Variability Fault Localization: A Benchmark

Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo
{tuanngokien,trang.nguyen,sonnguyen,hieuvd}@vnu.edu.vn
Software Engineering Department, VNU University of Engineering and Technology, Vietnam

ABSTRACT

Software fault localization is one of the most expensive, tedious, and time-consuming activities in program debugging. This activity becomes even much more challenging in Software Product Line (SPL) systems due to the variability of failures in SPL systems. These unexpected behaviors are caused by variability faults which can only be exposed under some combinations of system features. Although localizing bugs in non-configurable code has been investigated in-depth, variability fault localization in SPL systems still remains mostly unexplored. To approach this challenge, we propose a benchmark for variability fault localization with a large set of 1,773 buggy versions of six SPL systems and baseline variability fault localization performance results. Our hope is to engage the community to propose new and better approaches to the problem of variability fault localization in SPL systems.

CCS CONCEPTS

• **Software and its engineering → Software product lines**.

KEYWORDS

variability bug, variability fault localization, benchmark

ACM Reference Format:
Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. 2021. Variability Fault Localization: A Benchmark. In *Proceedings of 25th ACM International Systems and Software Product Lines Conference (SPLC'21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A Software Product Line (SPL) is the highly configurable system that enables developers to tailor the family of products from reusable software assets [5]. This can be done by offering numerous *features* controlled by *options*. In Linux Kernel, there are +12K features, which is able to generate billions of scenarios. Basically, a *feature* is defined as a unit of functionality additional to the *base software*. A set of *selections* of all the features (*configurations*) defines a *product*. The presence or absence of some features might require or preclude other features. Feature dependencies are specified in a *feature model* which constraints over features and defines valid configurations [5].

In practice, to verify an SPL system, a subset of all of its valid products is selected for testing. In order to systematically test a system, various configuration sampling strategies have been proposed. Some popular sampling algorithms such as Combinatorial Interaction Testing [12], One-enabled [2], and One-disabled [2] can be used for the configuration selection process. For each selected product, a test suite is generated to verify its behaviors.

However, the variability that is inherent to SPL systems challenges quality assurance (QA) [11, 18, 21]. In comparison with non-configurable code, finding bugs through testing in SPL systems is more problematic as a bug can be *variable* (so-called *variability bug*), which can only be exposed under some combinations of features [11, 19]. In other words, a system contains variability bugs if among the sampled products, some products pass all their tests while the others fail. Hence, the buggy statements can only expose their bugginess in some particular products, yet cannot in others.

Despite the importance of finding variability bugs, the existing fault localization (FL) approaches are still limited. It is because these techniques are designed to find bugs in a particular product. To isolate the bugs causing failures in multiple products of an SPL system, the slice-based methods [25] could be used to identify the failure-related slices for each product independently of others. Consequently, there are a large number of statements in the whole system that need to be examined to find the bugs. This makes the slice-based methods become more impractical in SPL systems [25].

In addition, the state-of-the-art FL technique, Spectrum-Based Fault Localization (SBFL) [4, 14, 22] cannot be directly applied for locating variability bugs. Indeed, SBFL assigns the suspiciousness scores to the statements based on the test execution information of each product independently of the others. For each product, it produces a ranked list of statements. As a result, there are multiple ranked lists for a single system which is failed by variability bugs. From these lists, developers cannot determine the right starting point to diagnose the root causes of the system failures.

A naive solution to adapt SBFL for variability bugs in a system is that one can treat the whole system as a non-configurable code. This can be done by refactoring the mechanism controlling features in the system (e.g., `#ifdef`) to the corresponding `if-then` statements. By this adaptation, for a faulty system, a single ranked list of the suspicious statements can be produced according to their suspiciousness scores. The score of each statement is measured based on the total number of the passing and failing tests executed by the statement in all the products. However, this adaptation has two key problems which are caused by the incompatibility in testing of the different products. First, *in an SPL system, since the roles of a statement in different products are different, the statement behaves and is expected to behave differently in these products.* Hence, the tests in a product, which are designed to verify the behaviors of the specific statements in this product, could not be used to verify the behaviors of those statements yet in another product. Consequently, counting all the tests in the different products to measure the suspiciousness of a statement could cause inaccurate assessments. Secondly, *for a*

SPLC'21, 06–11 September, 2021, Leicester, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

product having more tests, it will have more impact on the suspiciousness scores of the statements in the whole system. This bias increases more clearly for the cases when the numbers of tests in the products are significantly different. It is because the total number of the tests used to measure the suspiciousness of statements in the system is counted from all the products. As a result, the suspiciousness of a statement in the whole system might not be holistically measured. Thus, these two problems of this adaptation might cause the ineffectiveness of SBFL in localizing variability bugs.

