LETTER

Spectrum-Based Fault Localization Framework to Support Fault Understanding

Yong WANG[†,††,†††a)], **Zhiqiu HUANG**[††,†††], **Yong LI**[††], **RongCun WANG**[††††], ***Nonmembers***, ***and*** **Qiao YU**[††††], ***Member***

SUMMARY A spectrum-based fault localization technique (*SBFL*), which identifies fault location(s) in a buggy program by comparing the execution statistics of the program spectra of passed executions and failed executions, is a popular automatic debugging technique. However, the usefulness of *SBFL* is mainly affected by the following two factors: accuracy and fault understanding in reality. To solve this issue, we propose a *SBFL* framework to support fault understanding. In the framework, we firstly localize a suspicious fault module to start debugging and then generate a weighted fault propagation graph (*WFPG*) for the hypothesis fault module, which weights the suspiciousness for the nodes to further perform block-level fault localization. In order to evaluate the proposed framework, we conduct a controlled experiment to compare two different module-level *SBFL* approaches and validate the effectiveness of *WFPG*. According to our preliminary experiments, the results are promising.

key words: *fault understanding, spectrum-based fault localization, method- and block-level debugging*

1. Introduction

Spectrum-based fault localization (*SBFL*) has been shown to be a very practical and efficient strategy, which uses instrumentation with a low computational overhead and a good scalability to produce good results in large codebases [1]. In this field, researchers have been mainly focused on proposing new techniques and investigating their performance without considering human factors [2], [3]. Some studies show that two factors, accuracy and fault understanding, mainly affect the usefulness of *SBFL* techniques [3], [4], [6].

Lucia et al. [5] observed that many real-life bugs span across multiple lines. Parnin and Orso pointed out that "perfect bug detection" is generally unrealistic [6]. Therefore, giving a suspicious statement to directly localize the root cause of failures is unrealistic. Generally, module design should enforce adherence to the principle of high cohesion and low coupling, and a control flow graph for a module could easily aid programmers in performing program understanding. Therefore, in this work, we propose a novel *SBFL* framework, which includes the following two phases: localizing module-level fault based on *SBFL* and visualizing a weighted fault propagation graph (*WFPG*) for a hypothesis fault module to be checked. The two main factors that influence the effectiveness of the proposed approach include (1) accurate module-level fault localization and (2) effective *WFPG*.

Considering the cost of fault localization, here, we only focus on *SBFL* techniques. Those heavy-weight fault localization techniques, such as [8], are also easily integrated into the proposed fault localization framework. For those *SBFL* techniques, selecting the best profiling type and granularity level is problematic when *SBFL* comes to operationalization in industry [9], which is mainly factor to affect accuracy for module-level fault localization. In our framework, programmers use the node weighted information to perform a search-based fault localization and understand the program through *WFPG*. It is difficult to directly compare the two techniques. We turn the research question of effective *WFPG* into answering whether *WFPG* is more effective than *SBFL* to improve block-level fault absolute ranking. Therefore, an empirical study was conducted to answer the following two main research questions:

- *RQ1:* What is the best profiling type and granularity level to locate module-level faults using SBFL techniques?
- *RQ2:* Will *WFPG* effectively improve block-level fault absolute ranking?

2. Our Approach

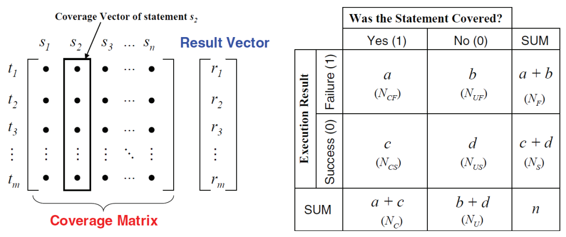
2.1 Module-Level Fault Localization

We propose two approaches to localize potential fault modules based on module-hit (called *SFL-m*) and block-hit (called *SFL-b*). The two hit spectra can be mapped as *Coverage Matrix A*, which is a binary $N \times M$ matrix, where N indicates the number of successful/failed executions, and M indicates the number of the instrumented program entities of the buggy program. The result of each execution can be viewed as N-length *Result Vector e*, in which '0' indicates successful, and '1' indicates failed. The pair (A, e) serves as an input for *SBFL*. The *coverage matrix* and *result vector*

Manuscript received November 8, 2018.
Manuscript publicized January 15, 2019.

