Configuration-dependent Fault Localization

Son Nguyen
The University of Texas at Dallas
800 W. Campbell, Richardson, TX 75080, USA
sonnguyen@utdallas.edu

***Abstract*—In a buggy configurable system, configuration-dependent bugs cause the failures in only certain configurations due to unexpected interactions among features. Manually localizing configuration-dependent faults in configurable systems could be highly time-consuming due to their complexity. However, the cause of configuration-dependent bugs is not considered by existing automated fault localization techniques, which are designed to localize bugs in non-configurable code. Thus, their capacity for efficient configuration-dependent localization is limited. In this work, we propose COFL, a novel approach to localize configuration-dependent bugs by identifying and analyzing suspicious feature interactions that potentially cause the failures in buggy configurable systems. We evaluated the efficiency of COFL in fault localization of artificial configuration-dependent faults in a highly-configurable system. We found that COFL significantly improves the baseline spectrum-based approaches. With COFL, on average, the correctness in ranking the buggy statements increases more than 7 times, and the search space is significantly narrowed down, about 15 times.**

I. Problem Statement and Background

Configurable system supports the diversification of software products by providing *configuration options* that are used to control different *features*. However, this induces challenges in program analyses and quality assurance [13], [4], [14].

In quality assurance for configurable system, *configuration-dependent faults*, which cause the failures in only certain *configurations* because of unexpected *interactions* among several features, are not rare [7], [8], [11], [17]. Manually localizing configuration-dependent faults in configurable systems could be highly costly due to their complexity [12], [14].

Meanwhile, existing automated fault localization techniques [16] are designed to localize the faults in non-configurable code. Specifically, for configurable code, they do not consider the cause of configuration-dependent bug(s), which is the unexpected feature interactions. Thus, many parts of the buggy system, which are not related to those unexpected interactions, are inappropriately considered as suspicious. Indeed, for example, despite that one can adapt spectrum-based techniques [10], [2], [16] for configurable code by considering static conditional statements (e.g., `#if`) on configuration options as `if`-statements, the adapted techniques still access and rank all executed statements including the ones that might not affect the fault-inducing interactions, even not the program's states. For slice-based methods [15], [3], the suspicious domain is reduced to all slices that are related to failed test execution information, which might include the slices irrelevant to the unexpected feature interactions. Therefore, the capacity of the traditional techniques [16] for efficient configuration-dependent fault localization is limited.

II. Motivation and Observation

Let us start with a real configuration-dependent bug in Linux kernel to motivate our approach (Fig. 1). In this example, the maximum value of `KMALLOC_SHIFT_HIGH` is `25` (lines 9–10). This indicates that `kmalloc_caches` contains a maximum of 26 elements (line 13). When `PPC_256K_PAGES` is enabled and `PPC_16K_PAGES` is disabled, the maximum index used to access `kmalloc_caches` is defined as `(PAGE_SHIFT + MAX_ORDER-1)` (line 18), which is `28`. This leads to an exception that array `kmalloc_caches` is accessed out of its bounds. However, this bug is not revealed by any configuration, except the configurations in which `PPC_256K_PAGES`, `SLAB`, `LOCKDEP`, and `SLOB` are enabled, and `PPC_16K_PAGES` is disabled.

Observations. From the example shown in Fig. 1, we have the following observations:

O1. *In a configurable system containing configuration-dependent bug, there are certain features that are (ir)relevant to the visibility of the bug*. For example, in Fig. 1, feature `NUMA` (line 27) does not involve in the bug because when `PPC_256K_PAGES`, `SLAB`, `LOCKDEP`, and `SLOB` are enabled and `PPC_16K_PAGES` is disabled, the system still fails regardless of whether `NUMA` is enabled or disabled. Meanwhile, for some configurations, enabling/disabling certain features might make the test results (passing all tests or not) of the resulting configurations change. In Fig. 1, the all-enabled configuration behaves as expected, while if `PPC_16K_PAGES` is disabled and all other options enabled, the resulting configuration fails.

O2. *In the features f_Es that* ***must be enabled*** *to make the bug visible, only the statements that implement the interaction between them are more likely to be buggy than others*. In `LOCKDEP`, the buggy statement is at line 18, which is one of the statements realizing the interaction between f_Es. In contrast, if the bug is caused by the statements not related to the interaction between f_Es, the visibility of the bug would not depend on all of those f_Es. In Fig. 1, the enabled features f_Es include `PPC_256K_PAGES`,`SLAB`, `LOCKDEP`, `SLOB`, and `PPC_16K_PAGES`. The bug is not related to the statement at line 21 in `LOCKDEP`, which is not used to realize the interaction of f_Es.

O3. *In the features f_Ds that* ***must be disabled*** *to make the bug visible, the statements that implement the interactions with f_Es also provide useful indication to help us find suspicious statements in f_Es*. In Fig. 1, `PPC_16K_PAGES` is a disabled feature f_D. Although line 6 in `PPC_16K_PAGES` (being disabled)

```
1  #define MAX_ORDER 11
2  #if defined(CONFIG_PPC_256K_PAGES)
3  #define PAGE_SHIFT    18
4  #endif
5  #if defined(CONFIG_PPC_16K_PAGES)
6  #define PAGE_SHIFT    14
7  #endif
8  #ifdef CONFIG_SLAB
9  #define KMALLOC_SHIFT_HIGH ((MAX_ORDER+PAGE_SHIFT-1)\
10             <= 25 ? (MAX_ORDER + PAGE_SHIFT - 1) : 25)
11 #endif
12 #ifdef CONFIG_SLOB
13 int* kmalloc_caches[KMALLOC_SHIFT_HIGH + 1];
14 #endif
15 #ifdef CONFIG_LOCKDEP
16 static void init_node_lock_keys(int node) {
17     int i, lock;
18     for (i = 1; i < PAGE_SHIFT + MAX_ORDER; i++) {
19     //Patch: for (i = 1; i < KMALLOC_SHIFT_HIGH; i++){
20         int* cache = kmalloc_caches[i];
21         lock = slab_set_lock_classes(node);
22     }
23 }
24 #endif
25 static void cpuup_prepare(int node){
26 #ifdef CONFIG_NUMA
27     node = 0;
28 #endif
29     init_node_lock_keys(node);
30 }
```

