Front. Comput. Sci., 2023, 17(3): 173207
https://doi.org/10.1007/s11704-022-1729-x

LETTER

# VulLoc: vulnerability localization based on inducing commits and fixing commits

**Lili BO[1,2], Yue LI[1], Xiaobing SUN (✉)[1], Xiaoxue WU[1], Bin LI[1]**

1 School of Information Engineering, Yangzhou University, Yangzhou 225127, China
2 Key Laboratory of Safety-Critical Software, Ministry of Industry and Information Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China



## 1 Introduction

Software evolves with the code change, including fixing the vulnerabilities, refactoring the software and adding new functions. Particularly, developers from open-source projects often look for the information of inducing commits when devising the fixing patches of a vulnerability [1]. In the process of fixing the current vulnerabilities, it is inevitable to introduce new vulnerabilities. As is shown in Fig. 1, after *vulnerability1* occurs, developers committed the code changes to fix *vulnerability*1, but cause *vulnerability*2. In this paper, commits that introduce new vulnerabilities are called *vulnerability-inducing commits*, and commits that finally fix the vulnerability are known as *vulnerability-fixing commits*. In the following, they are abbreviated as vul-inducing commits and vul-fixing commits, respectively.

Our work is motivated by the empirical studies of the correlations between vul-inducing commits and vul-fixing commits. To create the dataset, we first collected more than 1,700 vulnerabilities from Common Vulnerabilities & Exposures (CVE) by examining the vulnerability descriptions with the keywords "*caused by*", "*regression*", "*introduce*" and "*because of* ". Then, we filtered out the vulnerabilities which contain noises and got 453 vulnerabilities. Finally, through manual screening, 71 vulnerabilities were obtained with the vul-inducing commits and vul-fixing commits.

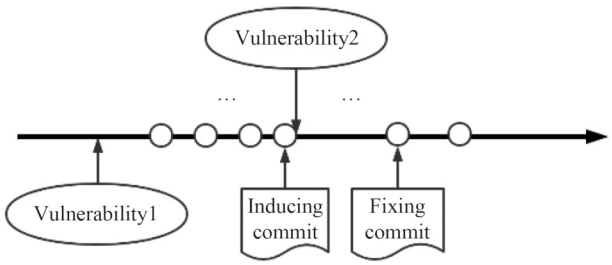


**Fig. 1** The example of vul-inducing commits and vul-fixing commits

Received December 15, 2021; accepted July 5, 2022

E-mail: xbsun@yzu.edu.cn

We explore the reasons for introducing a new vulnerability by analyzing both vulnerability descriptions and code changes in vul-inducing commits as well as vul-fixing commits. Overall, there are three reasons:

- Incorrect fix (24%): The statements modified in vul-inducing commits are modified again in vul-fixing commits.
- Incomplete fix (48%): The statements that should be modified but not be modified in vul-inducing commits are modified in vul-fixing commits.
- Incomplete & Incorrect fix (28%): Both incorrect fix or incomplete fix exist in vul-fixing commits.

Furthermore, we summarize the root causes of vulnerability introduction, which is shown in Table 1. It can be seen that, logic error is the main root cause (66.7%), followed by assignment error (22.2%).

Similar to the work by Wen et al. [1], we measure the file coverage and statement coverage with the following formulas.

Measuring file coverage: It is designed to measure how many source files that are modified by the vul-fixing commits are also modified by the associate vul-inducing commits. The formula is as follows.

$$File\ \ Coverage(v) = \frac{|S_{I_v} \cap S_{F_v}|}{|S_{F_v}|},$$

where $S_{F_v}$ denotes all the source files modified by $v$'s vul-fixing commits $F_v$, and $S_{I_v}$ denotes those by $v$'s vul-inducing

**Table 1** Root cause for the introduction of vulnerabilities

| Root cause | Proportion/% | Description |
|---|---|---|
| Logic error | 66.7 | Caused by repeated logic, incorrect conditions, lack of branching, incorrect loops, or incorrect logical order, etc. |
| Assignment error | 22.2 | Caused by incorrect initialization, or incorrect parameter types, etc. |
| Computation error | 8.3 | Caused by inaccurate values, wrong operands, etc. |
| Interface error | 1.4 | Caused by calling the wrong method interface, etc. |
| Others | 4.6 | Not in the above list |

commits.

Measuring statement coverage: It is designed to measure how many statements that are modified by the vul-fixing commits are also modified by the associate vul-inducing commits. The formula is as follows.

$$Statement\ \ Coverage(v) = \frac{|L_{I_v} \cap L_{F_v}|}{|L_{F_v}|},$$

where $L_{F_v}$ denotes all the source statements modified by the $v$'s vul-fixing commits $F_v$, and $L_{I_v}$ denotes those by $v$'s vul-inducing commits.

We calculated the file coverage rate and the statement coverage rate based on our dataset, and found that they achieve 69.8% and 29.8%, respectively, which indicates that most of the code changes in vul-inducing commits and vul-fixing commits are in the same files but are not the same statements.

