Prompt Fix: Vulnerability Automatic Repair Technology Based on Prompt Engineering

1st Peng Liu
Guangzhou Institute of Technology, Xidian University
liup2@nipc.org.cn

2nd He Wang
Guangzhou Institute of Technology, Xidian University
hewang@xidian.edu.cn

3nd Chen Zheng
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Nanjing
zhengchen@iscas.ac.cn

4th Yuqing Zhang
University of Chinese Academy of Sciences
Xidian University
Hainan University
zhangyq@nipc.org.cn

2024 International Conference on Computing, Networking and Communications (ICNC) | 979-8-3503-7099-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICNC59896.2024.10556123

***Abstract*—With the emergence of large-scale language models (LLM), the powerful capabilities of LLM in natural language processing have attracted attention. Based on programming language LLM (Programming Language Model, PLM), we use prompt templates to explore its potential in the field of automatic vulnerability repair, and combine it with a special workflow to improve its efficiency in automatic vulnerability repair tasks. Specifically, we design four prompt templates for handling vulnerable code, and design an iterative reasoning method to improve the efficiency of vulnerability fixing. We selected multiple typical LLMs for evaluation on multiple data sets. The results show that reasonable prompt templates can effectively improve the efficiency of automatic vulnerability repair, which is significantly improved compared with neural machine translation technology. In addition, we also discussed previous bug fixing related work and our work, and pointed out some of our shortcomings and directions for future improvements.**

***Index Terms*—automatic program repair; vulnerability repair; large language model; program language model**

I. Introduction

With the popularization and development of software technology, software systems have become an indispensable part of modern society. However, this also brings about the problem of increasing number and complexity of software defects. Security personnel spend a lot of time and effort fixing these flaws, constantly manually patching and testing. Software defects have long caused significant and lasting damage to all aspects of human activity.For example, in 2017, attackers exploited the EternalBlue series of vulnerabilities to create a widespread WannaCry ransomware incident [1]. Between 2018 and 2020, a software design flaw at Boeing caused two Boeing 737 Max aircraft to crash and jeopardized a critical test flight of the Starliner spacecraft [2].

Nowadays, the creation and use of automation tools has become the common pursuit and development trend of academia and industry. In this trend, in the face of increasingly complex repair tasks, automatic program repair technology (Automatic Program Repair, APR) [3] was born. Due to the promising prospect of automatic program repair (APR), researchers have proposed various APR techniques, the researchers leveraged recent advances in deep learning to further improve APR. Such learning-based techniques typically treat APR as a neural machine translation problem, using pairs of bug/fixed code snippets as source/target languages for translation. In this way, such techniques rely heavily on large numbers of high-quality bug-fix commits, which can be expensive and challenging, while potentially limiting their editorial diversity and contextual representation.

In this paper, we focus on whether LLM for code completion can help us fix security vulnerabilities. Since LLM can be guided by adding hints to hints, we use black-box, "off-the-shelf" LLMs to generate the feasibility of replacement code for identified security vulnerabilities, perhaps as part of an overall program repair framework. This is in stark contrast to previous work on training specialized neural machine translation (NMT)-based models to fix software bugs [4].

Furthermore, we also consider that previous methods simply resample the LLM given the same constructed input or cues created from the original error code, which not only leads to repeated generation of the same incorrect patch, but also misses key information. To this end, we propose an iterative repair method to add to the workflow of automatic vulnerability repair, regenerate the prompt content from the wrong code output by the model, and use it as input again for the model to reason.

Our contributions are as follows:

- An automated experimental framework is built, which can automatically use LLM for reasoning, automatically generate code prompt templates as model input, automatically process model output (such as compiling and deduplication), perform iterative reasoning, detect and generate repair codes accuracy and analysis of experimental results.
- We selected 7 different LLMs and designed 4 prompt templates with different complexity as the input of the

He Wang is the corresponding author (e-mail: hewang@xidian.edu.cn)

model, and conducted experiments on the dataset created by ourselves and two other benchmark datasets.
- A new workflow for automatic vulnerability repair is proposed. Iterative reasoning is added to the repair process, and the output of model errors is regenerated as a prompt template and used as input for re-reasoning to improve the effect of automatic vulnerability repair.
- We created a vulnerability dataset PromptFix.

