Research of Spectrum-based Fault Localization Techniques

Mingan Huang, Yuhang Li, Siyi Chen, Yanhan Lin

***Abstract*—The act to identify the locations of faults in a software is known as software fault localization, which is considered as one of the most time-consuming steps in software debugging. As the increasing size and complexity of programs, manual debugging is becoming quite expensive. In order to produce reliable software with lower cost in fault localization, there has been a great demand for advanced fault localization techniques that can help programmers to locate the faults in their programs. In this research, a comparison between six different spectrum-based techniques is conducted to suggest which technique is better than the others. Results indicates that Dstar is more effective at locating faults than other five techniques it is compared to. An empirical evaluation is also conducted to illustrate how effectiveness is determined, and how to compare these techniques.**

***Index Terms*—debugging, fault localization, spectrum based fault localization**

I. Introduction

SOFTWARE programming is a fundamental skill for technical support in our life and work today, and it has been widely used not just in software engineering and computer science, but also used in many safety-critical systems in industries such as banking business, medical service, and data analysis, etc. Given the important roles in various fields, software programming processes needs to be treated very cautiously. As a significant step in software programming, debugging is highly valued since it is a checkpoint for developers to verify their programs. Programs without appropriate debugging processes would have multiple failures in execution and may lead to huge loss. Furthermore, software faults in safety-critical systems would result in not only huge financial losses, but also information disclosure, or even potential harms of life [5].

In order to debug programs, programmers need to find errors first so that they could fix the bugs appropriately, and sometimes finding bugs is much more time consuming than solving them. This thought brought us the idea of using a fault localization technique to locate the faults. If the location and type of the faults are found out by the fault localization technique, the solving procedure using either repair tools or manual efforts can save a lot of time. Spectrum-based fault localization (SBFL) techniques are widely applicable fault localization techniques, which also have high efficiency since sometimes they do not need to read the entire code to locate the errors. So we decide to compare different SBFL techniques and try to find which one is better than the others.

Spectrum-based fault localization (SBFL) techniques detect the locations of faults in programs based on the execution information[11] sets which is called spectra in those programs. Given the spectrum of failing cases and passing cases, a SFL technique usually outputs a list of unordered suspicious program locations, including the test objects, lines, or declarations. Moreover, each SFL technique uses one formula to calculate the suspiciousness of each location in the program associated with the faults, so that programmers could save lots of time by concentrating on the locations with the highest suspiciousness value during debugging. In addition, when different programmers work on the same project, they dont need to worry about what other team members write when they can use SFL techniques to check for data control and flow control information to get the fault related information which makes group work much easier.

There are plenty of SFL techniques available on the internet, including some famous SFL techniques such as Tarantula and Ochiai, and some practical SFL techniques such as Dstar. It is meaningful and inspiring to conduct an evaluation between all these SFL techniques and to compare which one is the most practical or has the highest precision, so that programmers can have basic guide to choose one technique that helps them to solve program errors.

A fault localization technique is considered practical if its result is valid on real faults instead of artificial faults[9]. Therefore, this paper compares spectrum-based fault localization techniques on real faults.Each project contains only one correct version and several faulty versions with only one specific error. This setup generates a low failure rate for faults, maybe too low for SFL to work well. However, if we have multiple faults in one project, it will be hard to guarantee mutual exclusion and variable control.

The rest of this report is structured as follows. Section II describes our library, database and different SFL techniques in detail. Section III describes our work related to this topic. Section IV describes the measurement that how we get the final result and problem encountered. Section V summarizes our changes since the progress report. Section VI introduces our final results, including our study on different SBFL techniques and our study on different types of faults. Section VII describes the problem we have encountered. Section VIII discusses future work. Section IX highlights our lessons learned Section X is our conclusion.

II. Evaluating fault localization

A. Spectrum fault localization

An important part of diagnosis and repair consist in locating faults, and various tools for automated debugging and systems diagnosis implement spectrum-based fault localization (SFL), an approach to evaluate based on analysis of the differences in program spectra[2]. By shorting the test-diagnosis-repair cycle, SFL has contribution on reducing the debugging effort. One of SFL algorithms we selected has constructed an input matrix, including the detailed information of what components were executed. Each matrix column represents a System Under Test(SUT) component, and the last column indicates the error vector having the execution results[10].

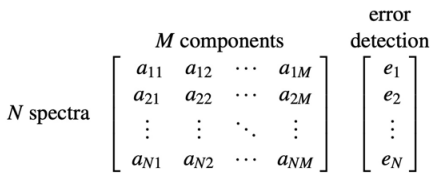

Fig. 1. Spectrum Based Fault Localization

As we can see from the figure 1, N represents the number of test execution, M represents the number of SUT components, a represents code coverage, and e represents test execution results. Therefore, SFL algorithms could calculate the failure probability by using each component.

B. Gzoltar(library)

GZoltar is a framework for automatic testing and debugging phases of the software development life-cycle[8]. At this time, the framework is provided as a command line interface, ant task, maven plugin, and Eclipse plugin. It is an on-going project, which is interesting features that have not yet been implemented. In our research, We have developed some features of it to implement debugging function. Gzoltar provides the infrastructure to automatically instrument the source code to produce runtime data, which is analyzed to return a ranked list of possible faulty locations[10]. The software developers(Gzoltars user) form a mental image of the faulty System Under Test (SUT) to make a conceptual representation of that system visible. By using Gzoltar, a software developer could quickly find the root-cause faulty with the interactive visualization of the SUT.