[†]The author is with School of Computer and Information, Anhui Polytechnic University, Wuhu, China.
[††]The authors are with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China.
[†††]The authors are with Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, China.
[††††]The authors are with School of Computer Science and Technology, China University of Mining and Technology, XuZhou, China.
a) E-mail: yongwang@ahpu.edu.cn

DOI: 10.1587/transinf.2018EDL8233


Coverage Vector of statement s_2
s_1 s_2 s_3 … s_n
Result Vector
t_1 t_2 t_3 ⋮ t_m
r_1 r_2 r_3 ⋮ r_m
Coverage Matrix

		Was the Statement Covered?		
		Yes (1)	No (0)	SUM
Execution Result	Failure (1)	a (N_{CF})	b (N_{UF})	$a+b$ (N_F)
	Success (0)	c (N_{CS})	d (N_{US})	$c+d$ (N_S)
SUM		$a+c$ (N_C)	$b+d$ (N_U)	n

Fig. 1 Coverage matrix and result error.

are shown in Fig. 1.

Note that the quantities in the right of Fig. 1 have been annotated with fault localization-specific terms, which can be used directly to calculate the suspiciousness score for each program entity. For program entity j, N_{PQ} is defined as:

$$N_{PQ}(j) = |\{i|a_{ij} = P \wedge e_i = Q\}| \tag{1}$$

where N_{PQ} is the number of runs in which the entity j has been covered during execution ('C') or not covered during execution ('U'), and where the runs failed ('F') or successful ('S'). For instance, N_{CF} counts the number of times program entity j has been covered in failed executions. Those suspicious metrics are all defined based on the four values N_{CF}, N_{UF}, N_{CS}, and N_{US} for each program entity. In this paper, we choose *Dstar*, which is a commonly chosen suspicious metric. The formula of *Dstar* is shown as Formula (2), which is detailed in [1].

$$S_{Dstar}(j) = \frac{(n_{CF}(j))^*}{n_{UF}(j) + n_{CS}(j)} \tag{2}$$

For *SFL-m*, we can choose a metric to calculate suspiciousness for modules directly. However, we cannot measure the suspiciousness based on *SFL-b* directly. Similar to [14], for *SFL-b*, we use the block suspiciousness scores to generate the suspiciousness. The idea behind this is that because a fault block is contained by its module, the higher the maximum of the block suspiciousness score in a module is, the higher the module suspiciousness score. We formally define this concept as Formula (3):

$$S(M_i) = max\{S(b_j)|b_j \in M_i\} \tag{3}$$

In Formula (3), $S(M_i)$ is the suspiciousness score for module M_i, and $S(b_j)$ is the suspiciousness score for block b_j contained by M_i.

2.2 Weighted Fault Propagation Graph

For a hypothesis fault module m_i, a weighted fault propagation graph ($WFPG$) is defined as $WG_b(N, E, W_N, E_N, m_i)$, where N and E are the set of *nodes* and *edges*, respectively. W_N is weighted information about a node N, W_E is weighted information about an edge E, and m_i is a hypothesis fault module that a programmer wants to verify.

It is generally known that a fault within a basic block may propagate a series of infected program states before it manifests a failure. That is to say, even if every failed run covers a basic block, the *basic block* is not necessarily the root cause of failures. Inspired by [12], for an edge e_i, the higher the suspiciousness score is, the more suspicious the propagation of program states, and the more suspicious the propagation of the program state by e_i is deemed to be. Therefore, we can generate a $WFPG$ to express fault propagation based on this idea.

We first calculate the weighted information for edges using Formula (4), where N_{CF} is the number of edge $<b_i, b_j>$ covered in failed runs, N_{CS} is the number of edge $<b_i, b_j>$ covered in passed runs, and N_F and N_S are the number of failed runs and passed runs, respectively. The intuition is that weighted information for an edge is proportional to the number of failed executions and inversely proportional to the number of passed executions that covered it. After executing block b_i, the execution may transfer control to one of its successor blocks. Suppose b_k is a successor block of b_i. The program states of b_i may be propagated to b_k by *edge*$<b_i, b_k>$. We approximate the expected number of infected program states of b_i observed at b_k as the fraction of the suspiciousness score of b_j from that of b_k. Therefore, we use Formula (5) to model the portion of the contribution of *edge*$<b_i, b_k>$ with respect to the total contribution by all incoming edges of b_k. In general, a block b_j may have any number of successor blocks. Therefore, we sum those fractions from every successor block of b_j to generate weight information of node b_j. The weighted score of node b_j is defined as Formula (6).

$$W_{edge}(b_i, b_j) = \frac{\frac{N_{CF}}{N_F} - \frac{N_{CS}}{N_S}}{\frac{N_{CF}}{N_F} + \frac{N_{CS}}{N_S}} \tag{4}$$

$$C(b_i, b_k) = \frac{W_{edge}(b_i, b_k)}{\sum\limits_{\forall edge(*,b_k)} [W_{edge}(*, b_k)]} \tag{5}$$

$$W_{node}(b_j) = \sum_{\forall edge(b_j,b_k)} [W_{node}(b_k) \cdot C(b_i, b_k)] \tag{6}$$

We normally calculate a block suspiciousness score via its successor blocks. However, a module also has some *exit* nodes such as statement *return*, *exit*, etc. Those nodes have no successor blocks in the interprocedure. Thus, Formula (6) cannot be used to calculate the weighted information. In this way, Formula (7) is used to calculate the weighted information.

$$W_{node}(b_j) = \sum_{\forall edge(*,b_j)} [W_{edge}(*, b_j)] \tag{7}$$

We construct an equation set for all nodes and solve the equations to generate weighted information for each node. Considering that some weighted values are negative, these values are mapped to $[0, 1]$ in order to easily visualize the results.

