Experimental Evaluation of Hybrid Algorithm in Spectrum based Fault Localization

A. Jonghee Park[1,2], **B. Jeongho Kim**[2], **and C. Eunseok Lee**[2]
[1]Samsung Electronics, Suwon, Gyeonggi-do, South Korea
[2]Sungkyunkwan University, Suwon, Gyeonggi-do, South Korea

Abstract— *During debugging process in software development cycle, fault localization is inevitable work. Diverse approaches have been proposed, such as program slicing, machine learning, and data mining for fault localization. In this paper we propose an effective hybrid fault localization algorithm based on a spectrum that enables fault detection in every statement. This algorithm distinguishes the location of a bug that causes a false positive score through the relationship between a test case and statement hit information. We also provide a fault localization tool named SKKU Fault localizer which enables source code instrumentation, test automation, test result comparison, extraction of distinct data, and fault ratio display. We applied it to the bug detection in Siemens test suite. Empirical results show that the hybrid algorithm not only decreases the amount of code to be reviewed by the programmer but also increases the effectiveness.*

Keywords: Program debugging, Spectrum based Fault Localization, Execution trace, Suspicious code, Fault localization

1. Introduction

Fault localization is time-consuming and costly, but, it is a very important task in the software development process. It is the hardest and most boring work for the programmer but it is essential work, needed to get rid of program bugs. To date, many studies have been carried out as people in the software industry have always been interested in fault localization. Firstly, program slicing [1] with a static and dynamic method was introduced. After that, various machine-learning methods were developed, such as artificial neuron network [2], SVM [3] and K-NN [4], data mining [5], and applied to the fault localization field. In addition, a major method called spectrum-based fault localization utilizes the relationship between the test result of a test case, and statement hit information. Tarantula [6], AMPLE [7], Jaccard [8], and Heuristic III [9] are representative algorithms in spectrum-based fault localization, which we will introduce in Section 2.

The main goal of this paper is to reduce the reading code coverage that the programmer should review, and effectively detect the exact location of any program bug. In addition, all processes are to be conducted with minimum human intervention. Then, we expect improved software product quality as well as reduced human workload in debugging activity. Therefore, we focused on spectrum-based fault localization which can provide suspiciousness that shows the probability of a program bug in every statement. In particular, we proposed a Hybrid fault localization technique which combines advantages from previous equations which were introduced by former researchers. In addition, we applied distinct data extraction technique to remove redundant duplicated data among test result.

The rest of this paper is as follows. In Section 2, we present some backgrounds and related work on spectrum-based fault localization. In Section 3, we propose methodology including the Hybrid technique and SKKU Fault localizer that we have developed for fault localization automation. In Section 4, we discuss the results of experimentation from two perspectives regarding reading code coverage and correctness of fault localization. Finally we concluded in Section 5, and presented future work.

2. Related work

Spectrum-based fault localization provides fault suspiciousness ratio by analyzing the relationship between a test result (pass or fail) and the visiting information of a statement. We assume that if failure test case happened, a fault exists among statements that were visited during a test in runtime. However, we cannot expect to determine the exact fault location only by the fail test case. Therefore, the pass test case was also utilized, to narrow down the fault statement.

Table 1. describes some notations that are commonly used in the fault localization field. h_i contains binary information as to whether the statement was visited or not. e_i contains binary information to describe the test result (pass or fail). If test case (T_i) which is one of the test cases in the test pool, was executed in runtime and the test result was fail, a certain statement (s_i) can be described as a_{11} (this line was visited) or a_{01} (not visited). In the same way, if the test result was pass, a certain statement (s_i) can be described as a_{10} (this line was visited) or a_{00} (not visited). Therefore, according to test result, every statement will be counted with four types of notation (a_{11}, a_{10}, a_{01}, a_{00}).

