A Comparison of Popular Software Fault Localization Techniques and Improvements

Brandon Hart
B.S – Computer Science
Missouri State University

Scott Popken
B.S – Computer Science
Missouri State University

Alex Webber
B.S – Computer Science
Missouri State University

***Abstract*—**Software Fault Localization is a growing and important topic within the field of Computer Science. We discuss several popular Software Fault Localization (SFL) techniques, their advantages and disadvantages, and limitations for each one. Slice-based, spectrum-based, and block-based with a novel kernel measure are each discussed before introducing a new contribution to the field. We then introduce a technique composed of slicing and spectrum-based localization, designed to improve the fault localization process. Our technique takes advantage of several commonly used SFL tools and utilities, and makes an attempt to prove the validity of a solution involving GZoltar and a manual slicing of a program to identify and detect the location of software faults with a numerical and visual approach. The objective of the spectrum-based is to identify the location of a fault, where the slice-based component is designed to reduce the surface area that a bug takes up, allowing for much quicker testing and an even better ability to quickly find faults throughout a program of any given size. After conducting our experiment and data collection, we propose the creation of an automated slice-based tool to ensure accurate slicing occurs, then, we propose a combination tool that conducts spectrum-slice based testing automatically for a user. The creation of such tool would drastically help the field of Software Fault Localization. Finally, we discuss and acknowledge possible threats of validity to our idea and justify our rationale and reasoning behind the presented improvement.

Keywords*—*software fault localization, debugging, testing, comparison of software fault localization, manual, automated, slice-based localization, spectrum-based localization, novel kernel measure, block-based localization, GZoltar, manual slice, automated slicing, eclipse plugin, Kaveri, Indus, JSlice

I. Introduction

Software Fault Localization is the process of identifying where faults occur in a computer program. No team wants to deploy software riddled with errors, so teams turn to Software Fault Localization techniques to reduce that likelihood. Software errors were estimated to cost the United States economy around $59 billion in 2002, with that cost likely increasing year-over-year [1]. This paper will discuss several widely known Fault Localization techniques, identifying advantages and disadvantages for each one, and will then propose a new technique of SFL, consisting of a combination of spectrum and slice based localization methods, designed to be a hybrid to improve the process using two of the most popular localization techniques (see Fig. 1). This proposal will include data supporting it, and appropriate steps to utilize it in software projects.

There are a wide range of methods for determining where software faults occur in code. At the most fundamental level, these can be broken down into two different sections – traditional, "manual" techniques that require a high level of developer interaction with the process, and advanced, automated techniques that are more suited for a large project with background execution capabilities.

II. An Overview Of Techniques

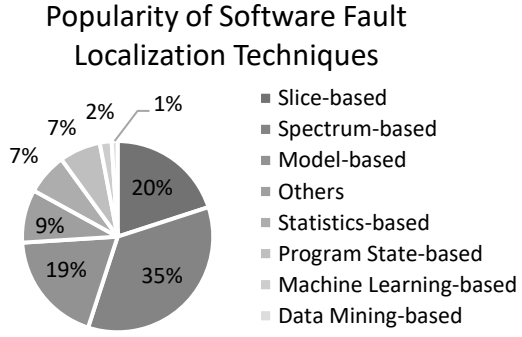
A. Traditional Methods

There are several different traditional localization methods. These methods all require user involvement and are incredibly time consuming. Additionally, they require manual interaction from the developer to the software unit, making them less suitable for large, complex projects. Two of these techniques include but are not limited to logging and breakpoints.

- Logging – inserting statements into a computer program to determine the state of variables, location of code execution, or other information about a program. [1].
- Breakpoints – allows a developer to run a program with preset pauses inserted throughout the code. At each breakpoint, a developer can analyze the variables and program state information. Many Integrated Development Environments (IDEs) include these [1].