## 2 Methodologies

### 2.1 Feature definition

Given a vul-inducing commits $c$, with $S_{file}$ as the set of files from the vul-inducing commits, $S$ as the set of statements from $S_{file}$.

**Feature1** Given that $S_m$ is the modified statements set, if $s \in S$, then extract $s$. For example, Fig. 2 shows a segment of vul-inducing commit of CVE-2019-5747. The statements modified in the vul-inducing commit are the added statements (which are shown after "+") or deleted statements (which are shown after "-"), and when it is an update operation, only the statements after "+" are extracted, that is, "*if* (*code* == *DHCP_SUBNET && temp*[*-OPT_DATA + OPT_LEN*] == 4)".

**Feature2** Given that $S_{key}$ is the set of method names and variable names split from the statements $s$ in $S_m$, if $s \in S \& S \supset S_{key}$, then extract $s$. For example, first, we extract the method names and variable names from the statements in feature1, e.g., "code", "DHCP_SUBNET", "Temp" in the statement. Then, we extract the statements that contain these method names and variable names in the file "networking/udhcp/dhcpc.c" as well as other files involved in the inducing commit.

**Feature3** Given that $S_{if}$ is the if statements set from the $S_{file}$, $s$ is the statement in $S_{if}$, if $s \in S$, then extract $s$. For example, we extract all of the *if* statements in the file "networking/udhcp/dhcpc.c" shown in Fig. 2 and other files in the vul-inducing commit.

### 2.2 Suspicious statements space construction

The statements extracted by feature1, feature2, and feature3 constitute the suspicious statements space. We conduct a specific control experiment to determine the weight of features, i.e., the score of different features. We changed the value of each feature in the step of 0.1 from 0 to 1. We found that, when the scores of feature1, feature2, and feature3 arrive at 0.4, 0.2, and 0.3, the approach obtained the best vulnerability localization performance in terms of MAP, MRR, and TOP-K.


**Fig. 2** Part of inducing commits from CVE-2019-5747

### 2.3 Suspicious statements ranking

VulLoc trains a machine learning model to rank the statements with three features and suspicious statements space. Learning-to-rank is a family of supervised machine learning techniques that solves ranking problems in information retrieval. It is widely used in information retrieval and natural language processing. To be specific, we choose rankSVM as it has demonstrated its effectiveness in the existing bug localization work [2]. Here, we replace various technical features with feature1, feature2, and feature3, and train a learning model through ten-fold cross-validation. After the learned model assigns a score to each suspicious statement, a ranked list is obtained.

## 3 Experiment and evaluation

We built our dataset containing 71 pairs of vul-inducing commits and vul-fixing commits as well as their description. In our experiments, the first 75% (53 vulnerability data) samples in the dataset are used as training samples, and the remaining 25% are used as test samples. We use ten-fold cross-validation to train the model. In the process of model tuning, the result of the model is affected by adjusting the penalty term C. Thus, we increase the penalty term C between (0, 1) with a step size of 0.1. The result shows that when C is 0.1, the model achieves the best effect.

In addition, we choose BugLocator [3] for comparison due to the lack of dynamic information, e.g., test cases, stack trace information, etc. In our experiments, BugLocator takes as input all the pairs of a vulnerability description with each method code. The input of both BugLocator and our approach contains method code of the same project. In addition, we use the metrics $E_{inspect}$@n, MAP and MRR to evaluate our approach.

Table 2 lists the comparison result with BugLocator on $E_{inspect}$@n, MAP and MRR. The value of VulLoc $E_{inspect}$@n is between 35% and 53%, and the value of BugLocator is between 8.3% and 33.3%. It can be seen that the effectiveness is significantly improved. For VulLoc, 35% of vulnerabilities are ranked first for suspicious statements. $E_{inspect}$@3 accounts for 45%, $E_{inspect}$@5 accounts for 46%. 53% of the vulnerabilities were ultimately located in the top ten

**Table 2** Comparison results of effectiveness

| Tec/Metrics | VulLoc/% | BugLocator/% |
|---|---|---|
| $E_{inspect}$@1 | 36.6 | 8.3 |
| $E_{inspect}$@3 | 45 | 15.3 |
| $E_{inspect}$@5 | 46 | 22.2 |
| $E_{\text{inspect}}$@10 | 53 | 33.3 |
| MAP | 41.7 | 14.13 |
| MRR | 46.1 | 14.0 |