Representative equations have previously been introduced in spectrum-based fault localization methods, such as Tarantula, AMPLE, Jaccard, CBI [10], Ochiai [7] and Heuristic

Table 1: Relation between statement hit and test result

	Value		Description	
Notation	h_i	e_i	Statement hit	Test result
a_{11}	1	1	O	Fail
a_{10}	1	0	O	Pass
a_{01}	0	1	X	Fail
a_{00}	0	0	X	Pass

III. They calculate suspicious fault ratio in a different way, as below.

$$\text{Tarantula } S_j = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}} \quad (1)$$

J. A. Jones and M. J. Harrold (2005) developed Tarantula [6] which is aim to show suspiciousness in every statement. In addition, they conducted an experimental program which is based on C language.

$$\text{AMPLE } S_j = |\frac{a_{11}}{a_{01}+a_{11}} - \frac{a_{10}}{a_{00}+a_{10}}| \quad (2)$$

AMPLE [7] was developed to collect information about the hit spectra of method call sequences. Therefore, it is known to check faults in object-oriented language, such as Java and C++ language.

$$\text{Jaccard } S_j = \frac{a_{11}}{a_{11}+a_{01}+a_{10}} \quad (3)$$

Jaccard [8] developed from similarity coefficients in the mathematics field. It normally was used to find meaningful data among sets that consist of nominal elements. In particular, it was used in the Pinpoint framework.

$$\text{Heuristic III } S_j = [(1.0)*n_{f1}+(0.1)*n_{f2}+(0.01)*n_{f3}] - [(1.0)*n_{s1}+(0.1)*n_{s2}+\alpha*X_{(F/S)}*n_{s3}] \quad (4)$$

Wong et al. (2010) recently introduced Heuristic III [9]. They considered additional failed and passed test cases, and how they contribute to locating program fault. They divided failed and passed test cases into three groups (see more details in reference paper). In addition, they employed scaling factor α. Heuristic III (a), (b) and (c) were made according to the α value 0.01, 0.001 and 0.0001, respectively. Through various experiments, they presented Heuristic III (c) as the best equation.

3. Methodology

In spectrum-based fault localization field, there are approximately 30 types of existing equations [11][12]. We observed their behaviors and found out the characteristics of each algorithm. For the same software program, the suspicious fault ratio was different in every equation, according to their peculiarity. In this paper, we considered each equation's strength and finally proposed a Hybrid algorithm. To prove the proposed Hybrid equation, we developed the SKKU Fault localizer.

3.1 Hybrid algorithm

Hybrid algorithm is basis of two assumptions.

- *Assumption 1: Each algorithm has strength and weakness at the same time.*
  Therefore, once we get advantages of their algorithms, general outperformed Hybrid algorithm can be generated.
- *Assumption 2: To remove redundant duplicated test result makes better effectiveness of suspiciousness.*
  Test result data is huge and contains redundant data. Once we remove unnecessary data, it localizes exact bug location.

We figured out that Tarantula and Jaccard show zero suspiciousness when the numerator a_{11} was zero. This means that suspiciousness should be zero in statement (s_i), if there are no fail test cases. Otherwise, it shows suspiciousness over zero even if all test cases were hit. In addition, AMPLE described zero suspiciousness, if all test cases were visited in statement (s_i). We realized that this characteristics makes variation of fault localization effectiveness when we applied those algorithms to several test programs. Finally, we develop the Hybrid algorithm to combine the Tarantula, AMPLE, and Jaccard characteristics as follows.

$$\text{Hybrid } S_j = \begin{cases} \text{if } |\frac{a_{11}}{a_{01}+a_{11}} - \frac{a_{10}}{a_{00}+a_{10}}| = 0, 0 \\ \text{else} \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{11}}{a_{11}+a_{01}} + \frac{a_{10}}{a_{10}+a_{00}}} + \frac{a_{11}}{a_{11}+a_{01}+a_{10}} \end{cases} \quad (5)$$

We present the mid function in the example in Figure 1, which is well known for describing fault localization. There is a program bug in the 7th statement. Only three test cases hit that statement, in addition to one test case being failed, and the others being passed. This means that four types a_{11}, a_{10}, a_{01}, a_{00} mapped to 1, 2, 0, and 4. Based on the formula, Tarantula shows 0.84, and Hybrid indicates 1.33. In addition, Tarantula has six statements to be reviewed (over than zero suspicious fault ratio), and Hybrid has only three statements. It shows the Hybrid equation not only reduced the reading code coverage, but also detected the exact fault location in an early rank. Removing redundant data, such as T1, T2 (T7 is not redundancy cause test result was fail.) is important because redundant data affects suspiciousness. Only distinct data, such as T1 shall be used for fault localization.