Figure 1. A Configuration-dependent Bug in Linux Kernel

is not considered as faulty, however analyzing the impact of the statement at this line (defining `PAGE_SHIFT`) on the statements in `LOCKDEP` and `SLAB` can provide the suggestion to identify the statement need to be fixed (`i < PAGE_SHIFT + MAX_ORDER`). The intuition of this phenomenon is that despite that the statements in f_Ds are not faulty, f_Ds have the impact of "hiding"/"masking" the bug when they are enabled. Thus, we need to consider the interactions of other features with f_Ds in localizing configuration-dependent bugs.

O4. Because certain statements in the enabled features to make the bug visible are considered as suspicious, the statements in the same/different features having impacts on the suspicious statements via program dependencies [5], [6] should also be considered as suspicious. For example, although line 1 does not belong to any f_E, that statement is also suspicious since it has an impact on the statements at lines 9, 10, and 18.

III. Approach

We propose, CoFL, a novel approach for configuration-dependent fault localization. For a buggy configurable code, to reduce the suspicious domain, **we analyze the test results of the executed configurations, the code, and the test execution information to identify the executed statements related to the interactions among the features whose enabling/disabling affect the visibility of the bugs which potentially cause the failures**. These statements are ranked by their suspiciousness levels assigned by existing techniques [16] based on their test execution information.

In particular, CoFL first determines minimal sets of feature candidates whose enabling/disabling (feature selection) make the bugs visible (based on **O1**). Let us call such a set of feature selections the *suspicious partial configuration (SPC)*. For example, {`SLAB=T`, `PPC_16K_PAGES=F`, `PPC_256K_PAGES=T`, `LOCKDEP=T`, `SLOB=T`} is considered as the SPC of the bug in Fig.1. The selection of `NUMA` does not belong to the SPC of the bug because they do not have any impact on its visibility.

Next, CoFL aims to detect the suspicious statements that are responsible for the feature interactions and potentially cause the faults. To do that, it analyzes the features in SPC to detect the interactions between them that are potentially cause/disguise the configuration-dependent bugs. Then, CoFL detects the statements that realize those interactions (based on **O2** and **O3**). The interactions are detected via the shared program entities including *variables* and *functions* controlled by different features and the operations including *define* and *use* performed on them. For example, `PPC_256K_PAGES` *define* `PAGE_SHIFT` which is *used* by `SLAB` and `LOCKDEP`. In the example, the statements realizing the interactions among the f_Es in the SPC are at lines 3, 9, 10, 13, 18, and 20 (S_1). Meanwhile, the statements in f_Es for interactions between the f_Es and the f_Ds in the SPC are at lines 9 and 18 (S_2).

After that, the suspicious statements are used to detect other suspicious statements that are executed and have dependencies on the statements in both S_1 and S_2 in the failed configurations (based on **O3** and **O4**). The output for the running example is the set of statements at lines 3, 9, 10, 18, and 1. Finally, these statements are ranked by their suspiciousness scores computed by existing techniques [16] such as spectrum-based methods based on their test execution information.

IV. Empirical Evaluation

We evaluate CoFL's efficiency in localizing configuration-dependent bugs over 2 spectrum-based techniques, Tarantula [10] and Ochiai [2]. We randomly seeded the set of 32 artifical configuration-dependent bugs into the subject system BusyBox [1]. For each bug, the output rank are evaluated via $EXAM$ [9] and the suspicious domain size (SDS). The lower $EXAM$ and smaller SDS the more efficient the technique.

Table I
Comparison Results

	$EXAM$	SDS
Tarantula	37.50	147.17
CoFL with Tarantula	5.12	10.58
Ochiai	36.54	147.17
CoFL with Ochiai	4.97	10.58

Table I shows the average $EXAM$ and average SDS of Tarantula, Ochiai and CoFL with their formula. As seen, on average, the correctness in ranking the buggy statements increases more than 7 times, and the search space is significantly narrowed down, about 15 times.

Conclusion. The novel idea of CoFL, our configuration-dependent fault localization method for configurable code, is to leverage the test results and code analysis to detect interactions between features that potentially cause the bugs and use these interactions to reduce the suspicious domain.

REFERENCES

[1] Busybox: The swiss army knife of embedded linux, 2018.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[3] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[4] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 1st edition, 2016.

[5] Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[7] Brady J. Garvin and Myra B. Cohen. Feature interaction faults revisited: An exploratory study. In *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ISSRE '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.

[8] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 244–254, Piscataway, NJ, USA, 2012. IEEE Press.

[9] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[10] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 467–477. IEEE, 2002.

[11] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, June 2004.

[12] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. On essential configuration complexity: Measuring interactions in highly-configurable systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 483–494, New York, NY, USA, 2016. ACM.

[13] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.

[14] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, June 2014.

[15] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[17] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 159–172, New York, NY, USA, 2011. ACM.