II. Background

A. Automatic Program Repair

Automatic program repair is a technology in the field of computer science, which aims to identify, locate and repair errors, defects or vulnerabilities in software programs through automated methods. The goal of this technique is to reduce human intervention, speed up the software development process, and improve software quality and reliability.

Subsequently, with the emergence of large-scale generative language models, the field of automatic vulnerability repair has gradually shifted to the use of large-scale pre-trained program language models trained in program languages for automatic vulnerability repair tasks. This approach has great advantages, such as no need for paired training data sets, no need to start training from the initialization state of the model, and so on. This article mainly uses a large-scale pre-trained program language model for automatic vulnerability repair tasks.

B. Automated Repair Tasks For Software Vulnerabilities

A software vulnerability is an error, defect, or design deficiency in a computer program or system that could be exploited by a malicious attacker to gain unauthorized access, information, or control.

Research on security vulnerability fixes seems to be in its infancy at the moment. Vulnerability remediation targets specific security vulnerabilities in APR, which usually require carefully crafted inputs to trigger specific vulnerabilities due to their high complexity and stealth. Therefore, automated remediation of security vulnerabilities is more challenging. While bug and vulnerability fixing in general are remedial tasks in nature, there are differences in the problem areas they focus on. This issue has been discussed by scholars [5], and it may not be appropriate to directly migrate APR tools suitable for bug repairing to vulnerability repairing tasks.

C. Large Language Models

A large language model (LLM) is a complex computer model capable of processing and generating natural language. They learn the structure, syntax, semantics, and context of language by training on large amounts of text data, so that they can generate reasonable text responses, articles, summaries, etc. The large-scale pre-trained program language model is a model that is trained with a large number of program languages on this basis.

D. Prompt Template

Prompt learning usually refers to in artificial intelligence systems, by providing specific input text to the system to guide it to generate corresponding output text. In the field of APR, the content of the prompt can be information related to the vulnerability, the code segment of the vulnerability, the natural language that prompts how to fix the vulnerability, etc. If you can use this information, you can further tap the ability of LLM in the field of automatic vulnerability repair. Prompt learning [6] has been proven to be a technique that can effectively improve the accuracy of LLM text generation tasks, but further research is needed to improve its efficiency. Therefore, a research focus of this paper is how to design a reasonable prompt as input to let the model Perform automatic vulnerability repair tasks.

III. Method Design

A. Workflow

The vulnerability automatic repair workflow designed in this article is shown in Figure 1. We use vulnerability type as the classification standard, and all related files for each vulnerability are stored in a folder with the same name. There are some key files in this folder, including the original code file of the vulnerability, the configuration file of the vulnerability scenario, the functional test file of the vulnerability, the input file of the original code after template processing, and the folder where the generated results are saved. After the scene is generated, it will enter the second stage for vulnerability repair, which is model inference. After the reasoning is completed, enter the third step to verify the repair results.

B. Experimental Environment Construction

Our experiment is carried out on the Gemini cluster, which integrates 7 RTX 3090 GPUs, has 500GB memory and 150 Core CPU, and the CUDA version is 11.7. The Ubuntu 22.04 operating system was started on the cluster. The version of Python used is 3.8.10, and the version of CodeQL is 2.7.2.Codeql is an engine that converts code into a database-like form and performs analysis based on the database. In CodeQL, code is treated as data. Security vulnerabilities, bugs, and other errors are modeled as queries that can be executed against a database extracted from the code.

C. Model Selection And Hyperparameter Determination

CodeGen series [7]: This is a generative model based on the GPT series, divided into three versions: NL, Multi, and Mono, with parameters ranging from 350M–16B. The model excels at the downstream task of code generation, which is left-to-right reasoning.