B. Advanced Methods

There are many different advanced software fault localization methods available today. For instance, there are eight different techniques discussed in [1], ranging from slicing a program into multiple segments, to automated fault localization techniques utilizing machine learning and data mining. Discussed comprehensively in [1], researchers compiled a database of available papers for the eight different techniques. The pie-chart visualization below shows those techniques and their popularity throughout research papers.



software fault localization techniques. [1]

For the scope of this paper, we have decided to discuss in detail the three most-popular techniques shown in the pie-chart above. These three techniques include (1) Slice-based, (2) spectrum-based, and (3) a block-based novel kernel solution. The three selected techniques account for almost 75% of research done on fault localization [1] and will allow for each team member to thoroughly research their own technique. A brief summary of the three techniques is below, and each one will be discussed in greater detail throughout the paper.

- Slicing-based localization – a technique that breaks programs into multiple usable components. The overall objective of this technique is to isolate sections of a program, making it easier for a developer to find a fault in the software [1].
- Spectrum-based localization – a technique that details the program state information and execution data from different branches and loops within the program. Spectrum-based considers test cases throughout the program and assigns a suspiciousness score to code segments that it believes could be the fault of the bug [1].
- Block-based localization with novel kernel – a technique that forms a matrix of statement blocks, and valid input data. That input data is then compared to the proper desired output data for each block [4].

III. Detailed Techniques

Below we discuss in detail each of the three techniques mentioned previously in this paper, including slice-based, spectrum-based, and block-based with novel kernel localization methods. Each technique will be expanded on to include different methods of that technique, if applicable.

A. Slice-based Fault Localization

The most common usage of slicing programs into segments is to make the domain containing bugs smaller, breaking the program into pieces rather than searching through thousands of lines of code for an error that may or may not exist. A bug in the software should be able to be easily traced to a segment, and, if spliced correctly, reduce the time and effort needed to locate and correct the fault [2]

Consider the code segment on the top of the page in Fig. 2. The final cost of tuition is stored in the cost variable at the end of the code. An appropriate splice for the rate variable of this program (that is, only the code that affects the rate variable) would be lines 1, 3, 5, 6, 7, and 8. The other lines are not required for the rate variable, and while they are relevant to the program, they can be removed to make detecting a bug in the rate variable easier.

The slice-based technique can be broken down further into two segments, static and dynamic. Static slicing is comparable to the traditional techniques, as human interaction is needed to splice the program into numerous workable segments. A

```
1   boolean onlineStudent = false;
2   int credits = 15;
3   int rate = 0;
4   int cost = 0;
5   if (onlineStudent){
6       rate = $299;
7   } else {
8       rate = $228; }
9   cost = rate * credits;
```

Fig. 2. A code sample demonstrating an appropriate slice for a simple Java program multiplying variables.

common disadvantage of this technique is that variables spanning across multiple segments of code can be incredibly difficult to isolate and can behave differently when spliced making it hard to determine the validity of a static splice [1]. Due to this difficulty, dynamic slicing has become a more common and popular trend amongst developers.

Dynamic slicing was introduced due to the inability for static slicing to account for variable values at run-time. Dynamic slicing can remove statements that can affect the value of a variable, making the slice smaller and more accurate than using a static splice. [1] discusses that while dynamic slicing is a better solution than static slicing, it still requires intervention from a developer, and can omit statements that affect the values of variables unintentionally. Finally, slicing is only effective if the bug is inside of the splice. If the bug is outside of the splice, the time has been completely wasted and would have been better spent utilizing a more efficient software fault localization model.

B. Spectrum-based Fault Localization

Spectrum-based fault localization (SFL) is a method of fault localization which examines the pass/fail outcomes of different parts of a program, then estimates the fault risk for the corresponding area [3]. SFL is a low cost and accurate method to detect program faults. While SFL is often utilized in the development phase, the realization that complex programs may never be truly fault free has led to the utilization of SFL in the operational phase as well. Fault detection depends almost entirely on the error detection (pass/fail information) within the program. Since test oracles[1] aren't available in the operational phase of a program, the accuracy of the fault detection is dependent on error detection built into the program (an error is a mistake made by the programmer, whereas a fault is a condition that results in the failure of a software's desired function due to an error). Error detection can be application specific or generic. Application specific error detectors are invariants [2] that are user-programmed, while generic error detectors (known as "screeners") are invariants that are compiler generated. The low cost and acceptable accuracy of screeners can save money by preventing otherwise needed programmer involvement, and also minimizes the time & space cost during runtime [3].