To encourage the researchers and practitioners to propose better solutions for variability fault localization, we contribute a dataset including 1,773 buggy versions of 6 SPL systems with extensive test suites. In this dataset, there are 339 versions contain a single bug each and 1,434 versions contain multiple bugs. The proposed techniques should be evaluated using the following standard metrics: *Rank*, *EXAM* [24], *Recall at Top-N* [15], and *PBL* [14], which are widely applied in the existing FL studies [4, 14, 22, 24, 25].

> *Variability Fault Localization Challenge*: Given a faulty SPL system containing variability bug(s) and a set of its sampled products with the test suites and test execution data, participants must propose new FL techniques to locate the buggy statement(s) in the system. The proposed techniques must be better than our baseline on the standard metrics in FL.

Our benchmark can be found at:

> https://tuanngokien.github.io/splc2021/

2 A DATASET OF VARIABILITY FAULTS

Since constructing a dataset of the real variability bugs with corresponding tests is a considerable task, until now there is no such public dataset. The available datasets of variability bugs [2, 3, 20] often lack of test information. Thus, they cannot be used to evaluate FL techniques which require test execution information. Additionally, Just et al. [13] has shown that the performance of FL techniques on real bugs can be estimated based on their performance on artificial bugs. That motivates us to propose a dataset of the artificial variability bugs with the corresponding tests which are systematically generated. Furthermore, we categorize our dataset by different dimensions of bugs. Based on that, the proposed FL techniques could be evaluated in some different circumstances.

2.1 Subject Systems

We collected 6 Java SPL systems in *SPL2Go* [1], which are widely used in the existing studies about configurable code [7, 8, 19], to construct our dataset (Table 1). In addition, the products of each system are composed by *FeatureHouse* [7], a popular automated software composer. Indeed, there are some other systems in *SPL2Go*, but they raise some errors while composing and compiling their products. Thus, they cannot be used in our dataset.

2.2 Single Variability Bug Generation

We design a process to systematically generate variability bugs, including three main steps as shown in Figure 1: *Product Sampling* and *Test Generating*, *Bug Seeding*, and *Variability Bug Verifying*.

Step 1 - Product Sampling and Test Generating. Firstly, for an SPL system, a set of the products of the system is systematically sampled by the existing techniques [12, 18]. Particularly, we sample a set of valid configurations based on the system's feature model with 4-wise coverage by using *SPLCA* [12]. For each configuration, a corresponding product is composed from the implementation of all the enabled features by using *FeatureHouse* [7]. For each product p, a test suite is automatically generated using an existing test generation technique, *Evosuite* [10], to capture the original behaviors of p. For each test of p, the output will be recorded and used as the test oracle of the corresponding test in the product having the same configuration as p in the mutated system in *Step 2*.

Step 2 - Bug Seeding. To inject a fault into an SPL system, we randomly apply a modification to the original source code of the system by using a mutation operator. In essence, the operator changes the original behaviors of several sampled products. The changes are expected to be captured by the products' generated tests. In other words, these products produce output that is different from the output recorded in *Step 1*. Particularly, we use *μJava* tool [17] to create mutants at the statement-level of the system's source code. We do not apply any operator which deletes a whole code statement. Since when the whole statement is removed from the original code, the buggy statement which is expected to be localized by a FL technique might not be determined. Hence, we only use the operators modifying or deleting a part of a statement.

