Research on the LLM-driven Vulnerability Detection System Using LProtector

Ze Sheng
Computer and Information Science
University of Pennsylvania
Philadelphia, USA
zesheng.upenn@gmail.com

Fenghua Wu
Integrated Circuit Science & Engineering
University of Electronic Science & Technology of China
Chengdu, China
qicai@std.uestc.edu.cn

Xiangwu Zuo
Computer Science & Engineering
Texas A&M University
College Station, USA
dkflame@tamu.edu

Chao Li
Graduate School of Arts and Sciences
Georgetown University
Washington, D.C., USA
cl1486@georgetown.edu

Yuxin Qiao
Computer Information Technology
Northern Arizona University
Flagstaff, USA
yq83@nau.edu

Hang Lei
LG Energy Solution(Nanjing)Co., Ltd
Nanjing, China
leihang988@gmail.com

2024 IEEE 4th International Conference on Data Science and Computer Application (ICDSCA) | 979-8-3503-6823-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDSCA63855.2024.10859408

***Abstract*—The security issues of large-scale software systems and frameworks have become increasingly severe with the development of technology. As complexity of software grows, vulnerabilities are becoming more challenging to detect. Although traditional machine learning methods have been applied in cybersecurity for a long time, there has been no significant breakthrough until now. With the recent rise of large language models (LLMs), a turning point seems to have arrived. The powerful code comprehension and generation capabilities of LLMs make fully automated vulnerability detection systems a possibility. This paper presents LProtector, an automated vulnerability detection system for C/C++ codebases based on GPT-4o and Retrieval-Augmented Generation (RAG). LProtector performs binary classification to identify vulnerabilities in target codebases. To evaluate its effectiveness, we conducted experiments on the Big-Vul dataset. Results show that LProtector outperforms two state-of-the-art baselines in terms of F1 score, demonstrating the potential of integrating LLMs with vulnerability detection.**

***Keywords*—Large Language Models, Deep Learning, Defect Detection, Operating System, Cybersecurity, Software Engineering**

I. Introduction

AI has significantly progressed in various defect detection areas over a long period of time. An example is when Wang, et al. [1] utilized AI for identifying medical problems, while Wu and collaborators [2, 3] employed it for detecting Electromigration problems. Defects in software, also referred to as software vulnerabilities, pose a significant challenge in the cybersecurity field. Xu, et al. [4] demonstrate the various ways in which harm and software loss can arise from these vulnerabilities. Several restrictions are associated with conventional detection methods. Yao et al. [5] and Li et al. [6] discovered that Automated Program Repair (APR) depends on predetermined patterns or strategies to create patches. Nevertheless, the patches do not meet the expected quality standards. Code generated by APR may successfully meet certain test cases, but it could still struggle to address the underlying issue, leading to ineffective outcomes in different situations.

Klees et al. [7], Kai et al. [8], and Manès, et al. [9] propose that while fuzzing tests are good at uncovering memory management errors, they are not as efficient at identifying intricate problems like race conditions and privilege escalation. Understanding the goals and procedures of a program is crucial in identifying logical errors, as depending only on fuzzing is insufficient for detecting and correcting them. Drawbacks are also linked to Static Analysis Tools (SAT). Pereia [10] highlighted that SAT tools generate many incorrect results because they do not take into account dynamic factors such as variable value changes in real-time. They frequently sound the alarm for issues that are not real. An instance could be discovering a possible buffer overflow in a code path that is not used. Additionally, Liu et al. [11] pointed out the difficulties SAT faces when handling intricate dynamic behaviors, such as dynamic memory allocation or conditional branches, leading to constraints in identifying dynamic vulnerabilities such as race conditions. .These limitations highlight the need for more advanced methods. Similar innovations are needed for road damage detection. Han-Cheng Dan et al. [12] successfully improved detection accuracy and efficiency using the enhanced YOLOv7 algorithm, demonstrating the advantages of improved model strategies.