PolyCoder model [8]: This model is based on the GPT-2 architecture, has 2.7B parameters, and is trained for a total of 249GB of code in 12 programming languages. In the C programming language, PolyCoder outperforms commercial models including Codex.

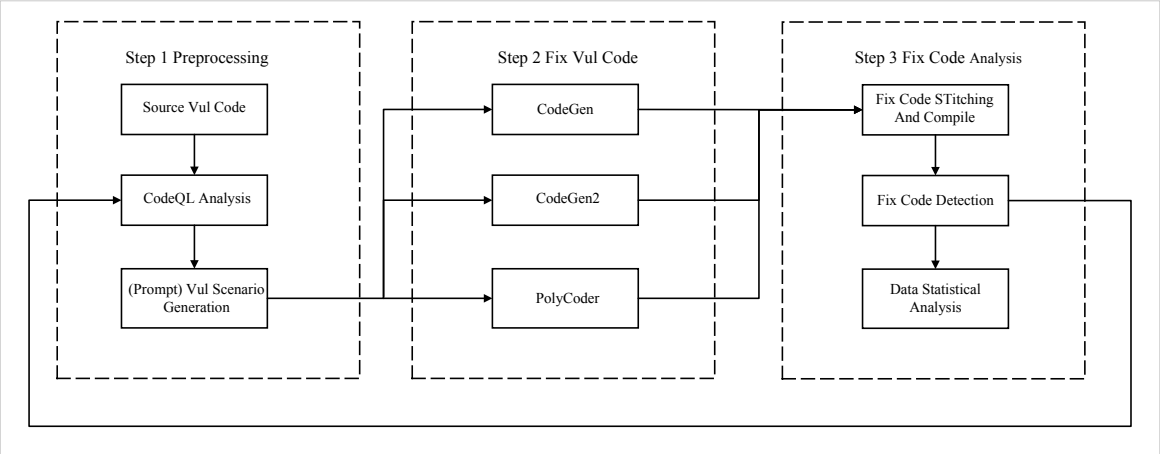

Fig. 1. Automated vulnerability repair workflow

CodeGen2 series [9]: CodeGen2 is an upgraded version of the first generation of CodeGen. This generation of models is different from the original CodeGen model series. CodeGen2 has filling capabilities and supports more programming languages.

The determination of model hyperparameters is a very important process, to this end, we selected two vulnerabilities, CWE787 and CWE-89, and designed experiments to determine hyperparameters.For this experiment, we used the PolyCoder model, and combined temperature and top-p, and divided them according to the step size of 0.1, so the number of all their combinations is 100, and the model must generate 100 output results under each parameter , which means that 20,000 fix codes will eventually be generated for two vulnerabilities. Taking CWE 787 as an example, we found that the combination of higher temperature and higher top-p works better. But for CWE 89, the opposite is true, the combination of lower temperature and lower top-p works better. This shows that there is no specific combination of temperature and top-p that is most suitable. So we refer to PolyCoder's suggestion, fix top-p to 1, and divide the temperature with a step size of 0.1, which can significantly reduce the number of combinations that need to be processed in the experiment.

D. Dataset Selection

Three data sets were used in this experiment, namely PromptFix, ExtractFix and VulFix data sets constructed by hand.

The PromptFix dataset is manually written code files containing the top 10 CWE vulnerability types in 2023 and individual special CWE vulnerability types, and functional test functions have been written for these 10 vulnerability code files. The main purpose of this dataset is to verify whether the designed prompt template can promote the model's understanding of the vulnerable code and improve the repair efficiency on a small and relatively simple dataset.

The ExtractFix dataset [10] is a real-world vulnerability dataset. Vulnerabilities are collected from real-world projects. The projects are complex, so it is difficult to manually write test codes. However, some vulnerabilities in this dataset provide test suites themselves, for easy evaluation. The purpose of using this dataset is to verify that hint templates can improve vulnerability remediation rates in real-world vulnerability scenarios.