Important strengths of using Spectrum-based fault localization and screeners include a low cost to use, screeners can produce acceptable accuracy, and are optimized for testing

[1] Test oracles are mechanisms that determine whether a test has passed or failed within a program.

[2] Invariants are conditions that must be met through all stages of a computer program.

with reducing performance and memory overhead [3]. However, while they feature acceptable accuracy, they suffer from limited error detection quality.

C. Block and Novel Kernel based Fault Localization

In [4], the authors conduct an experiment to determine how viable a block system technique of Software Fault Localization is in determining errors. The author writes that the "problem of software fault localization may be viewed as an approach for finding hidden faults or bugs in the existing program codes which are syntactically correct and give fault free output for some input instances, but fail for all other input instances." The authors argue that the basic idea behind Software Fault Localization (SFL) is to first form a matrix of order S x B, where S is the number of input instances and B is the number of blocks in the programs code. For each input instance that an error occurs, record a '1,' and if no error occurs, record a '0' instead. Then, implement a block-hit function denoted by Hc<Bik, Bik>.

Bik	Bik	Hc<Bik, Bik>
Block miss (0)	Block miss (0)	-1
Block miss (0)	Block hit (1)	0
Block hit (1)	Block miss (0)	0
Block hit (1)	Block hit (1)	1

Fig. 3. A table showing the relation between Bik and Bik, which are binary variables indicating the block hit or block miss for input instances Si and Sj respectively, with I and j being the index of input instances and k representing the block number.

[4] also proposes a seven-step algorithm for SFL using the proposed Kernel Measure designed to be an effective method for Software Fault Localization. These steps include:

1. Obtain the program flow graph for the given code.
2. From the graph, identify program blocks.
3. Create random sample input instances, by considering worse, average, and best-case situations. (best case input is 1, 2, 3, 4, 5, and worst-case is the opposite).
4. Run the program code on the above legitimate input instances and record corresponding outputs.
5. Classify the outputs as faulty or error free.
6. Form a table with the first 'm' columns indicating blocks and the last column denoting a decision class. From the previous step, obtain the block vectors, denoted by B and the decision vector, D.
7. Find similarities between the decision vector, D, and the block vectors, B that were generated from the algorithm.

In this, there are several advantages and disadvantages. The model was in fact proven to be an effective way of catching bugs, however, did not seem like a good tool to predict where faults might occur. Additionally, this technique is mathematically intensive, and is more complicated for the common user. For advanced programmers and mathematicians, this is a good solution, but to the everyday programmer, a simpler solution should be explored [4].

IV. Improving Fault Localization

To improve the field of Software Fault Localization, we propose a new testing technique, consisting of a combination of slicing-based and spectrum-based elements. Both techniques have their advantages and disadvantages, and a concurrent combination of the two of them produces initially promising signs. Our goal is to be able to detect more faults than simply running slicing-based fault localization. To accomplish this improvement, we propose the following timeline to ensure success.

Step 1 (by July 12th) – discuss our ideal improvement and possible different routes to take this project in. Review each paper summarized and become familiar with the different SFL techniques discussed.

Step 2 (by end of Week 6, July 19th) – look at our individual improvement idea, "a combination of spectrum and slicing based localization" and develop a plan to gather data to prove the merit of our proposed solution.

Step 3 (by end of Week 7, July 26th) – finalize plan to gather data, and complete most of the data collection and compilation process, and repeat rounds of testing to ensure an ample data set is present.

Step 4 (by end of Week 8, due date) – finalize the data, analyze the data, and summarize our findings. Finally, finish the paper and submit it to Blackboard.

We believe that the above steps will properly allow us to complete our improvement idea on time. Each step will consist of a Zoom meeting with team members to ensure that all group members are on the correct track.