In contrast, LLMs have powerful code generation and understanding capabilities [13-15], along with a rich knowledge base and strong generalization ability [16-18]. For instance, Tan et al. demonstrated that neural networks could effectively convert textual descriptions into 3D models using encoder-decoder architectures, showcasing the potential of LLMs to handle multi-modal tasks across various domains. Zhang et al. finds the black-box LLMs like GPT-4o can label textual datasets with a quality that surpasses that of skilled human annotators, which offers a new perspective for automated software defect detection, as LLMs can achieve more efficient training and labeling when handling large volumes of unlabeled data. Thus, we selected the GPT-4o, which is currently one of the most capable models, as the AI agent for LProtector. This ensures good robustness even in systems with strong interference. We used the Big-Vul dataset to evaluate how well LProtector works by measuring its performance against VulDeePecker and Reveal. To enhance LProtector's cybersecurity knowledge, we used RAG methods to pick 500 random instances of CWE/CVE from the Big-Vul dataset and stored them in a vector database.

II. Proposed Methodology

A. Architecture

Fig. 1 shows the architecture diagram of LProtector. In LProtector, all input data is code snippets from the Big-Vul Dataset.

For data preprocessing, Pandas is used to extract CWE-ID, code description, vulnerability name and code snippets in the dataset. Then the metadata will be stored into a .json file which will be later embedded into vectors by OpenAI Embedding Algorithm. In this paper, ChromaDB is the vector database.

The input data is first mapped into vectors through word embedding. Deep reinforcement learning shows similar potential for complex decision-making, such as in autonomous navigation. Then, using an Augmented Prompt, the vectors are queried against the database to obtain the relevant results.Next, with Chain of Thought (CoT) prompt engineering method, the AI agent attempts to determine whether the code block contains a vulnerability and performs binary classification, where 1 indicates "yes" and 0 indicates "no".

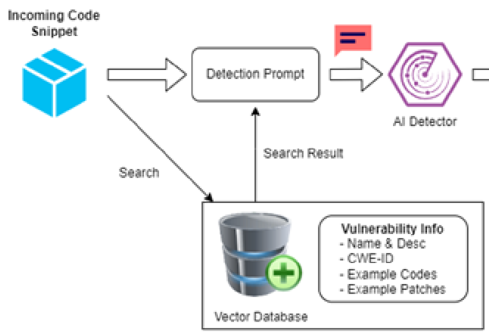


Fig. 1. Architecture of LProtector

B. Retrieval-Augmented Generation

RAG was proposed for a long time, but with the rise of LLMs, their powerful understanding capability can significantly improve the accuracy and precision of RAG's retrieval results. The principle of RAG is to combine retrieval and generation into a unified framework. Similar feature selection techniques have shown robustness in dealing with complex datasets, as demonstrated by Shen et al.

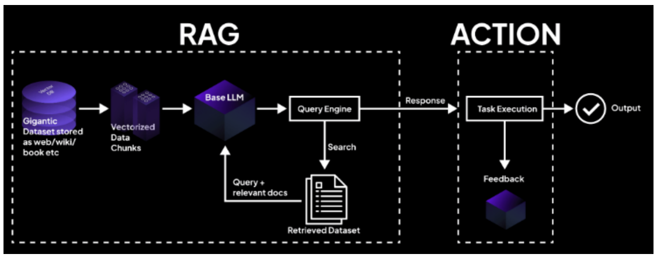


Fig. 2. Architecture of RAG

Fig. 2 shows the architecture of RAG. RAG enhances response quality by combining retrieval and generative modeling. Given a query q, the retriever selects relevant documents D = {d_1, d_2, . . . , d_N } from a knowledge base. The retrieval step is represented as:

$$P(d_i|q) = \frac{e(q)\cdot e(d_i)}{\|e(q)\|\|e(d_i)\|} \tag{1}$$

where $e(q)$ and $e(d_i)$ are the embeddings of the query and document di. This is calculated using cosine similarity.

The generator uses these retrieved documents to generate a response y:

$$P(y|q) = \sum_{d \in D} P(y|q,d)P(d|q) \tag{2}$$

where $P\,(y|q, d)$ is the conditional probability of generating response y given query q and document d.

OpenAI embeddings are vector representations capturing the semantic meaning of text. Ensemble methods have been shown to improve financial forecasting performance. Given input x, the embedding $e(x) \in Rd$ is generated by a pre-trained model. The similarity between two embeddings $e(x_1)$ and $e(x_2)$ is calculated as:

$$sim\big(e(x_1), e(x_2)\big) = \frac{e(x_1)\cdot e(x_2)}{\|e(x_1)\|\|e(x_2)\|} \tag{3}$$

where represents the dot product, and $\|e(x)\|$ is the Euclidean norm.