Step 3 - Variability Bug Verifying. In this step, we verify each generated bug to ensure that the fault is a variability bug and caught by the tests. Particularly, for each mutated system, the collection of products, which are corresponding to the same set of the configurations sampled in *Step 1*, are composed. After that, we run each of these products against its generated test suite. A product is considered as a *passing product* if it passes all the tests. In contrast, a product, which fails at least one test, is classified as a *failing product. A bug is considered as a variability bug if it causes failures in certain sampled products*. In other words, after testing, among the sampled products, there are both passing and failing products. Besides, during the testing process, test execution information will be recorded by a code coverage tool, *OpenClover* [2].

2.3 Multiple Variability Bugs Generation

For a more challenging setting, we create a dataset of the buggy systems which contain multiple variability bugs. Particularly, we extend *Step 2* and *Step 3* in Figure 1 to generate such dataset:

Multiple Bugs Seeding. In *Step 2*, instead of applying a single modification, a random number, $n > 1$, of mutation operators are continuously used to mutate the source code of an SPL system.

Variability Bugs Verifying. In *Step 3*, all the bugs need to be verified to ensure that they are variability bugs. Particularly, *when one or more bugs are fixed, the remaining bugs still cause failures in certain products*. In other words, unless all the bugs are fixed, there still exists both the passing and failing products. We aim to simulate the bug-fixing process in practice. For each case, we gradually fix each bug by reverting the modification to the original state and do regression testing. The case would be accepted if there still exists both the passing and failing products until all the bugs are fixed.

[1] http://spl2go.cs.ovgu.de/

[2] https://openclover.org

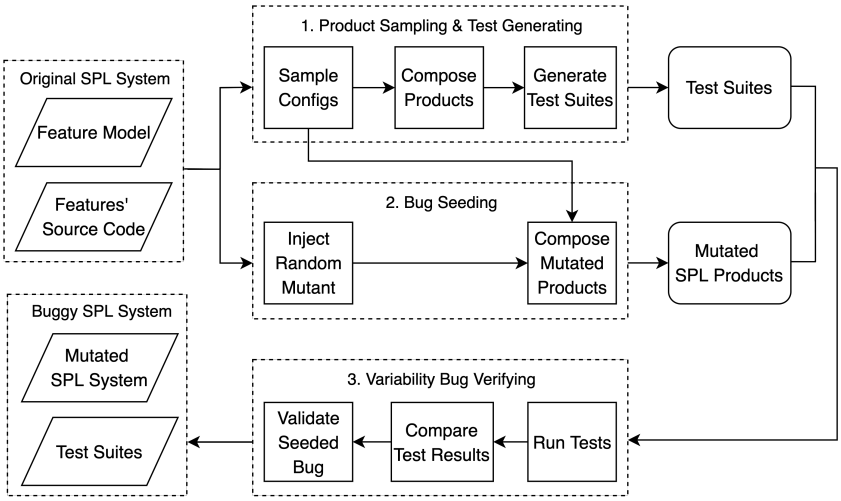

Figure 1: Bug-Generating Process Overview

Table 1: Dataset Statistics

System	Details		Test info			Bug info	
	#LOC	#F	#SP	#Tests	Cov	#V	#IF
ZipMe	3460	13	25	255.0	42.9	150	2.7
GPL	1944	27	99	86.9	99.4	372	13.0
Elevator-FH-JML	854	6	18	166.0	92.9	122	3.6
ExamDB	513	8	8	133.3	99.5	620	1.1
Email-FH-JML	439	9	27	86.0	97.7	126	4.1
BankAccountTP	143	8	34	19.8	99.9	383	4.8

#F and **#SP** stand for the number of features and the average sample size.
Cov and **#V** stand for the statement coverage (%) and the number of buggy versions.
#IF stands for the average number of the involving features.

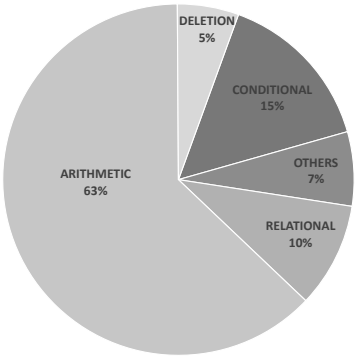

Figure 2: Variability Bugs by Applied Mutation Operators

2.4 Dataset of Variability Bugs

Table 1 provides the general information of our dataset. For each system, a set of products are sampled with 4-wise coverage. In general, for an SPL system, the number of the sampled products depends not only on the number of features but also the feature model of the system. For instance, although ExamDB and BankAccountTP have the same number of features, to achieve 4-wise coverage by sampling technique, BankAccountTP needs to generate 34 products while this figure for ExamDB is only 8 products.