C. SFL equations

Tarantula[6]

$$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed + passed(s)/totalpassed}$$

Tarantula equation uses information gathered from large amounts of data about system tests, such as the pass/fail information about each test, and the total pass/fail information about the whole system test[6]. In this equation, we define passed(s) to be number of passed test cases that executed statement s one or more times. And totalpassed is the total number of test cases that pass in the entire test suite. failed(s) and totalfailed are defined similarly to passed(s) and totalpassed.

Ochiai[2]

$$S(s) = \frac{failed(s)}{\sqrt{totalfailed * (failed(s) + passed(s))}}$$

Ochiai equation utilizes the data collection, which provides a perspective view on the dynamic behavior of software system[2], to compute the Ochiai coefficient. passed(s), failed(s) and totalfailed are defined similarly to Tarantula equation.

Op2[8]

$$S(s) = failed(s) - \frac{passed(s)}{totalpassed + 1}$$

Op2 utilizes a fault localization technique called Weighting Tests Cases with Multiples Faults, which is based on the assumption that if we observe failed test case executing multiple faults, suspiciousness of these statements will be more than executing a single fault. passed(s), failed(s) and totalpassed are defined similarly to previous equations[8].

Barinel[1]

$$S(s) = 1 - \frac{passed(s)}{passed(s) + failed(s)}$$

For a single-fault program, Barinel produces a theoretically optimal diagnostic ranking for our observations[1]. First the maximum likelihood estimation passed(s)/(passed(s) + failed(s)) returns the so called pass fraction, where passed(s), failed(s) and totalpassed are defined similarly to previous equations. Then using one minus the pass fraction should yield the suspiciousness of a statement.

DStar[12]

$$S(s) = \frac{failed(s)^*}{passed(s) + (totalfailed - failed(s))}$$

Dstar is different from other techniques in our research since it has a parameter failed(s)*, and it is constructed to realize four intuitions appropriately[12].

1) : If a suspiciousness value is assigned to a statement, it should directly proportional to the number of failed tests that cover it. It also means that a statement should have a high suspiciousness if it is frequently executed by failed tests.

2) : If a suspiciousness value is assigned to a statement, it should inversely proportional to the number of successful tests that cover it. It also means that a statement should have a low suspiciousness value if it is frequently executed by successful tests.

3) : If a suspiciousness value is assigned to a statement, it should inversely proportional to the number of tests that do not cover it. It also means that a statement should have a low suspiciousness value if the failed tests case does not cover a statement.

4) : Since we are trying to locate the true faults in the program, we should assign higher importance to the information obtained from Intuition 1 than Intuition 2 or Intuition 3.

The first three intuitions can be constructed by the Kulczynski coefficient which is expressed as a/(b + c), or in the case of our research as failed(s) / (passed(s) + (totalfailed - failed(s)), where totalfailed -failed(s) represents the number of failed tests does not cover a statement. For the fourth intuition, a coefficient as $failed(s) * /(passed(s) + (totalfailed - failed(s))$ should be used, where the value of * is greater than or equal to 1. Thus Dstar uses a coefficient modified from the original Kulczynski coefficient to fulfill all four intuitions.

jaccard[3]

$$S(s) = \frac{failed(s)}{totalfailed + passed(s)}$$

Jaccard constructs its similarity coefficients to calculate suspiciousness ratings based on the knowledge of data clustering[11]. Where failed(s) is the number of failed test cases that cover a statement, passed(s) is the number of passed cases that cover a statement, and totalfailed is the total number of failed test cases.

D. Mutation-based fault localization(MBFL)

Mutation analysis will be used in this research to design new software tests and evaluate the quality of original software tests. It is an extended Spectrum-based fault localization (SBFL) techniques by considering not just the execution of the statement itself, but also whether a change to that statement affects the final results. This technique requires slightly modifying a program in different ways. Each mutated version will be called a mutant and the test will try to kill the mutant by detecting and rejecting mutants according to the different behaviours between the original version and mutated version.The most important part of our MBFL is the evaluation of suspiciousness of injected mutants, which means the likeliness of a statement to affect failing or passing tests. A test case will be considered killing a mutant if we observe a different test outcome from the mutant compared to the outcome yielded by the original version.

Two famous MBFL techniques will be used in our study: MUSE and Metallaxis. MUSE: The MUSE approach based on the assumption that modifying correct statements is more likely to cause passing tests to fail than modifying faulty statements, and modifying faulty statements is more likely to cause failing test to pass than modifying correct statements.

MUSE could generate a set of mutants mut(s) for each statement s, then we define M(m) to be the suspiciousness for each mutant, and each M(m) yields a statement suspiciousness score S(s). Then it ranks M(m) based on the equation $M(m) = f(m) - (f2p/p2f) * p(m)$, where f(m) is the failing test that passed with m inserted, and f2p is the total number of cases in the research where a mutant caused a failing test to pass. p(m) is the passing test that passed with m inserted and p2f is defined similarly to f2p.