ChromaDB is a vector database for managing highdimensional embeddings. The retrieval process finds the embedding $e_j \in E$ that minimizes the distance to a query embedding e(q):

$$e^* = \arg\min_{e_j \in E} \|e(q) - e_j\| \tag{4}$$

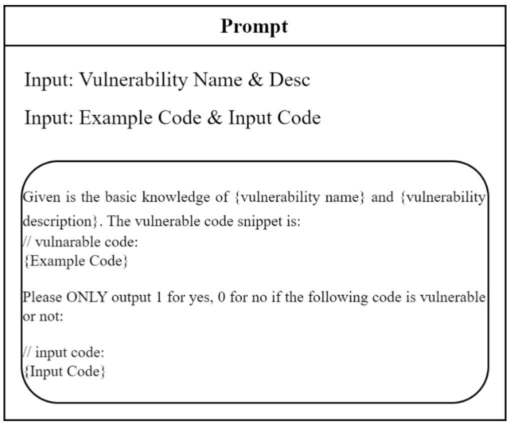
where $\|e(q)\text{-}e_j\|$ represents the distance (typically Euclidean or cosine) between the query embedding $e(q)$ and a candidate embedding e_j.

C. Vulnerability Detection

LProtector uses binary classification for vulnerability detection, techniques that have also been reviewed in other AI/ML contexts. It was observed that GPT-4o identifies most input code snippets as vulnerable. Therefore, it is important to maintain balance in our test dataset to achieve unbiased detection results, which will be further investigated during the experiment section.

The OpenAI Embedding algorithm is used to transform every new input code snippet into word vectors. However, the revised embeddings are not saved directly in the database. Instead, they are compared with the existing vectors in the database in order to identify the top 5 most similar entries. GPT-4o later examines these records and selects the best one based on contextual similarities. The improved hint is created, as shown in Fig. 3.

The input is put into GPT-4o, resulting in the final classification result. Basically, LProtector uses a vector database offline to perform semantic or pattern matching, ensuring that the achieved outcomes provide the necessary context for accurate vulnerability identification, as shown in Fig. 4.



Prompt

Input: Vulnerability Name & Desc

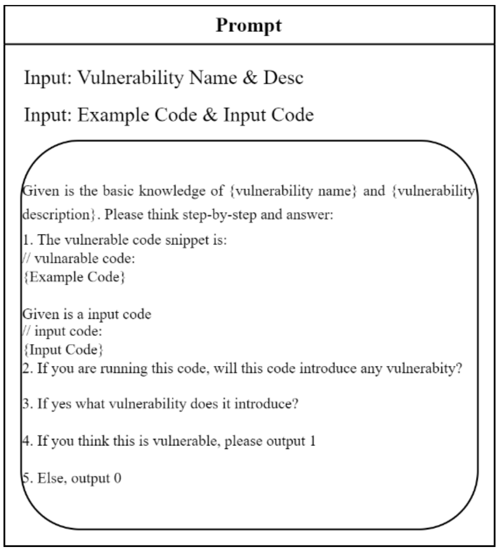
Input: Example Code & Input Code

Given is the basic knowledge of {vulnerability name} and {vulnerability description}. The vulnerable code snippet is:
// vulnarable code:
{Example Code}

Please ONLY output 1 for yes, 0 for no if the following code is vulnerable or not:

// input code:
{Input Code}

Fig. 3. Prompt without CoT



Prompt

Input: Vulnerability Name & Desc

Input: Example Code & Input Code

Given is the basic knowledge of {vulnerability name} and {vulnerability description}. Please think step-by-step and answer:
1. The vulnerable code snippet is:
// vulnarable code:
{Example Code}

Given is a input code
// input code:
{Input Code}
2. If you are running this code, will this code introduce any vulnerability?

3. If yes what vulnerability does it introduce?

4. If you think this is vulnerable, please output 1

5. Else, output 0

Fig. 4. Prompt with CoT

III. EXPERIMENTS

A. Experiments Procedure

The Big-Vul dataset has a severe data imbalance problem, where the number of non-vulnerable samples significantly outweighs the vulnerable samples. Specifically, as shown in Table I, the ratio of vulnerable to non-vulnerable samples is only 5.88%.