For generating tests, Table 1 shows that the sizes of the generated test suites in different systems are different. For example, more than 8.5K test cases are created for GPL in total. In our dataset, there are 5/6 systems whose generated test suite reaches +90% statement coverage. Moreover, three of them almost reach 100% statement coverage. Especially due to a large code base, ZipMe has 255 tests per product, but its statement coverage only stays at 42.9%.

As seen in Table 2, we generated 1,773 buggy versions of the subject systems. Among them, 339 versions contain a single bug each, while 1,434 versions have two or more bugs. In the dataset, the number of bugs might not be proportional to the size of systems. For instance, ZipMe contains a larger number of statements and has more features than ExamDB. However, there are more bugs generated in ExamDB. The reason is, the number of mutation operators applicable to ExamDB is greater. Thus, there are more mutants and variability bugs that can be generated in ExamDB than in ZipMe. Moreover, the quality of test suite also plays a critical role in generating variability bugs. The test suite with higher statement coverage is more effective in detecting the unexpected behaviors caused by the seeded bugs in the system. Hence, for a system with a better test suite, there might be more variability bugs accepted.

Variability Bugs by Applied Mutation Operators. Figure 2 shows the proportions of the bugs categorized by the groups of mutation operators [16]. As seen, there are more than haft of them (2,288) generated by using *Arithmetic* group. The reason is, comparing to the other groups, *Arithmetic* group contains more mutation operators, such as *AODS*, *AODU*, *AOIS*, *AOIU*, and *AORS* [16], that are applicable for mutating the source code of our selected systems.

Variability Bugs by Code Elements. Figure 3 shows the proportions of the bugs classified by code element types [23]. As seen, groups of *Conditional* and *Assignment* contain more bugs than the others. These proportions are similar to the distribution reported by the prior study on the popular real-fault repository [23].

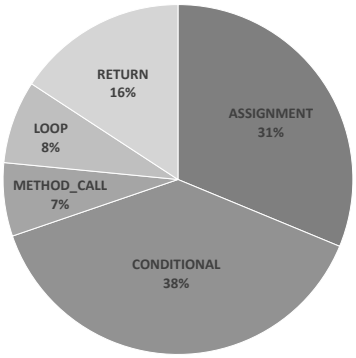

Figure 3: Variability Bugs by Code Elements

Table 2: Buggy versions categorized by number of contained bugs

System	Single-Bug	2-Bug	3-Bug
ZipMe	56	44	50
GPL	105	190	77
Elevator-FH-JML	20	41	61
ExamDB	49	483	88
Email-FH-JML	36	34	56
BankAccountTP	73	238	72

Variability Bugs by Involving Features. Furthermore, for a specific SPL system, the performance of FL techniques might be influenced by the number of features which must be *actually* enabled/disabled to reveal the bugs (*involving features*). In this work, a feature is an involving feature to a variability bug if from a failing product, when switching its current selection (the state of being on or off) makes the resulting product pass all its tests. If the resulting product has not been sampled, we additionally compose the product and generate its tests. For a system containing multiple bugs, since the failures in the sampled products are caused by various bugs, detecting the involving features for each bug in this case might be impossible. For a particular bug, when one of its involving features is switched, the resulting product might still fail because of the other bugs. Thus, in our dataset, we only categorized the buggy versions containing a single bug by the number of involving features. In our dataset, the number of involving features is in the range of [1, 25], about 76% of the cases are less than or equal to 7.

2.5 Description of Dataset Artifacts

Our dataset is published on our website with all required information to evaluate participants' solutions. In particular, all the cases are organized in different folders and each represents a version of the entire SPL system. In each case, artifact structure is as below:

- **Feature Model:** The feature model is in GUIDSL format [9].
- **Sampled Configurations:** A set of configurations with 4-wise coverage which are saved in different files.
- **Source Code:** The system and all the composed products. Note that each product is composed based on feature superimposition mechanism of *FeatureHouse* [6]. For remapping purposes, the link between each code statement in a product and the corresponding statement in the system is recorded.
- **Test Cases:** All the generated test cases of each sampled product.
- **Bug Report:** The locations of modified statements in the system.
- **Test Execution:** This data is supplied in the execution log, informs how many times each statement is executed for each test.