III. RELATED WORK

Early studies only use failed test case for spectrum-based fault localization, through this approach has been deemed ineffectively. Later studies achieved improved results by using both the successful and failed test case, and then emphasizing the difference between them[11]. As we mentioned before, there is a rich collection of literature on various techniques that aims to locate the fault on software and make it more effective. Spectrum-based fault localization is the most expensive of all debugging activities, which has been certified by many studies[1]. Various measures have been proposed to improve fault localization[1] as well. Collofello and Cousins[4] develops that such spectra can be used for software localization. Specific information can be used to identify suspicious code that is responsible for the execution failing. Otherwise, code coverage or Executable statement hit spectrum indicates which part of the program under testing have been covered during the execution. In our study, it is related to fault localization and its techniques based on spectrum. Otherwise, there are additional examples of program spectrum-based fault localization. For instance, Program Invariants Hit Spectrum, which records the coverage of program invariants, and Predicate Count Spectrum Based that records how predicates are executed and can be used to track program behaviors that are likely to be erroneous [11].

For mutation - based fault localization, Metallaxis as an additional approach has not been used in our study, it has the same assumption and parameters with MUSE, but uses a different suspiciousness formula such that $M(m) = f(m)/\sqrt{totalfailed * (f(m) + p(m))}$, where f(m) is the number of failing tests whose outcomes are changed with m inserted, and totalfailed is the number of tests that failed on the original test. p(m) and totalpassed are defined similarly to f(m) and totalfailed. In addition, Metallaxis has a less restrictive definition of killing a mutant than MUSEs: Metallaxis detects all the changes in the test behavior, whereas MUSE defines a failing test to kill a mutant if and only if this mutant causes the test to pass.

IV. Measurement

The whole process is divided into five parts, including data processing, suspiciousness calculating, rank based on suspiciousness, divided fault information we obtained into different categories and rank based on different categories.

A. Data processing:

we collect fault information such as exact bug lines, source-code lines from 395 versions in Defects4j. Gzoltar is used to collect information during the entire test suite. It generated two kinds of files(Spectra and Matrix) for each bug version. Spectra file contains a list of components, and the granularity of components in our research is the line. Matrix file is a binary coverage matrix which rows represent the coverage of each test case and its output, and columns mean the components in Spectra file. When the test case failed, the output is -, otherwise +, and 1 represents that this test case covered that component, otherwise 0. Figure 3 is an output example of a simple test project, containing a spectra file and a matrix file.

B. Suspiciousness calculating:

In each version, the number of + in matrix file is totalpassed, the total number of passed test cases, while the number of - is totalfailed. For example,for each component in the Spectra file, we counted the number of passed/failed test cases that executed that component. Then, we calculated the suspiciousness value of each component via different SBFL formula.

C. Rank based on suspiciousness:

After suspiciousness calculating, we sorted the suspiciousness value to figure out at which line the SBFL techniques can detect the exact bug components. Sometimes, several components got the same suspiciousness value, and we gave them the average rank score. There are two formats of the rank score, one is a raw line number, and the other is a percentage of the whole project.

D. Divide into categories and rank:

By reading the source code and exact bug lines, we classify the bug versions into different categories such as conditional, initialization, return, method call. A conditional type fault is the fault happened at boolean conditions, comparing two values and then returning the value true or false. An initialization type fault represents creating a new error object. Returning means the exact bug line is a return statement. A method call type fault is an error in calling a method in the project.

V. Summary of changes since progress report

We planned to include Mutation-based fault localization techniques in our research, such as Muse and Metallaxis. It is expected that the results of SBFL techniques and MBFL techniques will have obvious differences. However, we could not successfully implement killmap because of limited time.

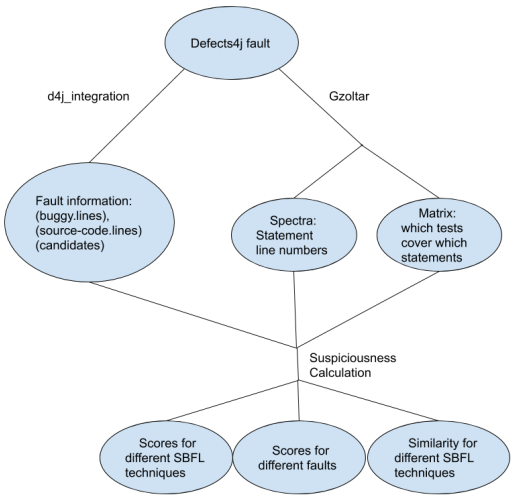

Fig. 2. A general workflow of Measurement

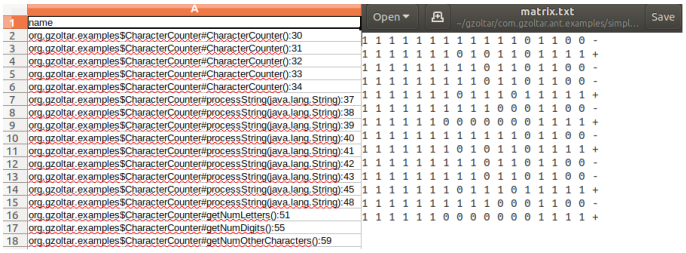

Fig. 3. output of Gzoltar

We decide to classify the faults into different categories and observe the effectiveness of spectrum-based technique for different types of faults. The reason for doing this is to detect whether SBFL techniques have high effectiveness on locating certain types of faults. Even though the classification of different faults is simple and straightforward, it is still considered as a right direction to work on.

Adding the percentage of average rank with total test objects to avoid information hiding. Originally our results only include tables of average rank for each project and SBFL technique to show their effectiveness, however these tables neglect the information of total test cases for each project. For example, if we have Ochiai detects the real faults for project Chart at rank 20 of 30, and Ochiai detects the real faults for project Closure at rank 1000 of 10000. Apparently Ochiai performs better for project Closure since it detects the real faults at 66.7% of the total test cases for project Chart and 10% for project Closure. But people may think rank 20 is much better than rank 1000 if the information of percentage is missing. And we believe this change is significant to complete our research.