3.2 SKKU Fault localizer tool

Software debugging is time-consuming work, where the programmer rectifies error; but it is essential work, in order to fulfill the required quality of software, and completeness of product. Therefore, those who work in Research and Development develop their own tools, or purchase commercial tools. However, those tools are not customized as much as they want, and even, require manual intervention from the user in many places, while they operate the tool. Therefore,

Source Code	Test Cases							Algorithm			
	T1	T2	T3	T4	T5	T6	T7				
mid() { int x,y,z,m;	3,3,5	4,4,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3	Tarantula	Rank	Hybrid	Rank
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	●	0.50	4	0.00	4
2: m = z;	●	●	●	●	●	●	●	0.50	4	0.00	4
3: if (y<z){	●	●	●	●	●	●	●	0.50	4	0.00	4
4: if (x<y)	●	●	●			●	●	0.63	3	0.88	3
5: m = y;			●					0.00	5	0.00	4
6: else if (x<z)	●	●				●	●	0.71	2	1.04	2
7: m = y; // *** bug ***	●	●					●	0.83	1	1.33	1
8: }else{				●	●			0.00	5	0.00	4
9: if (x>y)				●	●			0.00	5	0.00	4
10: m = y;				●				0.00	5	0.00	4
11: else if (x>z)					●			0.00	5	0.00	4
12: m = x;}								0.00	5	0.00	4
13: print("Middle number is:",m);	●	●	●	●	●	●	●	0.50	4	0.00	4
} Pass/Fail Status	P	P	P	P	P	P	F				
Redundant Test Case	Y	Y	N	N	N	N	N				

Fig. 1: Source code, test cases and suspiciousness statement (Mid function)

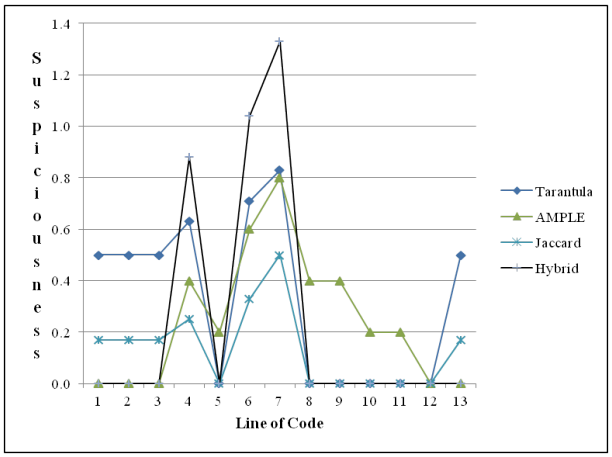

Fig. 2: Suspicious fault ratio of Talantula, AMPLE, Jaccard, and Hybrid

this paper introduces the SKKU Fault Localizer tool. If there is original source code, this tool can be applied to applications operated by Windows OS.

SKKU FL tool has many benefits, as follows. First, it saves time, because many procedures are automated, such as build source code, instrumentation, execution test, and visualization of suspicious fault ratio. Furthermore, it minimizes user invention, when the user utilizes the tool. Second, it reduces source codes that the developer needs to examine. Generally, when problem was happened, the developer debugs every statement that the program passed. However, SKKU FL tool executes many test cases, and automatically displays the most suspicious line. Then, the developer just checks the highly suspicious lines, which are shown with red background fill. Third, it supports a function to extract the test result as an Excel file. Then, it can be utilized in analyzing a test case and fault localization information data.

The SKKU FL tool GUI consists of an input space for source code, database file, answer sheet and test suite from the user, and displays space for test cases and test results of source code. It provides a function to extract Excel file that belongs to the display space (Test case, Source code, Result).