3 SOLUTION EVALUATION

This section describes several standard metrics to evaluate FL techniques for variability bugs in SPL systems. Furthermore, we present the results of the naive adaptation of SBFL technique on the proposed dataset as the baseline results of this challenge.

3.1 Evaluation Metrics

The main focus of FL is to help developers find a good starting point to inspect and initiate the bug-fixing process. Therefore, the effectiveness of FL technique generally based on the percentage of code that needs to be examined until the first faulty location is found. In this challenge, we apply the standard metrics which are widely used in evaluating FL techniques [4, 14, 22, 24, 25].

Rank. The lower *Rank*, the better approach. If there are multiple statements having the same scores, buggy statements are ranked last among them. For the cases of multiple bugs, we measured *Rank* by the position of the first bug in the ranked lists.

EXAM. *EXAM* [24] is the percentage of the statements that must be examined until the first faulty statement is reached:

$$EXAM = \frac{\text{Number of examined statements}}{\text{Total number of statements}} \times 100\%$$

Recall at Top-*N* (Top-*N*). *Top-N* [15] was devised to report the number of cases that at least one bug was found after examining *N* statements in the ranked list.

Proportion of Bugs Localized (PBL). *PBL* [14] measures the proportion of the bugs which are detected after examining a certain number of the statements in the system.

3.2 Baseline Results

To construct baseline results, we conducted several experiments with the naive adaption of SBFL which considers the whole SPL system as a non-configurable code. The suspiciousness score of a statement is measured based on the tests counted from all the products. We use this adaption with different SBFL metrics [4, 25] to evaluate the baseline performance on localizing variability bugs in both single-bug and multiple-bug settings.

3.2.1 Performance in localizing single bug. In this experiment, we use different SBFL metrics to localize the buggy statements in 339 cases where each case contains only one bug. Table 3 shows the average *Rank* and *EXAM* of SBFL using the 5 most popular SBFL metrics. The results of other SBFL metrics can be found on our website [1]. As seen, there are several SBFL metrics that obtained quite similar performance, such as Tarantula and Barinel, Ochiai and Dstar, etc. Op2 is the most effective metric which achieves significantly better performance in both *Rank* and *EXAM* compared to the others' results. Interestingly, the average *Rank* with Op2 in most of the systems is around 5^{th}. Meanwhile, the average *Rank* in

Table 3: SBFL Performance in Single-bug Setting

Metric	ZipMe		GPL		Elevator		ExamDB		Email		BankAccountTP	
	Rank	EXAM	Rank	EXAM	Rank	EXAM	Rank	EXAM	Rank	EXAM	Rank	EXAM
Taratula	64.88	2.80	10.36	1.07	18.40	4.11	5.61	2.24	13.81	5.59	5.53	7.23
Ochiai	59.39	2.56	9.09	0.94	8.55	1.91	3.31	1.32	4.56	1.84	3.95	5.15
Op2	**33.45**	**1.44**	**8.86**	**0.92**	**4.25**	**0.95**	**3.24**	**1.29**	**4.03**	**1.63**	**3.58**	**4.66**
Barinel	64.88	2.80	10.36	1.07	18.40	4.11	5.61	2.24	13.81	5.59	5.55	7.24
Dstar	59.20	2.56	9.09	0.94	8.40	1.88	3.29	1.31	4.61	1.87	3.92	5.11

Elevator, Email are the abbreviations for Elevator-FH-JML and Email-FH-JML systems, respectively.