V. EXPERIMENT

To conduct our experiment and propose a valid improvement, we will be utilizing two different software tools, the first utilizing spectrum-based testing, and the second utilizing slice-based testing. Should our data collection prove promising, we would like to propose a software tool that would combine this process, making it much more developer friendly and simpler rather than requiring two separate tools. To conduct the spectrum-based component, we chose GZoltar, an automatic testing and debugging tool that calculates suspiciousness values for lines of code based off of the Ochaia algorithm. After computing suspiciousness values with GZoltar, we conducted a manual slice of the bug.

Real software projects are incredibly complex and can take a long time to gain maturity from bugs and errors. For the sake of our research paper given the time constraints for this

course, we decided to utilize a common library for Java Software Fault Localization called Defects4J, which contains common bugs and defects from a large selection of open-source projects including JChart, Apache Commons libraries, Jsoup, and Gson. The complete database currently contains 835 bugs, and can be accessed at github.com/rjust/defects4j. We then narrowed the scope down to one bug from the Apache Commons Math library to assist in a more scalable form of data collection. The bug was compiled by Kevin Bi, a former faculty member at the University of Washington, and currently a software engineer at Google, and is available at gitlab.cs.washington.edu/kevinb22/fault-localization-research/. Our selected bug takes three parameters about the size of a triangle (side A, side B, and side C) to calculate what type of triangle the given inputs make, including scalene, equilateral, isosceles, or an invalid triangle if it cannot be created. The bug also comes with 33 test cases, with 4 failing due to the bug. In the attached Triangle.java file, the issue occurs on line 18, where "(if b == a)" should be changed to "(if b == c)." This is problematic as line 14, "(a == b)" accomplishes the same thing as the bug on line 18, however, causes the triangle value to be incremented by 3 after it has already been incremented once. This causes the program to say that some triangles are valid when they are invalid, and vice versa. The correct fix for this bug is notated in a comment on line 18.

GZoltar was run on the command line by selecting the triangle project, the test cases, and target output. This output also included the project test cases which were operating properly. We then removed the test cases from the output by adding "| grep "Triangle.java" to our command to ensure that only lines from the Triangle class would be included in the fault output. The below table shows the line # and suspiciousness value for the 10 highest lines with problematic potential.

Line	Suspiciousness	
11	.3481553119	if (a <= 0 || b <= 0 || c <= 0)
13	.4170288281	trian = 0
14	.4170288281	if (a == b)
15	.6708203932	trian = trian + 1
16	.4170288281	if (a == c)
18	.4170288281	if (b == a)
19	.6708203932	trian = trian + 3
20	.4170288281	if (trian == 0)
25	.5000000000	if (trian > 3)
26	.6708203932	return Type.EQUILATERAL

Fig. 4. A table showing the 10 highest suspiciousness values line number, and code in the Triangle.java class.

The above suspiciousness values are calculated using the popular Ochiai algorithm, which is an effective algorithm for identifying possibly problematic lines and statements. The formula below is a simplified representation of the Ochaia algorithm, adjusted for readability [2]. GZoltar calculates these values automatically, rather than requiring manual calculation from the developer or tester.

$$S(s) = \frac{\#\ of\ failed\ tests\ executing\ program}{\sqrt{total} failed\ tests\ \times (total\ \#\ of\ tests)}$$

Our results indicate that many lines have *S* values that are concerning. By looking at the data on the table, it could appear that there is more than one bug as lines 14 and 15 have the same *S* values as 18 and 19, where our bug occurs. However, this is because those lines currently do the same thing. A modification of line 18 to fix the bug results in no failed test cases, and a new suspiciousness score of 0. While we obtained success in correcting this code segment ensuring that it was bug free, our response is slightly flawed due to the following:

- We knew the bug in the code before conducting research and data collection.
- Triangle.java only contains one bug, whereas large software projects could contain hundreds of bugs.

The above factors made it easier to find the given bug compared to a larger, more complex, and multi-bug project. Because of this, we then propose adding a slice-based method into a testing plan to help further isolate problematic lines and statements where it is harder to immediately identify a bug given a numerical value. We then took the code for our bug, Triangle.java, and conducted a manual slice on it. We had hoped that a slicing tool would have been built into eclipse or had been offered in a plugin, however, we tried an assortment of slicing tools including Kaveri, Indus, JSlice, JDeoderant, and several other tools, all of which are no longer current and supported for recent versions of eclipse. To conduct our manual slice of the code, we created the below table, featuring all of the available variables within the program compared to the lines of the code. If a variable is included in the line, then it would be included in the slice, and denoted with an x in the respective variable column. By creating a tabular slice table, it becomes easier to visualize the data at hand.