**Table 3** The results on $E_{inspect}$@n, MAP, MRR

| Tec/Metrics | Feature1/% | Feature2/% | Feature3/% | (Feature1+Feature2)/% | (Feature1+Feature3)/% | (Feature2+Feature3)/% |
|---|---|---|---|---|---|---|
| $E_{inspect}$@1 | 5.0 | 0 | 0 | 35.0 | 22 | 0 |
| $E_{inspect}$@3 | 25.9 | 0 | 4.9 | 45 | 40 | 3.7 |
| $E_{inspect}$@5 | 33.3 | 1.2 | 4.9 | 46 | 43.2 | 3.7 |
| $E_{inspect}$@10 | 49.4 | 4.9 | 4.9 | 53 | 49.1 | 6 |
| MAP | 25.3 | 18.5 | 11.6 | 41 | 31.2 | 13.9 |
| MRR | 29.4 | 23.1 | 17.5 | 46.1 | 38.8 | 24.9 |

recommendation lists. In addition, it can be seen that the model achieves the highest performance on $E_{inspect}$@3. Developers usually only spend their energy checking the top-ranked suspicious statements. VulLoc achieves 41.7% and 46.1% on MAP and MRR, respectively. Compared with BugLocator, it improved 27.4% and 32.1% on MAP and MRR, respectively. The above analysis shows that VulLoc has a good localization performance and VulLoc is better than BugLocator.

Then, we performed ablation experiments to compare the contribution of feature1, feature2, and feature3. The results are shown in Table 3. From the perspective of $E_{inspect}$@n, we can see that the effect of fearture1 is greater than that of feature2 and feature3. For feature1, the change from $E_{inspect}$@1 to $E_{inspect}$@10 increases sharply, which increases rapidly to 25.9%. It is 49.4% on $E_{inspect}$@10, and the number of vulnerabilities finally found was close to half. For feature2, the value of $E_{inspect}$@1 and $E_{inspect}$@3 is 0, indicating that there are no top three statements in the ranked list. The top five suspicious statements account for 1.2%, and the $E_{inspect}$@10 statements account for 4.9%. For feature3, $E_{inspect}$@1 is 0, and there is no change on $E_{inspect}$@3, 5, 10, which indicates the limitation of a single feature. In terms of MAP and MRR, for feature1, the value of MAP is 25.3% and the value of MRR is 29.4%. This shows that the contribution rate of feature1 is greater than that of feature2 and feature3.

By comparing feature1, feature2, feature3 on $E_{inspect}$@1, 3, 5, 10, we can see that the modified statements extracted from vul-inducing commits as a technical feature is more effective than the other two features. However, the difference in the ratio of MAP to MRR is within 10%, indicating that the statements obtained by matching variable names and method names will have a higher rank. The combination of the features makes the value of $E_{inspect}$@1 increase rapidly. The combination of feature1 and feature2 or feature1 and feature3 can greatly improve the effectiveness of a single feature. MAP and MRR are improved with the combination of features, which shows that the combination of features is effective. Also, feature3 can help to improve feature2.

Finally, we compare the efficiency of VulLoc and BugLocator by calculating the runtime for each vulnerability to be recommended in the ranked list. Table 4 shows the results. We randomly used the test set to run 10 times. The average time of VulLoc is 1.9326 seconds, which is less than that of BugLocator. The increase of suspicious statements may cost more time. But in fact, VulLoc is more efficient than BugLocator. The reason is that, BugLocator needs to search the files in the project, while the VulLoc approach only needs to analyze and learn the statements in the suspicious statements space, which saves time.

**Table 4** Comparison results of efficiency

| Runtime | VulLoc | BugLocator/s |
|---|---|---|
| Time1 | 0.165s (↑1112%) | 2 |
| Time2 | 0.158s (↑659%) | 1.2 |
| Time3 | 0.156s (↑1246%) | 2.1 |
| Time4 | 0.149s (↑705%) | 1.2 |
| Time5 | 0.148s (↑2197%) | 3.4 |
| Time6 | 0.156s (↑1438%) | 2.4 |
| Time7 | 0.165s (↑809%) | 1.5 |
| Time8 | 0.228s (↑1075%) | 3.1 |
| Time9 | 0.249s (↑984%) | 2.7 |
| Time10 | 0.2s (↑650%) | 1.5 |
| Average | 0.1774s (↑1089%) | 2.11 |

## 4 Conclusion

In this paper, we explore the correlations between vul-inducing commits and vul-fixing commits, and propose an automated vulnerability localization approach called VulLoc to recommend a ranked list of suspicious methods. Compared with BugLocator, VulLoc can achieve an improvement on $E_{inspect}$@n, MAP, MRR, respectively and have significant efficiency on vulnerability localization.

**Acknowledgements** This work was supported by the National Natural Science Foundation of China (Grant Nos. 61872312, 61972335, and 62002309); the Natural Science Foundation of the Jiangsu Higher Education Institutions of China (20KJB520016); the Innovation (Science and Technology) Project of Scientific Research Base of Nanjing University of Aeronautics and Astronautics (NJ2020022).

## References

1. Wen M, Wu R, Liu Y, Tian Y, Xie X, Cheung S C, Su Z. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2019, 326−337
2. Zou D, Liang J, Xiong Y, Ernst M D, Zhang L. An empirical study of fault localization families and their combinations. IEEE Transactions on Software Engineering, 2021, 47(2): 332–347
3. Zhou J, Zhang H, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th International Conference on Software Engineering. 2012, 14−24