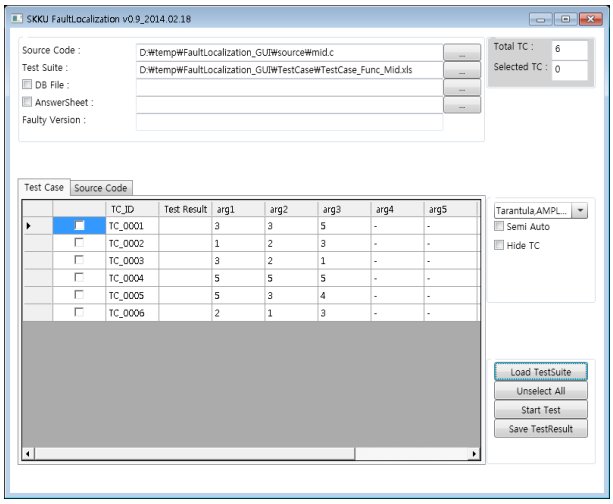

Fig. 3: Input screen : test case

- Input space: location of source code, database, answer sheet, test case, algorithm of suspicious fault ratio, start and load button
- Display space: view of test case, source code added suspicious fault ratio and rank

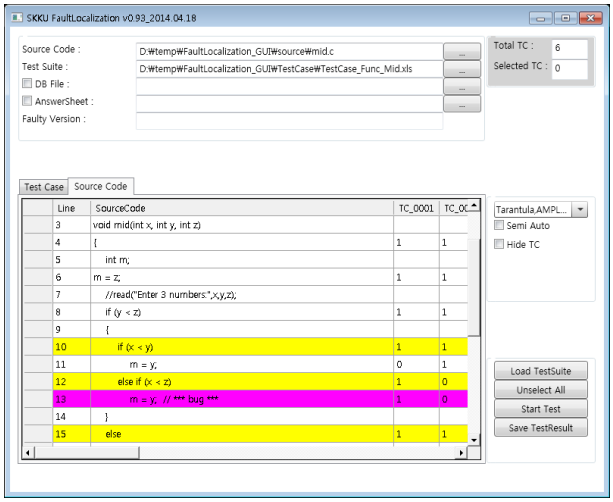

Fig. 4: Display screen : source code and suspicious fault ratio

In Figure 4, it displays line number, statements and passed information as well as colored suspicious fault ratio. Yellow means it is potential risky code, and red means it is highly risky code. As a result, it isolates the fault location, even guides the path where statements are passed, and reducing code lines unrelated to the fault. The functionality of SKKU FL tool has been continuously updated, in order to apply

Table 2: Test case format

TC_ID	Test result	arg1	arg2	arg3	Expected value	Actual value
TC_0001	Pass	3	3	5	3	3
TC_0002	Pass	1	2	3	2	2
TC_0003	Pass	3	2	1	2	2
TC_0004	Pass	5	5	5	5	5
TC_0005	Pass	5	3	4	4	4
TC_0006	Fail	2	1	3	2	1

various test programs, such as the Siemens test program, as well as to utilize database.

3.3 Test case design

The test case is managed in an Excel sheet, and it defines the TC_ID with a unique identifier, test result, arguments value, expected value, and actual value returned from the program. The test result and actual value are automatically inserted by SKKU FL tool, because it can decide pass and fail, by comparing the expected and actual value. If semi-auto is checked, the test result will be inserted by user decision.

3.4 Test process

The SKKU FL is operated in the following seven steps.

- Step 1: The user should select the test cases and source code to be examined. (This can be replaced with a database, if there is previous test case pool, such as test results and statement hits.)
- Step 2: Select an algorithm, such as Tarantula, AMPLE, Jaccard, Heuristic III (c), and Hybrid. If it is hard to reach a pass or fail verdict in the test result in the target application, check the Semi-Auto checkbox. Then it will ask for user input for the verdict after the test case is finished, in order to decide pass or fail.
- Step 3: Load the test case to the display screen.
- Step 4: Select as many test cases as you want to test, and click the Start button.
- Step 5: The tool instruments the original source code, and makes a binary file, through building an instrumented source code. After that, it passes parameters, in launching the binary file. Next, it monitors the test program running and recording line information, which is passed, until the program is terminated.
- Step 6: Compare the expected value with the actual value that the program returned. SKKU FL tool automatically decides pass or fail, if it does not require a user decision. Otherwise, it asks for user input, as to whether it is Pass or Fail.
- Step 7: SKKU FL tool computes suspicious fault ratio by using the line of statement passed and test result of pass or fail and finally displays suspicious fault ratio.