3. Experiment

In our experiment, three Siemens programs, *print_tokens2*,

Table 1 Subjects used for empirical studies.

Program	LOC	BBs	Versions	Modules	Test cases
print_token2	399	135	9	17	4115
replace	512	153	26	21	5542
schedule	292	73	9	17	2650

Table 2 Comparing effectiveness between *SFL-b* and *SFL-m*.

	SFL-b		*SFL-m*	
Absolute ranking	numbers	percentage	numbers	percentage
top1	9	20.5%	12	27.3%
top2-3	17	38.6%	5	11.4%
top4-5	5	11.4%	6	13.6%
top6-10	6	13.6%	11	25.0%
others	7	15.9%	10	22.7%

replace, and *schedule*, containing more modules, are used to evaluate the effectiveness of the proposed approach. Table 1 shows the characteristics of those programs.

3.1 Answer RQ1

In order to answer RQ1, we perform an experiment to compare *SFL-b* with *SFL-m* in terms of effectiveness. We choose *Dstar* [1], one popular *SBFL*, as the baseline. According to recent work by Xia et al. [4], programmers inspect the first several program entities outputted by a fault localization tool in sequence. Therefore, we use *absolute ranking* to evaluate the proposed fault module localization approach. For example, an *absolute ranking* range of *top3-5* implies the root faulty module ranked at *top 3* to *top 5* position in the fault ranking list. We give the number of program versions in the *absolute ranking* range to compare the two approaches.

Table 2 shows the result of the study. The *number* and *percentage* columns show the number of buggy program versions ranked in the range of absolute ranking, and its percentage for *SFL-b* and *SFL-m*. For example, *SFL-b* ranks 9 root faulty modules at the top, and its percentage is 20.5%; *SFL-m* ranks 12 root faulty modules at the top, and its percentage is 27.3%. In this case, *SFL-m* outperforms *SFL-b*. However, the number of *SFL-b* ranked at *top 1-3* and *top 1-5* by far exceeded the results of *SFL-m*. We conjectured that this result is because the *module-hit* program spectrum is coarse-grained, which would not reflect the $Fault\&Failure$ model of the fault. That is to say, *SFL-b* is effective to localize module-level faults. We could suppose the approach is effective if root cause is ranked in *top k* (*k* could be set as 3). Therefore, as a whole, *SFL-b* is better than *SFL-m*. The result may inspire future study on module-level fault localization based on *SBFL* techniques.

3.2 Answer RQ2

To answer RQ2, the node weighted information of *WFPG* was used as the suspiciousness score to generate a fault ranking list and observe whether the weighted information can further improve the fault absolute ranking. The fault ranking list was generated based on *WFPG*, called *WFPG-w*. To do this, *Dstar* and *WFPG-w* are run to generate

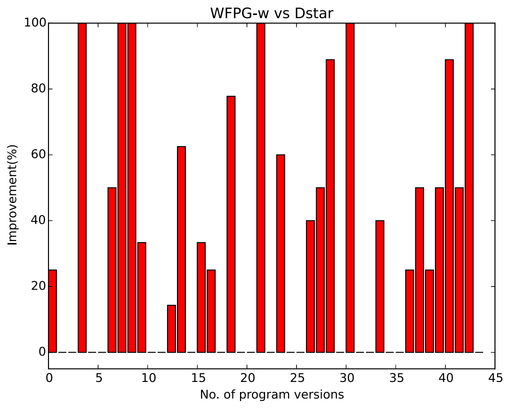


Fig. 2 Comparison of *WFPG-w* and *Dstar*.

a block-level fault ranking list. An *improvement metric* was used to estimate the approach improvement [10]. The *improvement metric* is defined as Formula (8).

$$Impr_{WFPG}^{SBFL}(A,B) = \begin{cases} 0, \ if\ B = 0\ and\ A = 0 \\ -100\%, if\ B = 0\ and\ A > 0 \\ \frac{B-A}{B} \times 100\%,\ Otherwise \end{cases} \quad (8)$$

Figure 2 shows the results of the study. There exist 11 fault versions ranked in the *top 1* in the 44 fault versions. Those program versions cannot be further improved; thus, they are ignored. The results show that the proposed approach improved over the *Dstar* for 26 versions and performs the same as *Dstar* for 7 versions, which indicates an effectiveness rate of 75.8%. The results show that no fault versions perform worse than *Dstar*. Therefore, the weighted information of *WFPG* is effective to guide programmers to localize block-level faults in practice. In those 26 versions, the average improvement is 59.6%, and it is worth mentioning that 6 root faults were ranked at the top of the fault ranking list.

