org.jfree.chart.renderer.AbstractRenderer	org.jfree.chart.renderer.AbstractRenderer	org.jfree.chart.renderer.AbstractRenderer
Ochiai	org.jfree.chart.plot.CategoryPlot	org.jfree.chart.plot.CategoryPlot	org.jfree.chart.plot.CategoryPlot
Jaccard	org.jfree.chart.plot.CategoryPlot	org.jfree.chart.plot.CategoryPlot	org.jfree.chart.plot.CategoryPlot
Bug	AbstractCategoryItemRenderer.getLegendItems		

4.3 Observation

Overall, we can see consistency between the result on our implementation of MBFL for C files and the result of MBFL analysis done on the Defects4J dataset. In both cases, MBFL struggled due to lack of fixing mutants(F -> P). The reasons for this behavior for each cases are the following:

- **MUSE for C** :

First of all, MULL doesn't support all the variants for mutating statements so that some of our expected patches were not at all supported.

Also, even if some mutation operation is supported, MULL does not create all possible mutants for all possible lines, which made it possible for fixing mutants not to be created.

- **MUSE for Java** :

We suspect that there were not sufficiently enough test cases in which the original test case failed, but its mutant passed; To be more specific, as seen from above, SBFL for Chart-1 sometimes includes the faulty statements within the top 5 according to the formula used. However, fault localization through MUSE ranked those lines at very low positions. This is because there were no appropriate mutants generated for the bug line in Chart-1, resulting in no fix occurring. Therefore, it is plausible to infer that the fixing mutants are not created as in "MBFL for C" case, which explains the consistency between two results.

Overall, the low performance that's observed in some cases throughout our research, is not proven to be a fundamental flaw of MBFL, but rather it seems that lack of appropriate mutants is the key reason for that.

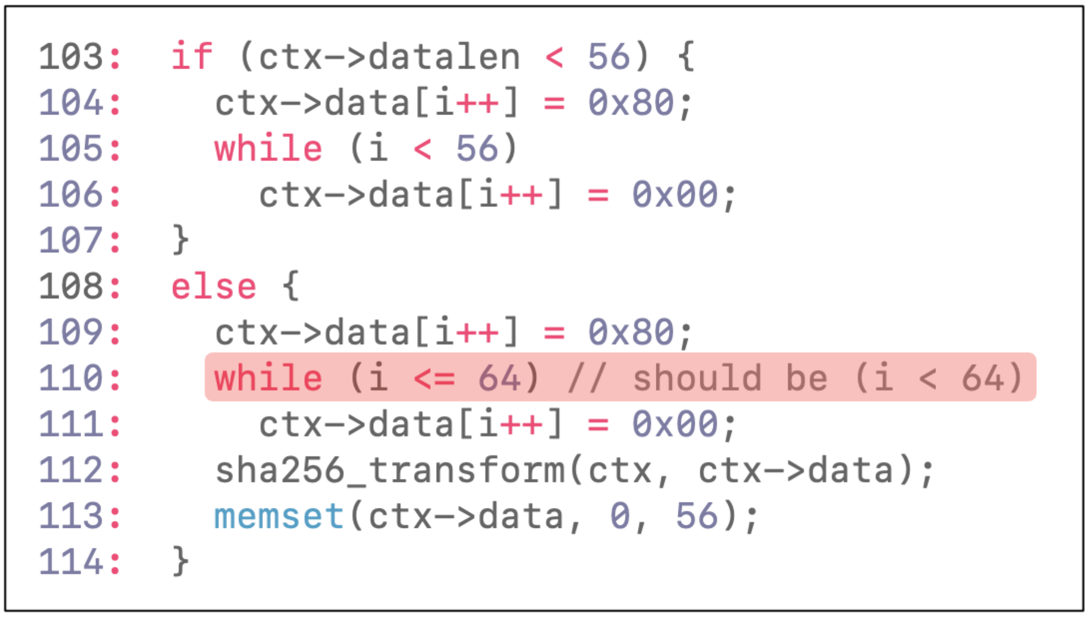
But due to lack of resources, it is not possible to mutate in every possible way, in any case. And in real world faults, it is not possible to know which mutation would fix a bug.

However, when applying MUSE to Java, it is not entirely meaningless, considering that it can mostly eliminate suspiciousness for lines that change the test results from Pass to Fail.

4.3.1 Case Study : modified_sha256.c

Additionally, the following is an example in which although the buggy line has many mutants, the bug is correctly located only when the fixing mutant (one that does F->P) is present.

The following code snippet is a sha256.c file from our setup, that is intentionally modified (mutated manually) for our case study.



```
if (ctx->datalen < 56) {
  ctx->data[i++] = 0x80;
  while (i < 56)
    ctx->data[i++] = 0x00;
}
else {
  ctx->data[i++] = 0x80;
  while (i <= 64) // should be (i < 64)
    ctx->data[i++] = 0x00;
  sha256_transform(ctx, ctx->data);
  memset(ctx->data, 0, 56);
}
```