Ln	Code	A	B	C	Trian
10	int trian				X
11	if (a <= 0 || b <= 0 || c <= 0)	X	X	X	
13	trian = 0;				X
14	if (a == b)	X	X		
15	trian = trian + 1;	X	X		X
16	if (a == c)	X		X	
17	trian = trian + 2;	X		X	X
18	if (b == a) // bug	X	X		
19	trian = trian + 3;	X	X		X
20	if (trian == 0)				X
21	if (a + b <= c || a + c <= b || b + c <= a)	X	X	X	

Fig. 5. A table showing a manual slice of 11 lines of code in the Triangle.java file.

The above table shows a slice of several important lines regarding the calculation of the triangle type as well as the line including the given error in the Triangle.java class. A copy of the Triangle.java class and a full document with a complete slice of this class are also attached to this paper.

To effectively conduct a spectrum-slice based Software Fault Localization and analysis technique that we propose in this document, attention must first be given to the spectrum values shown in Figure 4 on the previous page. In this table, it becomes clear that the lines 13, 14, 16, 18, 19, and 20 have the highest suspiciousness score. Then, with those scores, the developer should observe the code on those lines. A pattern becomes clear that all of the lines with the highest suspiciousness score directly affect the value of the trian value (which is the primary method of computing the triangle type). From there, a developer should take a slice on the Trian variable, which is shown in Fig. 5 alongside the other variables in the table. For simplicity, the direct slice of the trian variable is shown below in Fig. 6. Notice that every line in this slice has a direct implication on the value of trian.

Ln	Code	A	B	C	Trian
10	int trian				X
13	trian = 0;				X
14	if (a == b)	X	X		X
15	trian = trian + 1;	X	X		X
16	if (a == c)	X		X	X
17	trian = trian + 2;	X		X	X
18	if (b == a) // bug	X	X		X
19	trian = trian + 3;	X	X		X

Fig. 6. A table showing the simplified manual slice from Fig. 5, excluding irrelevant lines.

The slice-based technique has now condensed our Triangle.java file from a total of 37 lines down to 8 lines of relevant interest, which is a reduction in # of LOC by about eighty percent. From here, identifying the possible bug becomes significantly easier, as the problematic area has already been pinpointed with GZoltar, and now the file has been shrunk in size to only include relevant lines. A careful eye would be able to observe that lines 14 and 18 fulfil the same purpose, as (a == b) and (b == a) are equivalent. At that point, it should become clear that the issue resides on line 18, and that a proper fix would be to change (b == a) to (b == c), which will then resolve the bug and ensure that the triangles are being calculated correctly. The objective of the slice-based segment is to minimize the surface of which the bug possibly appears in, utilizing the already narrowed scope from the GZoltar tool. By simply conducting a slice (that is without a spectrum analysis of the code), a slice would be more time consuming and much harder to analyze, as it would lack a direction or region with a primary focus, and a developer would have to conduct a slice on an entire software piece, rather than just a certain segment or area.

VI. SUMMARY

For our spectrum-slice based improvement method, we propose the following algorithm to ensure that the process is completed both accurately and efficiently, which will ensure the correct outcome throughout the process.

1. Run the code and ample number of test cases through GZoltar (via command line, ant-task, maven plugin, or through Eclipse) and record the top suspiciousness values.
2. Conduct a slice on every variable associated with the lines of code with the highest suspiciousness value (*S score*) from the previous step.
3. Observe possible room for errors and common mistakes such as bad loops, bad branching and logic predicates and conditions, or other mistakes that are easier to identify on a smaller segment of code compared to an entire file.
4. Repeat this process for each bug in a file. Multiple bugs could exist in a file if there are multiple areas with distinctly high *S* scores throughout the program (such as a high value, a large gap, and more high values).