TABLE I. STATISTICS OF BIG-VUL DATASET

Dataset	**Data**		
	Samples	***Vul***	***Non-Vul***
Big-Vul	179,299	10,547 (5.88%)	168,752 (94.12%)

In order to enhance our experiments, we initially conducted sampling to equalize the data, aiming for a 1:1 balance between vulnerable and non-vulnerable test data, which produced a total of 5,000 test instances. Next, we selected 500 vulnerable cases from the non-test data at random and collected the associated vulnerable code sections, which were saved in a vector database using RAG. Afterward, we conducted a comparison of the results by running LProtector together with VulDeePecker and Reveal. Similar designs are used to study the effects of visual field on thermal perception.

Then, to assess the internal workings of LProtector, we investigated how two important factors affected its performance:

1) With/Without RAG:

To investigate the effect of RAG on LProtector's performance, we assess its results on the same dataset without the RAG component.

2) With/Without CoT:

We assessed the performance of LProtector on the same dataset without using CoT to understand its impact on vulnerability detection.

Then we need to evaluate the performance of LProtector with no RAG and CoT, which means purely LLM.

B. Performance Metrics

To evaluate the performance, We use *Accuracy, Precision, Recall, F_1* [6].

1) Accuracy: Accuracy is the most intuitive metric, representing the proportion of correctly classified instances (both vulnerabilities and non-vulnerabilities) to the total instances. It is calculated as the ratio of true positives (*TP*) and true negatives (*TN*) to the sum of all predicted classes, including false positives (*FP*) and false negatives (*FN*). Although accuracy is a useful metric, it can be misleading in the case of imbalanced datasets, where one class significantly outweighs the other, as is the case in our dataset.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \tag{5}$$

2) Precision: Precision measures the proportion of correctly predicted positive instances out of all instances predicted as positive. It is particularly important in cases where the cost of false positives is high. Precision is sensitive to the number of false positives, meaning that a model with high precision minimizes the occurrence of incorrectly classifying non-vulnerabilities as vulnerabilities.

$$Precision = \frac{TP}{TP+FP} \tag{6}$$

3) Recall: Recall (also known as sensitivity or true positive rate) quantifies the model's ability to correctly identify actual positive instances. It is especially important in contexts where missing positive instances (false negatives) can have severe consequences. For our vulnerability detection task, recall reflects the model's ability to correctly identify vulnerable code.

$$Recall = \frac{TP}{TP+FN} \tag{7}$$

4) FI Score: Finally, the F_1 Score is the harmonic mean of Precision and Recall, providing a balanced measure of the two when there is a tradeoff between them. The F_1 score is particularly useful when the dataset is imbalanced, as it provides a single metric that accounts for both false positives and false negatives. It offers a more nuanced measure of performance, especially when neither Precision nor Recall can fully capture the model's efficacy in isolation.

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision+Recall} \tag{8}$$

By combining these metrics, we can assess the model's performance across multiple dimensions, ensuring a balanced understanding of its classification capabilities. Each metric highlights different aspects of performance, providing insights into areas where the model excels and where improvements may be necessary.

C. Experiment Results

In the initial experiments, we compared the performance of VulDeePecker, Reveal, and LProtector on the Big-Vul dataset. As shown in the Table II, LProtector outperformed the other two baselines across all metrics.

TABLE II. OVERALL PERFORMANCE

Baseline	**Metrics (%)**			
	Accuracy	***Precision***	***Recall***	***F1 Score***
VulDeePecker	81.19	38.44	12.75	19.15
Reveal	87.14	17.22	34.04	22.87
LProtector	89.68	30.52	38.07	33.49

LProtector achieved an accuracy of 89.68%, which is higher than VulDeePecker's 81.19% and Reveal's 87.14%, indicating its stability in classifying both vulnerable and non-vulnerable samples. Although VulDeePecker had a higher precision (38.44%) compared to LProtector (30.52%), its recall was only 12.75%, much lower than LProtector's 38.07%, meaning it struggled to capture true vulnerabilities. In contrast, LProtector effectively reduced false positives while identifying more real vulnerabilities. As a result, LProtector achieved an F1 score of 33.49%, demonstrating a notable improvement over VulDeePecker's 19.15% and Reveal's 22.87%, showcasing its balanced performance in terms of precision and recall.