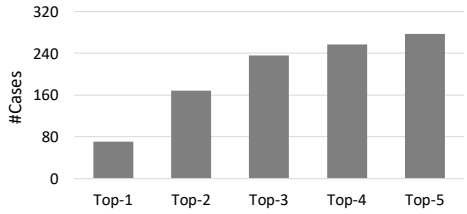

Figure 4: Top-N, $N \in [1, 5]$ of SBFL with Op2 in Single-bug Setting

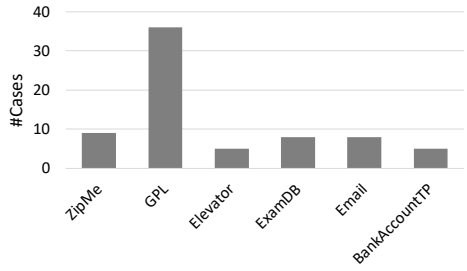

Figure 5: Top-1 of SBFL with Op2 in Single-bug Setting

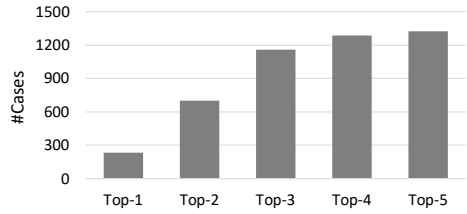

Figure 6: Top-N, $N \in [1, 5]$ of SBFL with Ochiai in Multiple-bug Setting

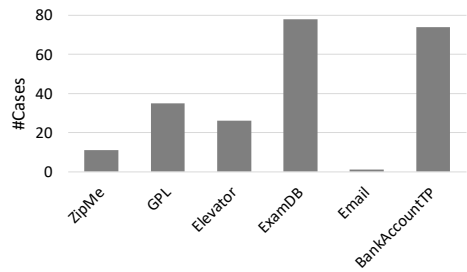

Figure 7: Top-1 of SBFL with Ochiai in Multiple-bug Setting

ZipMe is about 33^{rd}, perhaps because of the low-quality test suites. Particularly, the average test coverage of a variant in this system is only about 43% (Table 1). Although the baseline performance with Op2 in the system ZipMe is worse than in the other systems, it is still better than other SBFL metrics in ZipMe.

The baseline performance with Op2 in *Top-N* is illustrated in Figure 4. Overall, 71 bugs (about 21% of the bugs) are correctly ranked at *Top*-1. In addition, the number of the detected bugs gradually increases when more statements in the ranked lists are examined, about 82% of the bugs are ranked at *Top*-5 accuracy. Interestingly, there are more bugs correctly ranked at the *Top*-1 positions in GPL than in the other systems (Figure 5). This is because, for GPL, there is a large number of the sampled products, which provides more information to locate the bugs. Indeed, among the 71 bugs correctly ranked first, there are 36 cases are buggy versions of GPL.

3.2.2 Performance in localizing multiple bugs. We conducted an experiment on 1,434 cases where each buggy version contains n bugs, $n > 1$. Table 4 shows that Ochiai is the most effective metric for localizing multiple bugs in our dataset. On average, with Ochiai, developers only need to investigate about 4 statements in the ranked lists to find the first buggy statement of a faulty system. Meanwhile, this figure with other SBFL metrics is much worse, e.g., with Op2, which is about 13 statements.

In addition, the performance with Ochiai on *Top-N* in the multiple-bug setting is shown in Figure 6 and Figure 7. Overall, about 16% and 92% of the cases have at least one bug ranked at the *Top*-1 and *Top*-5 positions, respectively. Among the subject systems, Email is the system in which SBFL with Ochiai achieved the lowest performance in *Top*-1 (Figure 7). Especially, by examining the first statement, there is only one case that found the bug.

Furthermore, the average proportion of bugs that are localized in each case by SBFL with Ochiai is shown in Figure 8. On average, only 7% of the bugs can be found after examining the first statement. Moreover, by investigating the first 5 statements in the ranked lists, developers can find 50% of the bugs in a system. Additionally, in order to find about 80% of the bugs, they need to examine up to 24 statements in the ranked lists.

4 SUMMARY

We present a variability fault localization benchmark with a dataset of 1,773 buggy versions of 6 widely-used SPL systems. In our dataset, there are 339 cases of single-bug and 1,434 cases of multiple-bug.