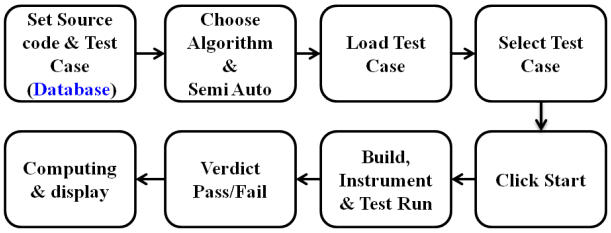

Fig. 5: Process of the SKKU Fault localizer tool

4. Experimental results

4.1 Test environment

To set up the test environment, two major procedures are required. The first procedure is that test case designer creates a test case file, as introduced in Section 3. When making the test case file, we used the Siemens input file that they provided as a text file. After that, all test cases are run using the original source code, in order to collect the correct test result, which will be used to obtain a pass or fail verdict test result.

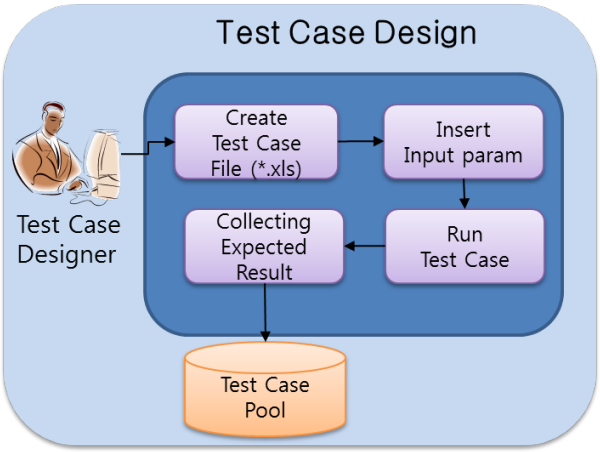

Fig. 6: Test case design architecture

The second procedure is that test executor runs SKKU Fault localizer, and collects the test result. We choose a faulty version source, and load the test case from the test case pool. After that, all test cases are run, and suspiciousness is obtained, using various algorithms.

4.2 Siemens test suite

We used 123 faulty versions in Siemens test suite seven test programs [13][14] among 132 faulty versions, which contain a single bug, seeded from the original non-faulty version. Test programs were designed to apply realistic software program commonly used in industry. We used only the available test programs. Therefore, we excluded 9 faulty versions: version 4 and 6 of printtokens because there is no

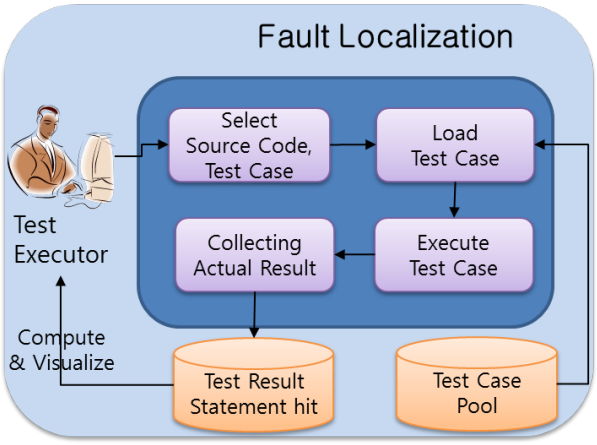

Fig. 7: Fault localization architecture

Table 3: Siemens test program

Program	Ver.	LOC	Test Cases	Description
printtokens	7	565	4140	lexical analyzer
printtokens2	10	510	4071	lexical analyzer
replace	32	563	5542	pattern recognition
schedule	9	412	2650	priority scheduler
schedule2	10	307	2680	priority scheduler
tcas	41	173	1578	altitude separation
totinfo	23	406	1054	information measure

change between the original and faulty version and version 6, 10, 19, and 21 of totinfo and version 38 of tcas and version 12 of replace because there is only change in declaration variable, such as define statement. We also exclude version 9 of schedule2 because there is no fail test case. Each faulty version was aimed at various human errors that they usually make in their work [15], such as missing code, wrong position of switch-case, or wrong boundary value.