Vulfix is a dataset of more than 1700 vulnerabilities and their human-committed fixes collected from CVEFixes [11] and Big-Vul [12]. This data set is part of the verification data set used in the work of Fu et al. The purpose of this dataset is to verify the repair effect of hint templates in a large number of real-world vulnerabilities, and to compare with the work of VulRepair.

E. Prompt Template Design

We argue that LLMs trained on large datasets may inadvertently acquire the ability to multitask, even in the "zero-shot" setting. Therefore, a large-scale program language model trained by a program language can tap more capabilities and perform richer tasks by carefully constructing "prompt".In order to design the prompt template, we have studied a large number of open source engineering project codes and their historical versions. We believe that the prompt information can contain description information of vulnerabilities, or a description for explaining the function of the code, or indicate where the indication should be markup for repair. By researching the commits on Github, we found that the prompt information may also include tags such as "Bug Version". At the same time, source code can also appear in comments, because we find that "commented out" source code often appears in code submissions, which will show both fixed code and bug code as part of the bug fixing process.

Based on the above information, we designed the following four prompt templates. A control group without any cue template attached was additionally added. Pormpt-0 does not add any prompt information; Prompt-1 deletes the vulnerable parts other than the method name, and adds comments: Generate Fixed Code For CWE-xx; Prompt-2 first adds the description

information of the vulnerability in the form of comments (this information is provided by the static analysis tool Obtained) Vulnerability Message: [message], and then prompt in the next line: Fix The Vulnerability After. Finally, carry the first 5 words of the function body in the next line to guide the model to generate repair code instead of generating comments. Prompt-3 will add vulnerability codes in the form of comments on the basis of Prompt-2, in order to give the model richer context information to help the model complete the task of automatic vulnerability repair. Based on Prompt-3, Prompt-4 adds a brief manual description of the vulnerability repair task. These templates can change the amount of context provided to the LLM, from providing no information to rich comments and prompts, where even slight changes in the prompt words can even affect the output of the model.

IV. Experiment And Analysis of Results

A. Prompt Function of the Template

We conducted experiments on three data sets in the manner described above and compiled the experimental results, as shown in Figure 2. After using the prompt template of Prompt-1, the compilation pass rate has increased, and the bug fix rate has also increased. This improvement is most obvious in Prompt-3. What is confusing is that after using the Prompt-4 prompt template, the vulnerability repair rate began to decline, which means that the model's understanding of the vulnerability repair task has deviated.

Why does the repair efficiency increase from Prompt-0 to Prompt-3? According to our analysis, from Prompt-0 to Prompt-3, all the information added in the prompt template is beneficial to the model. For example, compared to Prompt-0, Prompt-1 adds instructions to fix the vulnerability and specifies the type of vulnerability, which allows the model to understand more precisely what we want the model to do. Compared with Prompt-1, Prompt-2 adds vulnerability description information. We think this information is very critical, which is why the improvements here are huge. Vulnerability description information contains a large amount of information about vulnerability characteristics, which is beneficial to model understanding. However, Prompt-4 causes the repair efficiency of the model to decrease. We believe this is because the information is too lowly relevant to the vulnerable code and too subjective. Different people may have different descriptions of the same content, which is not conducive to the reasoning of the model we choose.

Conclusion: The prompt template is evaluated from the two perspectives of compilation pass rate and repair rate, both of which prove that it can effectively improve the efficiency of vulnerability repair, but an overly complex template will affect the repair of vulnerabilities.