Table 4: SBFL Performance in Multiple-bug Setting

Metric	ZipMe		GPL		Elevator		ExamDB		Email		BankAccountTP	
	Rank	EXAM	Rank	EXAM	Rank	EXAM	Rank	EXAM	Rank	EXAM	Rank	EXAM
Taratula	10.38	0.44	4.08	0.42	7.98	1.78	8.09	3.22	3.53	1.43	4.92	6.36
Ochiai	**8.23**	**0.35**	**2.76**	**0.29**	**4.84**	**1.08**	**2.75**	**1.10**	**3.10**	**1.26**	**2.58**	**3.34**
Op2	30.55	1.31	4.60	0.48	15.93	3.56	11.78	4.69	11.73	4.75	4.24	5.48
Barinel	10.38	0.44	4.11	0.43	7.98	1.78	8.09	3.22	3.53	1.43	4.93	6.36
Dstar	11.03	0.47	2.79	0.29	4.78	1.07	2.71	1.08	2.99	1.21	2.65	3.44

Elevator, Email are the abbreviations for Elevator-FH-JML and Email-FH-JML systems, respectively.

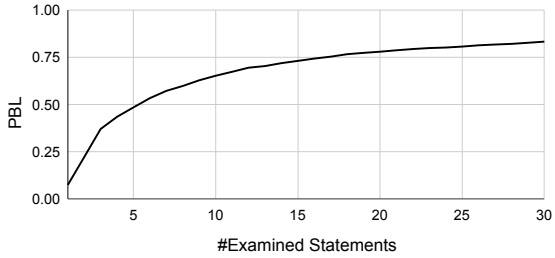

Figure 8: PBL of SBFL with Ochiai in Multiple-bug Setting

The variability bugs in the benchmark are systematically generated by diverse mutation operators, numerous code elements, and different numbers of involving features. We also provide several standard metrics which are broadly applied in evaluating FL techniques. Furthermore, a naive solution adapting SBFL for variability fault localization is evaluated to present the baseline results. We hope that our benchmark can be a common point of comparison for variability fault localization techniques and encourage the researchers to propose better solutions for the challenge case.

ACKNOWLEDGMENTS

In this work, *Kien-Tuan Ngo* was funded by Vingroup Joint Stock Company and supported by the Domestic Master/ PhD Scholarship Programme of Vingroup Innovation Foundation (VINIF), Vingroup Big Data Institute (VINBIGDATA), code VINIF.2020.ThS.04.

REFERENCES

[1] [n.d.]. https://tuanngokien.github.io/splc2021/
[2] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 421–432.
[3] Iago Abal, Jean Melo, Ştefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. 2018. Variability bugs in highly configurable systems: A qualitative analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 26, 3 (2018), 1–34.
[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION*. IEEE, 89–98.
[5] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated.
[6] Sven Apel and Christian Kästner. 2009. An overview of feature-oriented software development. *Journal of Object Technology* 8, 5 (2009), 49–84.
[7] Sven Apel, Christian Kästner, and Christian Lengauer. 2011. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering* 39, 1 (2011), 63–79.
[8] Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: case studies and experiments. In *2013 35th International Conference on Software Engineering*. IEEE, 482–491.
[9] Don Batory. 2005. Feature models, grammars, and propositional formulas. In *International Conference on Software Product Lines*. Springer, 7–20.
[10] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.
[11] Brady J Garvin and Myra B Cohen. 2011. Feature interaction faults revisited: An exploratory study. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE, 90–99.
[12] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*. 46–55.
[13] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.
[14] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 114–125.
[15] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.
[16] Yu-Seung Ma and Jeff Offutt. 2005. Description of method-level mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep* (2005).
[17] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. 2006. MuJava: a mutation system for Java. In *Proceedings of the 28th international conference on Software engineering*. 827–830.
[18] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering*. IEEE, 643–654.
[19] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On essential configuration complexity: measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 483–494.
[20] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 50–61.
[21] Son Nguyen, Hoan Nguyen, Ngoc Tran, Hieu Tran, and Tien Nguyen. 2019. Feature-interaction aware configuration prioritization for configurable code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 489–501.
[22] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.
[23] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 130–140.
[24] Eric Wong, Tingting Wei, Yu Qi, and Lei Zhao. 2008. A crosstab-based statistical method for effective fault localization. In *2008 1st international conference on software testing, verification, and validation*. IEEE, 42–51.
[25] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.