4.3 Results and analysis

We have performed an experiment using the Siemens test suite. There are two aspects, when comparing the effectiveness regarding fault localization among equations. One is EXAM score which was proposed by Wong et al. (2010). EXAM score consists of Best and Worst. Best means that during review, the fault statement may be found first, between the same suspiciousness. Worst means that the fault may be found last. The other aspect is the RCC (Reading code coverage) that we proposed. This means that a statement that has a suspiciousness of over zero should be reviewed by the programmer. Of course, in both measurements, the score is better when it closes to zero.

A comparison of effectiveness between the previous studies and Hybrid is shown in Figure 8∼10. We choose Tarantula and Heuristic III (c), to avoid describing many

Table 4: Measurement method

Name	Formula	Description (lower is the best)
EXAM (Best)	B_r/Total line	B_r : Rank of Faulty line
EXAM (Worst)	W_r/Total line	W_r : Rank of faulty line + number of same rank lines - 1
RCC (Reading code coverage)	Z_+/Total line	Z_+ : number of lines over than zero suspicious ratio

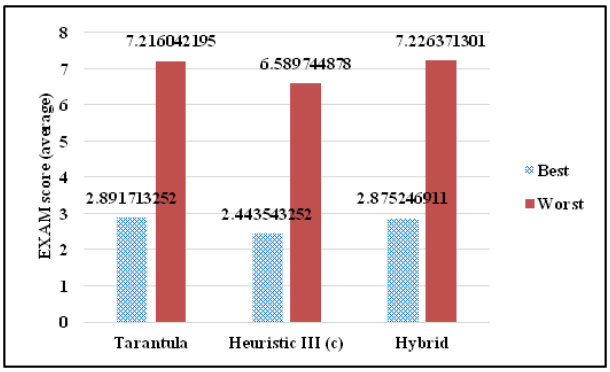

Fig. 8: Original EXAM score (average)

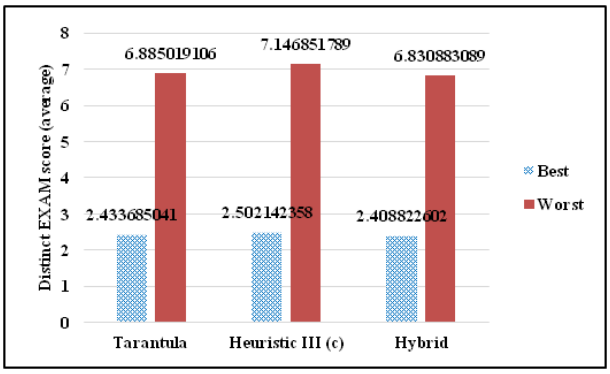

Fig. 9: Distinct EXAM score (average)

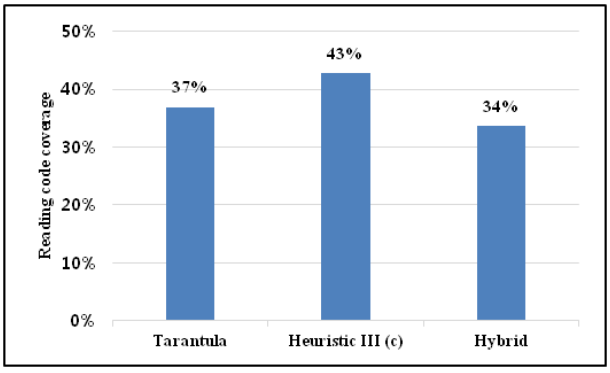

Fig. 10: RCC (Reading code coverage on average)