3.3 Experimental Limitations

The major limitation about the experiments is whether the technique can scale to large real programs. In the preliminary experiments, we use a subset of the Simiens Programs. As well as existing most experiments in fault localization community, those programs are all small programs contained artificial faults, which are generated either by mutation tools or manually [15]. Therefore, the result of preliminary experiments only gives indications that the technique can be useful.

4. Related Works

Various approaches already exist to locate module-level faults. For instance, Shu [8] proposed and evaluated a novel *method-level* statistical fault localization technique, called *MFL*, which is based on causal inference. However, those techniques are all heavy-weight fault localization techniques. Considering the cost of fault localization, we focus on *SBFL* techniques for method-level faults. In [14], De Souza et al. provided a search roadmap for code hierarchy (CH) and integration coverage-based debugging (ICD) techniques, which can guide a programmer toward faults. In order to aid programmers in understanding software, many graph models were proposed including *WSDG* [7] and *PPDG* [11]. In [13], Gouveia et al. proposed three dynamic graphical forms using HTML5 to display the diagnostic reports yielded by *SBFL*. Different from their work, the proposed *WFPG* is a fault weighted propagation graph based on control flow graph, which is commonly used for programmers in the software community. The proposed *WFPG* is a graph for a specific hypothesis fault module, which is better understood by programmers.

5. Conclusion and Future Works

In this paper, a *SBFL* framework is proposed to expedite software debugging. Compared with previous studies, the proposed approach locates bugs through two-level fault localization, which reduces interference between the mechanism of *SBFL* and the actual assistance needed by programmers in debugging.

As future work, we plan to investigate more empirical studies to further evaluate the effectiveness of our approach in large real programs. We would also like to build an oracle that can predict the effectiveness of a module-level fault localization technique using *SBFL*. Based on the result, we can perform our framework effectively if the module-level *SBFL* is effective for a buggy program.

Acknowledgments

This work was supported by the Anhui University Natural Science Fund Key Project (Grant Nos. KJ2018A0116, KJ2016A252, and KJ2017A104), the National Key Research and Development Program (Grant No. 2016YFB100), Project 61772270, 61562087 supported by NSFC of China, the Anhui Natural Science Foundation (Grant No. 1608085MF147), and the Safety-Critical Software Key Laboratory Research Program (Grant No. NJ2018014).

References

[1] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," IEEE Trans. Softw. Eng., vol.42, no.8, pp.707–740, 2016.
[2] R. Minelli, A. Mocci, M. Lanza, and T. Kobayashi, "Quantifying program comprehension with interaction data," 14th International Conference on Quality Software (QSIC), pp.276–285, IEEE, 2014.
[3] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," ICSE2016, pp.808–819, 2016.
[4] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," ICSME, pp.267–278, 2016.
[5] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?," MSR2012, pp.74–77, 2012.
[6] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," ISSTA, pp.199–209, 2011.
[7] F. Deng and J.A. Jones, "Weighted system dependence graph," ICST2012, pp.380–389, 2012.
[8] G. Shu, B. Sun, A. Podgurski, and F. Cao, "MFL: Method-level fault localization with causal inference," ICST, pp.124–133, IEEE, 2013.
[9] M. Golagha and A. Pretschner, "Challenges of operationalizing spectrum-based fault localization from a data-centric perspective," Proc. Software Testing, Verification and Validation Workshops (ICSTW), pp.379–381, IEEE, 2017.
[10] Y. Wang, Z. Huang, Y. Li, and B. Fang, "Lightweight fault localization combined with fault context to improve fault absolute rank," Science China Information Sciences, vol.60, no.9, 092113, 2017.
[11] G.K. Baah, A. Podgurski, and M.J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," IEEE Trans. Softw. Eng., vol.36, no.4, pp.528–545, 2010.
[12] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.43–52, ACM, 2009.
[13] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," 2013 First IEEE Working Conference on Software Visualization (VISSOFT), pp.1–10, IEEE, 2013.
[14] H.A. de Souza, D. Mutti, M.L. Chaim, and F. Kon, "Contextualizing spectrum-based fault localization," Information and Software Technology, vol.94, pp.245–261, 2018.
[15] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," Proc. 39th International Conference on Software Engineering, pp.609–620, IEEE Press, 2017.