We used to conclude our final results, which is the best SBFL technique for programmers, based on one table of

average rank for each SBFL technique. According to the suggestion of our instructor, we do not believe that it is the best way to do the comparison. Because a FL technique with the lowest average rank of detecting the true faults does not mean it will always perform better than other FL techniques. Instead of comparing them using one table, we create several histograms to compare two metrics A and B for effectiveness. And we compare these six SBFL techniques two by two based on the following logic: if we have metric A >metric B and metric B >metric C, we can conclude metric A >metric C. The details of these histograms will be discussed in the next Section.

VI. RESULTS

To compare two SBFL techniques A and B for effectiveness, we first use one of the techniques as the reference technique. Then we subtract the rank score of B from the rank score for A. Result of positive value means that A performed better than B and a negative value means B performed better than A. The difference corresponds to the magnitude of improvement. For example, for a given version, if the rank score of A is 30% and the rank score of B is 40%, then the improvement of A over B is 10%, which means that developers would go through 10% fewer statements if they used A. Figure 5,6,7,8,9,10,11,12 show the results for the studies. The horizontal axes represent the number of versions that show differences in the rank score of fault localization. The vertical axes represent the percentage difference in rank score, which is the magnitude of improvement. The zero-level lines represent the performance of the reference technique. Bars above the zero-level lines represent versions for which our technique performed better than the reference technique, and bars below the zero-level line represent versions for which our technique performed worse.

The basic comparison logic is to divide six techniques into three pairs first. In this case, we paired Ochiai and Barinel, Dstar and Jaccard, tarantula and Op2 into three competitor groups. After we got the winner group, we compare among the three winning techniques and among the three losing techniques. The least rank in the winners bracket needs to compete with the best rank in the losers bracket and if the least rank in the winners bracket wins, the comparison is over. We didnt use any computer program to run this algorithm, so we didnt need achieve traversal which will make the comparison count = 3+3+3+2+1 times and generate 12 comparison groups. Here we have 8 figures showing all 8 comparisons necessary to rank 6 techniques.

A. result of effectiveness studies

The first comparison group is between Barinel and Ochiai SBFL techniques. The purpose of this comparison is to compare the effectiveness of the Barinel to the Ochiai technique. To do this, we used the Ochiai as the reference point and subtracted the rank score of the Barinel from the rank score of the Ochiai technique. Figure 5 shows the

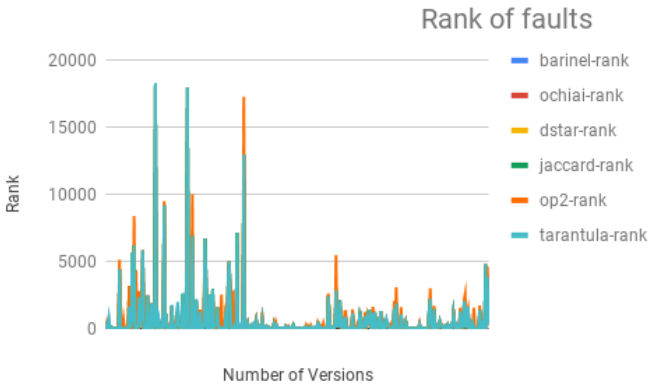

Fig. 4. rank of faults

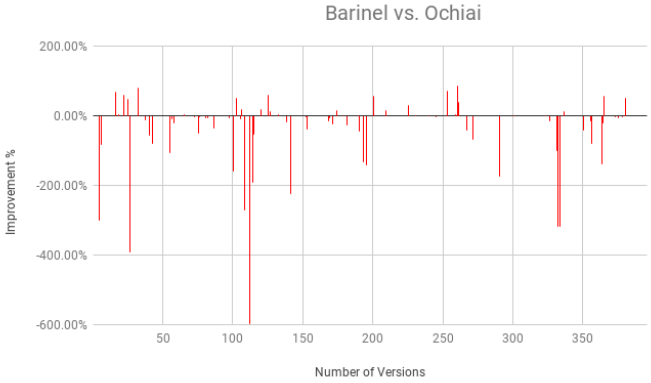

Fig. 5. Barinel vs Ochiai

results of the comparison. The Ochiai performs better than the Barinel technique on 62 versions but performs worse on 32 versions. The Barinel performs the same as the Ochiai technique on 301 versions. So we put Ochiai in winners bracket.

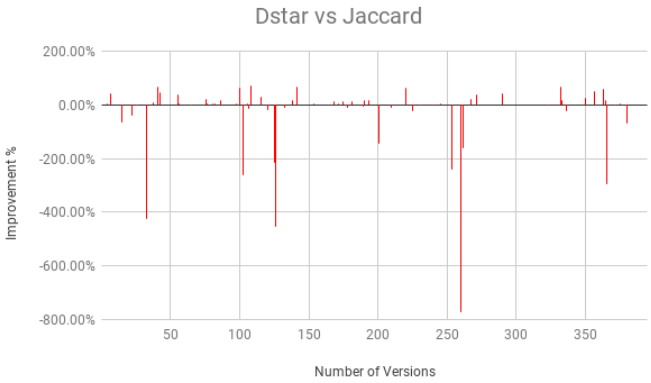

Fig. 6. Dstar vs Jaccard

The second comparison group is between Dstar and Jaccard techniques. The purpose of this comparison is to compare the

effectiveness of the Dstar to the Jaccard technique. To do this, we used the Jaccard as the reference point and subtracted the rank score of the Dstar from the rank score of the Jaccard technique. Figure 6 shows the results of the comparison. The Dstar performs better than the Jaccard technique on 58 versions but performs worse on 27 versions. The Dstar performs the same as the Ochiai technique on 310 versions. Then we put Dstar in winners bracket.