		Distinct test data									Notation				Algorithm	
Line	tcas v1 - SourceCode	T1	T2	T3	T4	T5	T6	T7	T8	T9	a00	a01	a10	a11	Hybrid	Rank
73	if (upward_preferred)	•		•	•	•	•	•	•		2	0	6	1	0.71429	4
*75	result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation > ALIM()))); /* c	•				•	•	•			5	0	3	1	0.97727	**2**
77	else	•				•	•	•			5	0	3	1	0.97727	2
79	result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());			•	•				•		5	1	3	0	0	6
81	return result;	•		•	•	•	•	•	•		2	0	6	1	0.71429	4
82	}	•		•	•	•	•	•	•		2	0	6	1	0.71429	4
...	...	•		•	•			•	•		4	0	4	1	0.86667	3
109	return (Other_Tracked_Alt < Own_Tracked_Alt);	•		•	•			•	•		4	0	4	1	0.86667	3
110	}	•		•	•			•	•		4	0	4	1	0.86667	3
133	else if (need_upward_RA)	•		•	•	•	•	•	•		2	0	6	1	0.71429	4
134	alt_sep = UPWARD_RA;	•				•					7	0	1	1	1.38889	1
135	else if (need_downward_RA)			•	•		•	•	•		3	1	5	0	0	6
136	alt_sep = DOWNWARD_RA;				•						7	1	1	0	0	6
137	else				•						7	1	1	0	0	6
138	alt_sep = UNRESOLVED;			•			•	•	•		4	1	4	0	0	6
142	}	•	•	•	•	•	•	•	•		1	0	7	1	0.65833	5
150	fprintf(stdout, "Error: Command line arguments are\n");									•	7	1	1	0	0	6
		F	P	P	P	P	P	P	P	P						

Fig. 11: Examaple of how Hybrid makes a distinct data (faulty version 1 of tcas)

previous algorithms. From Jones et al. (2005), the Tarantula method is more effective than set-union, set intersection and nearest-neighbor. Wong et al. (2010) presented Heuristic III (c) as being more effective than Tarantula in EXAM (Best) and EXAM (Worst). Through Figure 8: Original EXAM score, Heuristic III (c) outperforms Tarantula and Hybrid on average in both EXAM (Best) and EXAM (Worst). Hybrid only outperforms Tarantula in EXAM (Best). However, in the case of Figure 9: Distinct EXAM score, Hybrid is more effective than Tarantula and Heuristic III (c) in both EXAM (Best) and EXAM (Worst). Therefore, the Hybrid technique is more effective overall in EXAM score.

With respect to RCC, Hybrid always outperforms Tarantula and Heuristic III (c). In Figure 10, approximately 34% of the code has suspicious fault ratio. This means that when bug was found during testing the programmer should just review a third of the original source code. Hybrid decreased the reading code coverage to be checked.

4.4 Additional consideration

We found that there are similar test cases that contain the same test result, among thousands of test cases. To increase the effectiveness, we thought that distinct data (not a same statement hit information and test result) makes a better EXAM score, than the original experiment. In particular, faulty version 1 of tcas, there are 1578 test cases when calculating suspiciousness. Many redundant duplicated test data make lower suspiciousness. In Figure 11 shows that after applying distinct technique, test cases were reduced to only 9. Therefore, before calculating suspiciousness, except for unique one, the other test cases that have same statement information and result should be removed.

After removing duplicated data in the original data, Figure 12 indicates that most EXAM scores were decreased in every algorithm. In particular, Hybrid EXAM (Best) was improved by 17 percent over the original EXAM score.