B. Text Generation and Text Filling

Considering the modes supported by the model, we can get the experimental results shown in Figure 3. The result shown on the PromptFix dataset is that the repair effect of the

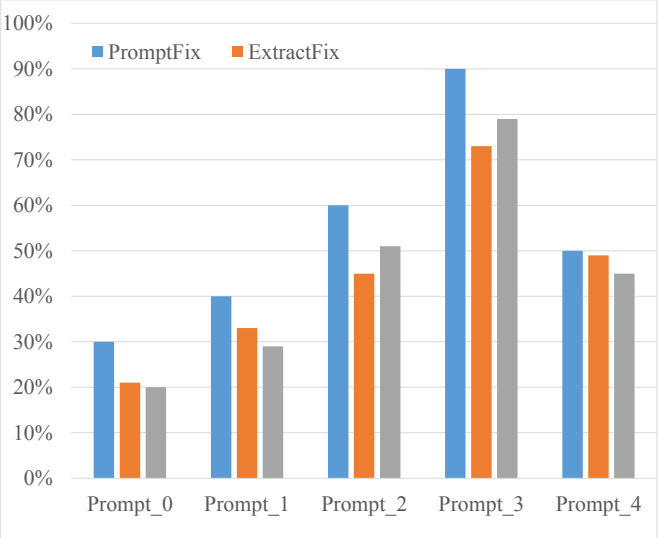

Fig. 2. The influence of the prompt template on the repair effect

generative model is better. In our opinion, it has to do with what we do with the dataset. In the Prompt dataset, since the vulnerable code is manually written, the number of lines is small, so the function where the vulnerable line is located is directly deleted and then modified with the prompt template as input. This means that the input does not provide the correct code after the vulnerable line, which causes the model to infer the repair code in a left-to-right or text-like direction. Obviously, the generative model is more suitable for this input. However, in real-world datasets, since the vulnerable code is in a very large function body, it cannot be directly modified with the prompt template and used as the input of the model. We intercepted and transformed it and then cooperated with the prompt template, which means that the final input to the model contains part of the safe code after the vulnerable line. Such an input construction more closely matches the text-padded model, thus leading to better inpainting performance of the padded model on real-world vulnerability datasets.

Conclusion: The generative model and the model that supports text filling have a certain influence on the effect of vulnerability repair, but this influence comes from the preprocessing of the vulnerability code.

C. Iterative Repair

In this section, we discuss how much the addition of the iterative repair process improves the repair effect of the vulnerability repair task, and the results are shown in Figure 4. Since the modified vulnerability code input to the model is the same, and the model is required to generate 50 outputs for one input, this will cause the model to generate a large number of repeated repair codes, and limit the model to perform vulnerability repair codes in a wider range reasoning work, so we consider increasing the working mode of iterative repair. Iterative repair will preprocess the error repair generated by the model again, and add new annotations as prompt information.

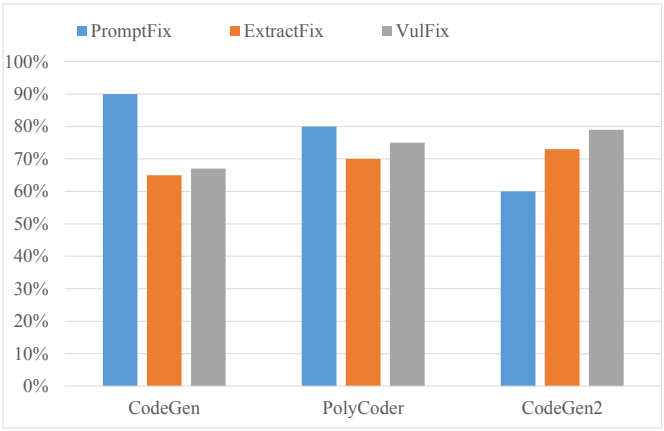

Fig. 3. The influence of model type on the repair effect (best result)

This means that each time after a round of iterative preprocessing, the input code will have an error repair code of the type of vulnerability and the description information of the error, and the model can avoid going to the same or similar error in the next reasoning process direction of reasoning. At the same time, due to the change of input, the repeated output code generated by the model is also significantly reduced. However, due to the limitation of the input length of the model and the consideration of repair time, the iteration round cannot be set too long. In other words, it is not worth paying high repair costs in order to achieve the best repair effect.