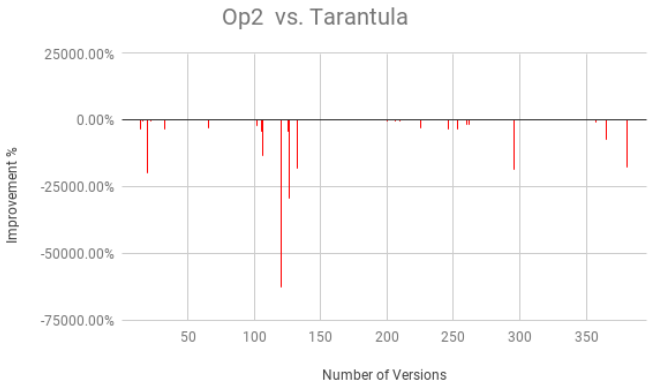

Fig. 7. Op2 vs Tarantula

The third comparison group is between Op2 and Tarantula techniques. The purpose of this comparison is to compare the effectiveness of the Op2 to the Tarantula technique. To do this, we used the Tarantula as the reference point and subtracted the rank score of the Op2 from the rank score of the Tarantula technique. Figure 7 shows the results of the comparison. The Op2 performs better than the Tarantula technique on 87 versions but performs worse on 82 versions. The Op2 performs the same as the Tarantula technique on 226 versions. So we put Tarantula in winners bracket. To this point, we have all three winners. The next step would be ranking among the winners bracket.

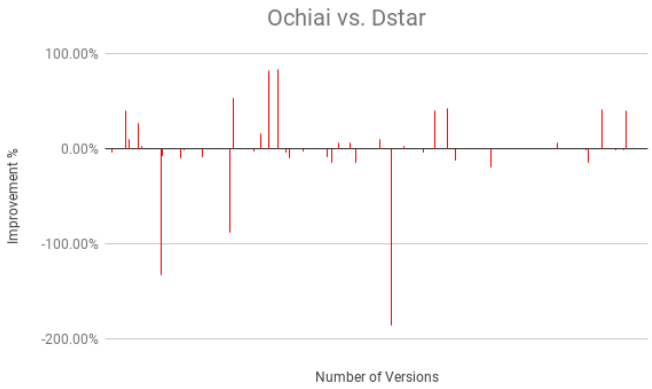

Fig. 8. Ochiai vs Dstar

The first comparison in winners bracket is between Ochiai and Dstar. The purpose of this comparison is to compare the effectiveness of the Ochiai to the Dstar technique. To do this, we used the Dstar as the reference point and subtracted the rank score of the Ochiai from the rank score of the Dstar technique. Figure 8 shows the results of the comparison. The Dstar performs better than the Ochiai technique on 30 versions but performs worse on 20 versions. The Ochiai performs the same as the Dstar technique on 345 versions. So we have Dstar is better than Ochiai(Dstar >Ochiai).

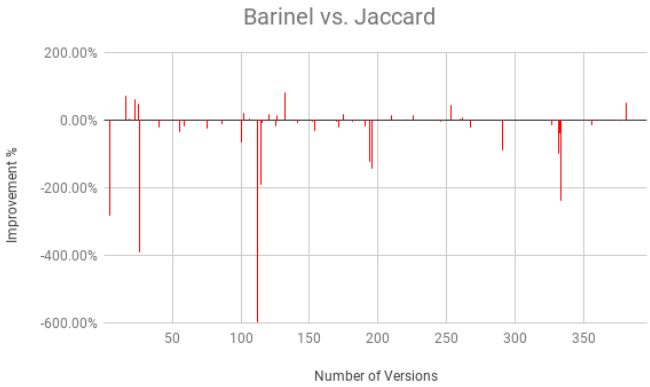

Fig. 9. Barinel vs Jaccard

The first comparison in losers bracket is between Barinel and Jaccard. The purpose of this comparison is to compare the effectiveness of the Barinel to the Jaccard technique. To do this, we used the Jaccard as the reference point and subtracted the rank score of the Barinel from the rank score of the Jaccard technique. Figure 9 shows the results of the comparison. The Jaccard performs better than the Barinel technique on 45 versions but performs worse on 24 versions. The Barinel performs the same as the Jaccard technique on 326 versions. So we have Jaccard is better than Barinel(Jaccard >Barinel).