Table 5: Comparison of EXAM score (average)

	Tarantula		Heuristic III (c)		Hybrid	
EXAM	Original	Distinct	Original	Distinct	Original	Distinct
Best	2.892	2.434	2.444	2.502	2.875	2.409
Worst	7.216	6.885	6.590	7.147	7.226	6.831

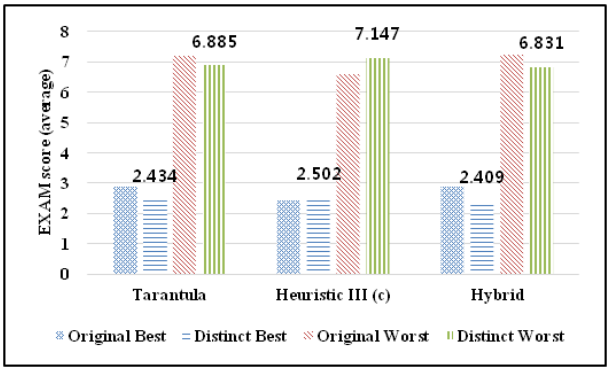

Fig. 12: Comparison of EXAM score between original and distinct data

5. Conclusion and future works

We proposed Hybrid algorithm and developed SKKU Fault localizer tool, in order to not only localize the exact fault location, but also to reduce the reading code coverage. Hybrid was proposed to overcome each characteristic of equation characteristic because there was no superior algorithm in every test program. Hybrid shows that it was more effective generally among existing algorithms. We presented distinct test data to clarify unique test data and get rid of redundant test data which degrades performance of algorithm performance. Furthermore, SKKU Fault localizer can help people to automatically perform all test procedures. In addition, it is easy to install and simple to operate the

GUI, because it is based on Windows OS.

For future work, we would like to extend the Hybrid algorithm to all of the Unix test suite, as well as a large-scale real world program. At the same time, we will analyze the program characteristics, in order to improve Hybrid algorithm according to the program. In addition, we figured out that distinct test data makes the algorithm better. However, it is still time-consuming and burdensome work for the software quality assurance department in a company. People who are in charge of software quality would like to do regression test, to avoid reproducing the same problem. They really want to reduce the amount of test cases that have the same capability as the original test cases. Therefore, we will do research regarding the reduction of test case.

6. Acknowledgment

This research was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MEST)(No. 2012008240)

References

[1] Agrawal, Hiraral, et al. "Fault localization using execution slices and dataflow tests." *Proceedings of IEEE Software Reliability Engineering* (1995): 143-151.

[2] Wong, W. Eric, and Yu Qi. "BP neural network-based effective fault localization." *International Journal of Software Engineering and Knowledge Engineering* 19.04 (2009): 573-597.

[3] Ascari, L. C., et al. "Exploring machine learning techniques for fault localization." *Test Workshop, 2009. LATW'09. 10th Latin American.* IEEE, 2009.

[4] Renieres, Manos, and Steven P. Reiss. "Fault localization with nearest neighbor queries." *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on.* IEEE, 2003.

[5] Nessa, Syeda, et al. "Software fault localization using N-gram analysis." *Wireless Algorithms, Systems, and Applications.* Springer Berlin Heidelberg, 2008. 548-559.

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

[7] Abreu, Rui, Peter Zoeteweij, and Arjan JC Van Gemund. "An evaluation of similarity coefficients for software fault localization." *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on.* IEEE, 2006.

[8] Chen, Mike Y., et al. "Pinpoint: Problem determination in large, dynamic internet services." *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on.* IEEE, 2002.

[9] Eric Wong, W., Vidroha Debroy, and Byoungju Choi. "A family of code coverage-based heuristics for effective fault localization." *Journal of Systems and Software* 83.2 (2010): 188-208.

[10] Liblit, Ben, et al. "Scalable statistical bug isolation." *ACM SIGPLAN Notices.* Vol. 40. No. 6. ACM, 2005.

[11] Wong, W. Eric, and Vidroha Debroy. "A survey of software fault localization." *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45-09* (2009).

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

[13] Do, Hyunsook, Sebastian Elbaum, and Gregg Rothermel. "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering* 10.4 (2005): 405-435.

[14] The Siemens test suite. SIR (Subject Infrastructure Repository) website. [Online]. Available: http://sir.unl.edu/portal/index.php

[15] Hutchins, Monica, et al. "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria." *Proceedings of the 16th international conference on Software engineering.* IEEE Computer Society Press, 1994.