Conclusion: Iterative learning can effectively reduce the proportion of repetitive vulnerability repair codes generated by the model. However, a higher iteration round cannot be set, we think that setting 2-3 iterations is an appropriate choice.

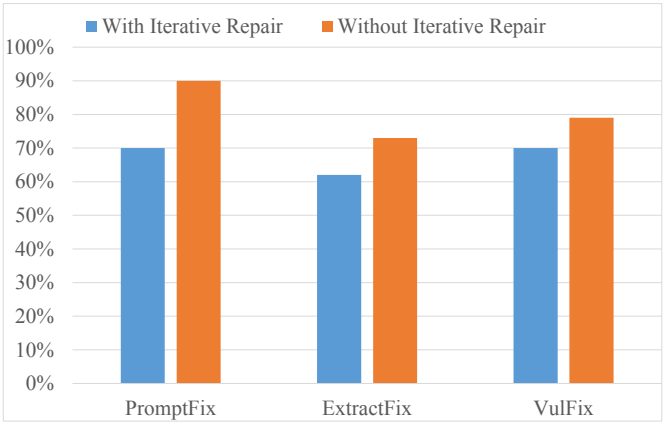

Fig. 4. Influence of Iterative Process on Restoration Efficiency(best result)

V. Conclusion

This paper proposes a novel vulnerability automatic repair workflow and designs reasonable prompt templates to tap LLM's ability to target different downstream task types in the vulnerability automatic repair task domain. Our experiments prove that reasonable prompt templates can significantly improve the efficiency of automatic vulnerability repair.Therefore, the characteristics of the model, the training method, and the density of the prompt information must be fully considered during the prompt template design process. The addition of iterative repair can also significantly improve the effect of automatic vulnerability repair, but too many rounds should not be set. Our experiments demonstrate the value of our work in automatic vulnerability repair. Our work provides a meaningful reference on how to utilize open source LLM and fully exploit the potential of LLM through suitable prompt templates.

Acknowledgment

This work was supported by the National Key Research and Development Program(2023YFB3106400, 2023QY1202),the National Natural Science Foundation of China (U1836210), and the Key Research and Development Science and Technology of Hainan Province (GHYF2022010).

References

[1] Carly Burdova. 2021. What Is EternalBlue and Why Is the MS17-010 Exploit Still Relevant. https://www.avast.com/ceternalblue
[2] Morgan McFall-Johnsen. 2020. Catastrophic software errors doomed Boeing's airplanes and nearly destroyed its NASA spaceship. Experts blame the leadership's 'lack of engineering culture'. https://www.businessinsider.com/boeingsoftware-errors-jeopardized-starliner-spaceship-737-max-planes-2020-2
[3] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. IEEE Trans. Software Eng. 45, 1 (2019), 34–67.
[4] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), May 2021, pp. 1161–1173, iSSN: 1558-1225.
[5] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A Comparative Study of Automatic Program Repair Techniques for Security Vulnerabilities. In 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE. 196–207.
[6] XIA C, WEI Y, ZHANG L. Practical Program Repair in the Era of Large Pre-trained Language Models[J]. 2022.
[7] Nijkamp E, Pang B, Hayashi H, et al. Codegen: An open large language model for code with multi-turn program synthesis[J]. arXiv preprint arXiv:2203.13474, 2022.
[8] Xu F F, Alon U, Neubig G, et al. A systematic evaluation of large language models of code[C]//Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. 2022: 1-10.
[9] Nijkamp E, Hayashi H, Xiong C, et al. Codegen2: Lessons for training llms on programming and natural languages[J]. arXiv preprint arXiv:2305.02309, 2023.
[10] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction," ACM Transactions on Software Engineering and Methodology, vol. 30, no. 2, pp. 1–27, Mar. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3418461
[11] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering. 30ś39.
[12] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In Proceedings of the 17th International Conference on Mining Software Repositories. 508ś512.