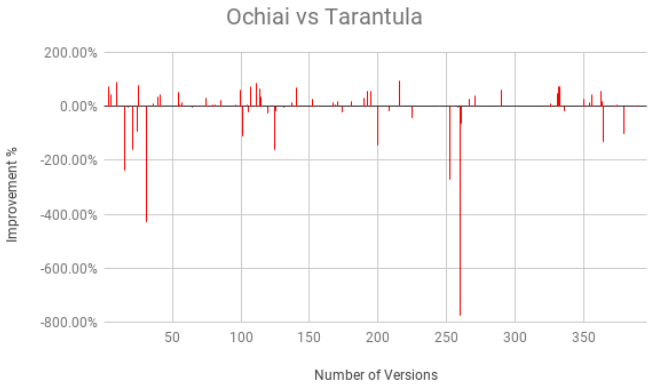

Fig. 10. Ochiai vs Tarantula

The second comparison in winners bracket is between Ochiai and Tarantula. The purpose of this comparison is to compare the effectiveness of the Ochiai to the Tarantula technique. To do this, we used the Tarantula as the reference point and subtracted the rank score of the Ochiai from the

rank score of the Tarantula technique. Figure 10 shows the results of the comparison. The Ochiai performs better than the Tarantula technique on 64 versions but performs worse on 32 versions. The Ochiai performs the same as the Tarantula technique on 299 versions. So we have Ochiai is better than Tarantula(Ochiai >Tarantula). So now we got the rank among winners bracket which is Dstar >Ochiai >Tarantula.

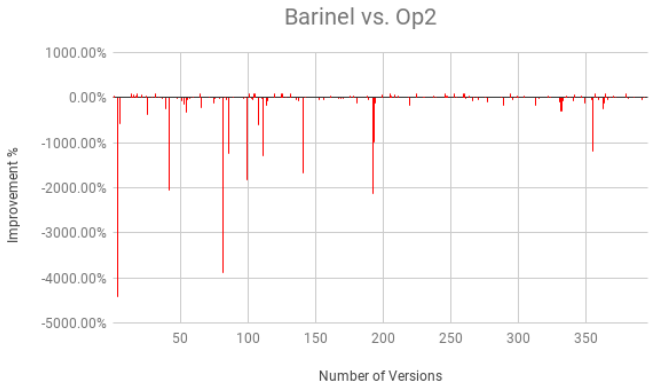

Fig. 11. Barinel vs op2

The first comparison in losers bracket is between Barinel and Op2. The purpose of this comparison is to compare the effectiveness of the Barinel to the Op2 technique. To do this, we used the Op2 as the reference point and subtracted the rank score of the Barinel from the rank score of the Op2 technique. Figure 11 shows the results of the comparison. The Op2 performs better than the Barinel technique on 85 versions but performs worse on 82 versions. The Barinel performs the same as the Op2 technique on 228 versions. So we have Op2 is better than Barinel(Op2 >Barinel). Since we have Jaccard >Barinel and Op2 >Barinel, we still need to perform one more comparison between Jaccard and Op2.

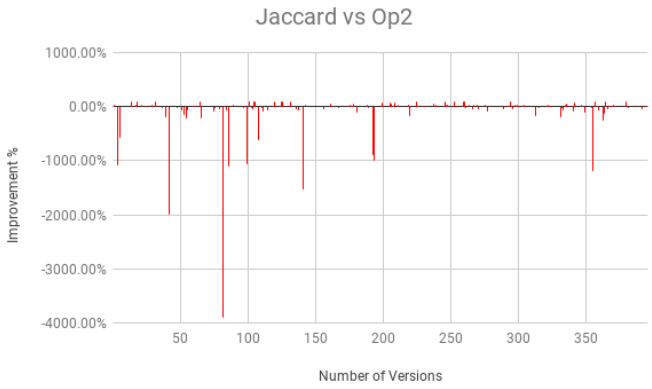

Fig. 12. Jaccard vs Op2

The third comparison in losers bracket is between Jaccard and Op2. The purpose of this comparison is to compare the effectiveness of the Jaccard to the Op2 technique. To do this, we used the Op2 as the reference point and subtracted the rank score of the Jaccard from the rank score of the Op2 technique. Figure 12 shows the results of the comparison. The Op2 performs better than the Jaccard technique on 81 versions but performs worse on 79 versions. The Op2 performs the same as the Jaccard technique on 235 versions. So we have Op2 is better than Jaccard(Op2 >Jaccard). So now we got the rank among losers bracket which is Op2 >Jaccard >Barinel.

We should perform one more comparison between the highest rank in losers bracket(Op2) and lowest rank in winners bracket(Tarantula), but since we already did that when we categorize winners and losers bracket, we can get the result which is Dstar >Ochiai >Tarantula >Op2 >Jaccard >Barinel.

B. results of average rank and percentage

We screened out our final results in two tables, Table 1 contains all bug version and Table 2 excludes bug versions of fault of omission. For missing code faults, the bug components(lines) are not recorded in the spectra file, and further, no suspiciousness value.

Closure is the most complex project, which contains a tremendously large number of statements. Even though it has the highest average rank score of over 1000, it is not considered as the project with the lowest performance. Because it only has a percentage score of 6% to 7% from Table 2. In other words, it is not too bad for a programmer to locate the real fault after checking the first 6% or 7% of the total test cases.

Mockito is considered as the project with the highest percentage score of nearly 19% (fault of omission not included) from Table 2, which means that for any faulty version in Mockito, a SBFL technique usually detects the real fault at rank one fifth of all test cases. However, project Mockito has 38 faulty version, and only 13 of them are not faults caused by omission, which is a rather small dataset compared to project Closure or Math. Therefore, in order to have a better estimation of each project, we need to increase the faulty versions or add different types of fault to them so that each project could generate similar dataset.

According to the Overall average row in Table 2, we should have a result of Dstar >Ochiai >Barinel >Jaccard >Tarantula >Op2, which is different from the result we have in Part A. We observed that there exist some extremely large rank values in our dataset that could affect the average, and because the methodology we used in Part A only involves the number of faulty versions that is performed better or worse, it is reasonable to have different results from Part A and Part B. However, we consider the final result from Part A is more precise and the results from Part B is more like a reference.