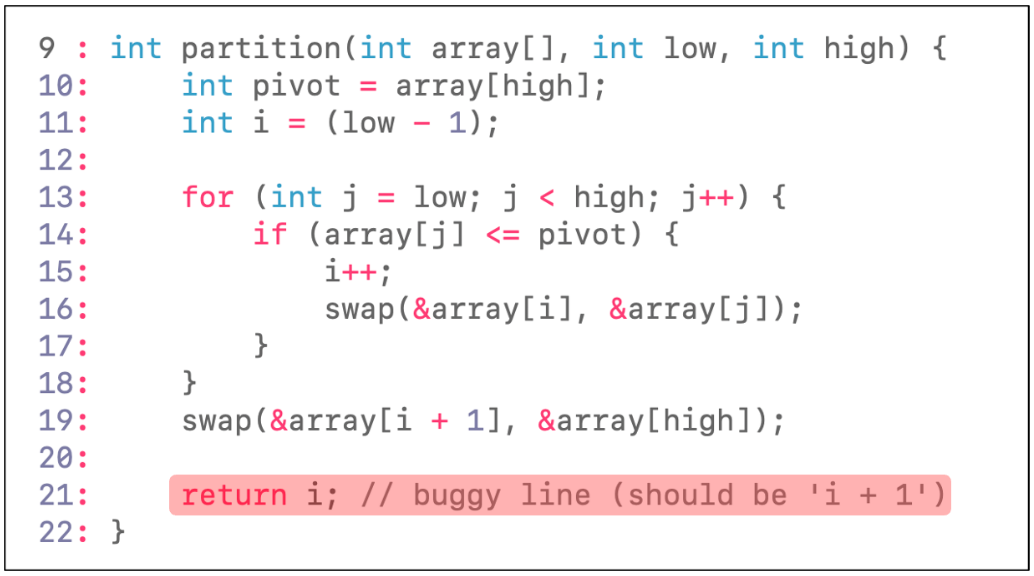
Figure: modified_sha256.c

Considerable mutants are as follows:

case	mutation operator	operator semantics	detect bug with high score
(A)	cxx_le_to_lt	Replaces <= with <	O
(B)	cxx_le_to_gt	Replaces <= with >	X
(C)	cxx_le_to_ge	Replaces <= with >=	X
(D)	scalar_value_mutator	Replaces 64 with 0	X
(E)	scalar_value_mutator	Replaces 64 with 63	O

Our MUSE generated mutants equivalent to the case of (A) detecting the bug with a high score. Yet, when making mutants such as case (B), (C), (D), the mutated cases fail to detect the bug. Though it would succeed detecting the bug when manually executing the mutation operator such as (E), mutating a constant of 64 to a constant of 63 is unexpectable in real-world mutation scenarios.

On the other hand, there exists a case that our MUSE basically fails to detect the bug but succeeds with manual mutation with a high score. The following source code snippet of quicksort.c shows the case when our MUSE failed to detect the bug and had a false alarm.



```c
int partition(int array[], int low, int high) {
    int pivot = array[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (array[j] <= pivot) {
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[high]);

    return i; // buggy line (should be 'i + 1')
}
```

Our MUSE lacks a mutation operator that patches the buggy line. Instead of mutating the buggy line, other mutants coincidentally derived F->P, thus involving the mutant to claim an irrelevant line to be suspicious.

Upon manually adding mutation operators and analyzing the results, it appears that the most crucial factor determining the bug detection performance of MUSE is the existence of a mutation operator that mutates the buggy line and induces a F->P transition.

The quantity and quality of test cases, particularly the proportion of test cases inducing a F->P or P->F transition and the coverage of each test case, affect the suspiciousness value. However, they do not determine the presence or absence of bug detection or false alarms.

4.4 Possible Improvements

Therefore, we suggest that it would be more beneficial to reduce search space of suspicious lines, using SBFL techniques, and then applying more variants to those reduced candidate lines. This way, it would increase the chance of creating a fixing mutant, by focusing on those who are the most suspicious candidates, according to SBFL.

Also, further optimization is possible for this method by introducing weights between “shrinkage of search space by SBFL” and “focus on top candidates more for mutation” based on some training real-world datasets, to find optimal ratio between these two factors. (For example, we could compare against 50 candidates, 2 mutants each vs. 10 candidates, 5 mutants each, to see which strategy is more likely to correctly locate the bug)

Additionally, we can conduct a semantic based search for finding a suitable mutant related to the buggy line, and find the most relevant/suitable mutants to apply there to enhance the probability of the mutant ‘fixing’ the program.

5. Conclusion

In this report, we present the implementation of two innately different fault localization techniques, namely SBFL and MBFL, in two distinct language domains, C and Java. This document presents a comprehensive methodology of our execution, followed by an analytical review of the outcomes derived from our experimental evaluations. Furthermore, we provide an in-depth exploration of our implementation challenges with the MUSE method, alongside the complexities faced while applying MBFL.

6. References

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

[2] Moon, Seokhyeon, et al. "Ask the mutants: Mutating faulty programs for fault localization." 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE, 2014.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In PRDC 2006, pages 39–46, 2006.

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

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

[6] Just, René, Darioush Jalali, and Michael D. Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs." Proceedings of the 2014 international symposium on software testing and analysis. 2014.

[7] Lukins, Stacy K., Nicholas A. Kraft, and Letha H. Etzkorn. "Source code retrieval for bug localization using latent dirichlet allocation." 2008 15Th working conference on reverse engineering. IEEE, 2008.

[8] Misherghi, Ghassan, and Zhendong Su. "HDD: hierarchical delta debugging." Proceedings of the 28th international conference on Software engineering. 2006.

[9] Denisov, Alex, and Stanislav Pankevich. "Mull it over: mutation testing based on LLVM." 2018 IEEE international conference on software testing, verification and validation workshops (ICSTW). IEEE, 2018.