VII. THREATS TO VALIDITY & FURTHER RESEARCH

There are several threats and factors that could possibly undermine our findings that a spectrum-sliced based method is more accurate than using one of the two test methods. To start off, we choose a relatively simple project with a single bug from the Defects4J library, which drastically limited the size of our data pool to sample from. This could be alleviated with further research utilizing more bugs from the Defects4J library, as it contains over 800 bugs. A larger data pool with less visible software defects would additionally help remove any possible bias or guesswork from the experiment. Another threat to validity emerges when considering that we use two isolated tools for data collection and gathering. An additional threat to validity is that our slice-based segment for the given bug was a manual slice rather than an automated dynamic slice which is better for accounting context changes [1]. Our team was unable to find a slicing tool as noted previously in this paper and resorted to conducting a manual slice. Manual slices are easy to do, however, this increases the possible room for error in our results and should be considered.

We recommend a tool be written to combine spectrum-based and slice-based testing, which would allow for easier yet more reliable and accurate data collection. Further research would prove the validity of such tool. Finally, differences amongst computer systems made installing and setting up the appropriate environments with the necessary software challenging, coupled with the lack of documentation available for the tools used throughout this paper. More cohesive documentation would be helpful for future research and would ensure accurate steps are being taken by everyone involved in the fault localization process. Despite these threats to validity, we believe that we present a valid case for future research, and that our proposal of a slice-based, spectrum hybrid localization method stands firm against the threats, and further research and data collection will support our findings.

VIII. CONCLUSION

To conclude our research and analysis to this process, we must first consider advantages and disadvantages to both Software Fault Localization techniques discussed, spectrum and slice based. Spectrum based testing is incredibly efficient and does a great job detecting locations of faults within software files, however, raw numbers, like *S scores* can be challenging to interpret and applied to the actual location of a bug in a piece of code [1]. On the other hand, slice-based testing is great for determining a trace of a variable or statement, however, tools are lacking to automate this process which can result in an increased margin of error and complicate the process, making it significantly more time consuming for developers and testers [1]. By combining the spectrum-sliced method that we proposed, we reduce the time needed to conduct a slice of a program, by narrowing the program scope before conducting the slice by using spectrum analysis of the code. By drastically reducing the space needed to conduct a manual slice (in our case, almost a 80% reduction), we were able to ensure that the manual slice was more accurate and manageable for a developer, resulting in quicker time to detect and patch the bug (or bugs) in a code segment. By merging the two localization techniques and combining them, we can drastically improve the software fault localization field.

More drastic improvements could be implemented with better tools to conduct spectrum-based and slice-based, and a combinatorial tool would be the most efficient and effective way to detect faults, while minimizing the room for human error and interaction. However, a tool does not currently exist to conduct this process. Additionally, there is no tool yet available that works for slicing Java files, as noted previously in this paper. The creation of either one of those tools would be instrumental in ensuring a correct and proper spectrum-slice based testing technique is available.

While more research should be conducted on this proposal, this team believes that the data presented in this paper, as well as our technique, are valid and worth exploring, with a starting point of creating a valid slice-based testing tool, then, by combining the newly created tool with spectrum functionality, to ensure a fully autonomous testing system for effective and efficient fault localization.

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu and F. Wotawa, "A Survey on Software Fault Localization," in *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707-740, 1 Aug. 2016, doi: 10.1109/TSE.2016.2521368.

[2] M. Weiser, "Program Slicing," in *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July 1984, doi: 10.1109/TSE.1984.5010248.

[3] R. Abreu, A. Gonzalez, P. Zoetewij and A. van Gemund, "Automatic software fault localization using generic program invariants", in *Proceedings of the 2008 ACM Symposium on Applied Computing*, pp. 712-717, 2008. doi: 10.1145/1363686.1363855.

[4] Vangipuram RadhaKrishna, "Design and Analysis of Novel Kernel Measure for Software Fault Localization," in *Proceedings of the International Conference on Engineering & MIS 2015 (ICEMIS '15)*. Vol. 15, no. 34, pp. 1-5, Aug. 2015, doi: 10.1145/2832987.2833042