C. results of faults of type

We classify the faults in Defects4j by checking its fault information. The table shows the result of four faults types.

TABLE I
AVERAGE RANK AND AVERAGE PERCENTAGE

project	tarantula		ochiai		op2		barinel		dstar		jaccard	
	rank score	percent	rank score	percent	rank score	percent	rank score	percent	rank score	percent	rank score	percent
Chart	308.37	7.74%	324.32	6.66%	398.55	8.43%	307.41	5.93%	328.53	7.04%	326.46	6.94%
Closure	1160.29	6.69%	1134.83	6.55%	1180.65	7.25%	1160.25	6.69%	1136.48	6.57%	1153.51	6.66%
Lang	59.87	10.21%	57.09	9.30%	70.12	11.33%	58.44	9.46%	56.22	9.27%	57.09	9.24%
Math	292.55	11.93%	289.45	11.90%	371.58	14.41%	292.55	11.93%	289.93	11.89%	290.20	11.77%
Mockito	322.75	17.47%	317.66	17.21%	365.10	19.14%	322.75	17.47%	318.59	17.25%	322.24	17.45%
Time	699.60	11.56%	667.39	11.51%	746.14	13.12%	699.60	11.56%	667.29	11.51%	669.04	11.54%
Overall	576.13	10.11%	566.70	9.81%	621.15	11.46%	575.84	9.87%	567.60	9.84%	573.89	9.85%

TABLE II
AVERAGE RANK WITHOUT FAULT OF OMISSION

project	tarantula		ochiai		op2		barinel		dstar		jaccard	
	rank score	percent	rank score	percent	rank score	percent	rank score	percent	rank score	percent	rank score	percent
Chart	22.58	8.05%	17.73	4.11%	14.69	3.69%	20.66	4.42%	17.84	4.15%	21.46	4.69%
Closure	1160.29	6.47%	1106.12	6.23%	1117.10	6.74%	1140.83	6.47%	1105.82	6.23%	1139.58	6.46%
Lang	63.13	8.51%	57.73	5.85%	46.75	4.72%	58.93	6.20%	54.74	5.64%	57.95	5.88%
Math	200.26	10.46%	199.72	10.43%	235.32	11.95%	200.26	10.46%	199.57	10.43%	199.93	10.44%
Mockito	412.87	18.84%	412.43	18.89%	589.32	27.74%	412.87	18.84%	415.29	19.03%	413.08	18.85%
Time	904.69	15.37%	895.36	15.20%	1052.69	17.76%	904.69	15.37%	892.57	15.13%	902.29	15.32%
Overall	520.60	9.59%	507.41	8.78%	541.76	10.11%	519.83	8.95%	506.98	8.77%	519.17	8.92%

TABLE III
AVERAGE RANK FOR DIFFERENT FAULTS

fault type	tarantula		ochiai		op2		barinel		dstar		jaccard	
	rank score	percent	rank score	percent	rank score	percent	rank score	percent	rank score	percent	rank score	percent
Conditional	409.781	6.08%	412.183	6.17%	555.176	7.76%	409.786	6.08%	410.466	6.12%	409.655	6.19%
Initialization	47.939	3.21%	47.803	3.21%	53.439	3.84%	47.939	3.21%	47.212	3.15%	47.939	3.21%
Return	330.071	13.44%	273.962	10.09%	334.494	12.28%	326.808	10.78%	275.123	10.13%	324.079	10.52%
Method call	321.978	17.42%	316.981	17.26%	344.867	18.52%	321.978	17.42%	316.152	17.24%	321.518	17.41%

Some categories only contain one or two faults, like doing NOT operator, and we do not analyze these in the result.
Conditional faults are the control flow statements faults, and the exact buggy lines are the boolean conditions. However, due to it is part of condition construct or iteration one, and the construct always executes as a whole, all statements in the construct might gain the same suspiciousness value as the exact buggy line. Therefore, it is hard to be localized by spectrum techniques. For initialization faults, which means newing an object, there are fewer statements bound with them, so the rank scores of them are lower than Conditional faults. The score of return type and method call type sometimes is one kind of fault, because calling one method or function means to get its return object. This can explain why the two rank scores are almost close.

VII. EVALUATION

Location of localization is not always located precisely to one line of code which leads to the fault. The data flow and control flow related codes can also be referred to the same fault rank as the leading cause. We know that Java compiler can actually trace back to the exact line which causes the fault. There may be some improvement in the algorithm of finding the faults.

When we tried to run the defects4j database on our computer, we encountered many problems. For example, we spent so much time on figuring out how to implement our program so that it could achieve traversal over the whole data folder. Before we figured out how to accomplish traversal over the whole database, we only implemented the suspiciousness checking program on one data folder (//use another description). We naively assumed that if our analysis approach worked on one data set, the other data sets would be the same. So we used one data set to test our suspiciousness program.

One of the problems we confronted was when we tried to analyze the data generated by running one data set of the Defects4j database called Chart, the results generated were exactly the same for all of the formulae. At first, we thought the code we had for ranking the suspiciousness was improperly implemented. We checked the source code several times collaboratively, but we didnt find any error that might cause the situation. We were stuck at this stage for days and then we decided to move on. The problem was solved when we successfully generated all the suspiciousness results and found out that the Chart data was just a coincidence and the ranking method turned out to be correct.

When we categorize the type of faults, we would sometimes see the situation that several types of faults appear at the same time. In this situation, we would generally trace back to the java source code and see which type actually triggered the error chunk. At the end, we found that almost all of the error chunks are triggered by fault of omission.

The meaning of the categorization is that some of the fault localization techniques are especially good at finding certain types of errors.

We found a research online about fault of omission and I think our results agreed with the research result that from 22% to 54% of faults were omissions of all the faults.

VIII. Future Study

One of our main concerns during the process of analysis is whether we should exclude fault of omission in our results. When we categorized different types of faults, we simply classified all the faulty clauses which contain fault of omission into the fault of omission category no matter they are complex fault or not. Sometimes the complex faults contains different kinds of errors and omission errors. If we dont trace back to the source code, we cannot know which error actually triggers the bug. The tracing process is really time consuming and human resource demanding. Moreover, fault of omissions can be really hard to recognize because its simply a missing part of code. If programmers cant find the missing code at the first place, there is a great chance that the same programmers still have a hard time finding the omission of code even when they know which part of their code contains the omission error. because their minds are still biased. There are three types of omissions,coding omissions, design omissions, and requirements omissions.[7] If we have more time, we want to explore more about fault of omission in future work. Some of the techniques are unsuitable for Defects4j database because of its low error rate. We also want to dig deeper and find out what other fields can those techniques be implemented.

IX. Lessons learned

In the whole process of our research, we firstly learnt how to be a responsible researcher. In detail, we searched a lot of materials to plan and understand what is our research. After deciding researchs topic, we did preparation before our research work and set up environment in order to run tools and library we used. Secondly, time management is a critically important concept in our graduate research. We learnt how to arrange our work at different stages for each team member. Distribute the balanced work part to each team member that avoid huge difference. On the other hand, some of our work is achieved before deadline, which means we need to strengthen our ability of time management. In technical aspect, we realized the application of fault localization and automatic debugging tools and apply to our research. Defect4j and Gzoltar are good examples that improve our knowledge in fault localization and debugging field. Thirdly, the most important part is the result analysis. We learnt how to analyze the suspiciousness and compare the rank between each techniques. In order to present correct and complete result, we used two approaches to avoid information hiding and fault. We changed the result we obtained using more accurate approach. Therefore, we realized that implementation could influence results. By categorizing the type of fault, we learnt which kind of fault that could lead to errors in research. Thus, we could avoid the occurrence of these errors as much as possible.

X. Conclusion

Software is fundamental to our lives today, its influence is practically ubiquitous with its increasing development and usage. Software fault as a primary problem confusing software developers. Fault localization techniques are developed to solve this problem, such as traditional fault localization and advanced fault localization. In our study, we focus on one of advanced fault localization techniques named Spectrum based fault localization as a main debugging tool. Spectrum based fault localization achieved accurate results by using successful and failed test cases and emphasizing the contrast between them. The defect4j and Gzoltar are two various tools we used to obtain the fault information like source code line, buggy line, candidate and matrics. In our study, we used six approaches including Tarantula, Barinel, Jaccard, Dstar, Ochiai and Op2 and evaluating their effectiveness by comparing rank. The fomulas of techniques are composed by four parameters, which includes the total number of passed test cases, the number of those that executed statement. Similarly, the total failed cases and the number of those that executed statement are important parameters in these fomulas. Through source code information, the fault of type is clarified as a critical standard when results are evaluated. Moreover, mutation based fault localization is learned and introduced in our case as well. By series of comparison, we found that Dstar is the best fault localization in our research. This result is same with most of research results. Barinel is the worst effectiveness in our research. The whole effectiveness results in order respectively are Dstar, Ochiai, Tarantula, Op2, Jaccard and Barinel. To summarize, this report analyzed and evaluated the effectiveness and its performance of each technique based on spectrum based fault localization. Finally, the goal of this research are successfully accomplished in time.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. “Spectrum-based multiple fault localization”. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2009, pp. 88–99.

[2] Rui Abreu et al. “A practical evaluation of spectrum-based fault localization”. In: *Journal of Systems and Software* 82.11 (2009), pp. 1780–1792.

[3] Shay Artzi et al. “Finding bugs in web applications using dynamic test generation and explicit-state model checking”. In: *IEEE Transactions on Software Engineering* 36.4 (2010), pp. 474–494.

[4] James S Collofello and Larry Cousins. “Towards automatic software fault location through decision-to-decision path analysis”. In: *afips*. IEEE. 1899, p. 539.

[5] S Gandel. *Why Knight lost $440 million in 45 minutes. Fortune*. 2012.

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

[7] Brian Marick. *Fault of Omission*. http://www.exampler.com/testing-com/writings/omissions.html. [Online; accessed 19-Janunary-2000]. 2000.

[8] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. “A model for spectra-based software diagnosis”. In: *ACM Transactions on software engineering and methodology (TOSEM)* 20.3 (2011), p. 11.

[9] Spencer Pearson et al. “Evaluating & improving fault localization techniques”. In: *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-16-08-03* (2016).

[10] André Riboira and Rui Abreu. “The GZoltar project: A graphical debugger interface”. In: *Testing–Practice and Research Techniques*. Springer, 2010, pp. 215–218.

[11] W Eric Wong et al. “A survey on software fault localization”. In: *IEEE Transactions on Software Engineering* 42.8 (2016), pp. 707–740.

[12] W Eric Wong et al. “The DStar method for effective software fault localization”. In: *IEEE Transactions on Reliability* 63.1 (2014), pp. 290–308.