TABLE III. LPROTECTOR VARIABLE TEST

Variables	**Metrics (%)**			
	Accuracy	***Precision***	***Recall***	***F1 Score***
RAG + CoT	89.68	30.52	38.07	33.49
No RAG	76.42	22.71	25.30	23.92
No CoT	79.73	24.85	28.40	26.51
No RAG & CoT	68.19	17.85	19.42	18.61

Removing the RAG component led to a significant drop in LProtector's performance(Table III). The accuracy decreased to 76.42%, which is lower than VulDeePecker's 81.19%, and the F1 score dropped to 23.92%. This result suggests that RAG plays a critical role in retrieving relevant context for accurate vulnerability detection. Without RAG, LProtector cannot get access to prior knowledge, which directly impacts its ability to correctly identify vulnerabilities.

Removing CoT causes a drop in performance, resulting in accuracy decreasing to 79.73% and the F1 score decreasing to 26.51%. This falls just under the baseline accuracy of 81.19%, suggesting that CoT improves the model's reasoning skills.

The accuracy of LProtector's forecasts decreases without CoT, impacting precision and recall.

When both RAG and CoT were removed, LProtector's accuracy fell sharply to 68.19%, and its F1 score was reduced to 18.61%. This performance is significantly lower than all other configurations, highlighting that RAG and CoT are both essential for LProtector's effectiveness. Without these components, the model struggles to process and interpret the code, making it almost ineffective in detecting vulnerabilities. In the two experiments, we initially showed that LProtector surpasses VulDeePecker and Reveal on the Big-Vul dataset, obtaining better F1 scores and overall performance. Next, we examined the inner workings by disassembling the RAG and CoT parts individually. The findings indicated that the model was most affected by RAG, as its removal caused a notable decrease in both accuracy and F1 score. Getting rid of CoT had a minor impact, but still led to performance dropping below the original level. LProtector's performance significantly decreased to its lowest level when RAG and CoT were both eliminated, underscoring the essential nature of these two components for successful vulnerability detection.

IV. CONCLUSION

In the latest studies on utilizing LLMs for identifying vulnerabilities, code is often viewed as regular text that is fed directly into the model, without the necessary contextual information and domain expertise needed for precise detection. In response to these restrictions, we suggest LProtector, a new vulnerability detection system using advanced retrieval methods and Chain of Thought (CoT) reasoning to improve prediction accuracy. LProtector uses RAG to insert expertise specific to domains, enhancing its comprehension of intricate code structures. The Big-Vul dataset experiment showed that LProtector outperforms other baseline methods with a 33.49% F1 score, surpassing VulDeePecker and Reveal. Additionally, analysis of components shows that both RAG and CoT are essential for the model's success, with RAG playing the biggest role in enhancing prediction accuracy. In our future efforts, we aim to improve the retrieval and reasoning methods of LProtector to increase its ability to detect threats. Furthermore, we will investigate the potential of utilizing LProtector in various intricate software systems, including mobile apps and cloud environments, to assess its capability to be applied broadly. Ultimately, our goal is to explore combining automated vulnerability repair methods with LProtector, which presents a difficult but promising avenue for future studies.

ACKNOWLEDGMENT

We would like to express our gratitude to the various institutions for their valuable contributions to data collection. Additionally, we extend our appreciation to all members of our research team for their dedication and collaborative efforts throughout the development of LProtector.

REFERENCES

[1] Z. Li, S. Dutta, and M. Naik, "LLM-Assisted Static Analysis for Detecting Security Vulnerabilities," arXiv:2405.17238, May 2024, doi:10.48550/arXiv.2405.17238.

[2] K. Huang et al., "A Survey on Automated Program Repair Techniques," arXiv:2303.18184, May 2023, doi: 10.48550/arXiv.2303.18184.

[3] Dan, P. Yan, J. Tan, Y. Zhou, and B. Lu, "Multiple distresses detection for Asphalt Pavement using improved you Only Look Once Algorithm based on convolutional neural network," International Journal of Pavement Engineering, vol. 25, no. 1, 2024.

[4] F. Liu et al., "Exploring and Evaluating Hallucinations in LLM-Powered Code Generation," arXiv:2404.00971, May 2024, doi:10.48550/arXiv.2404.00971.

[5] Z. K. Kebaili, D. E. Khelladi, M. Acher, and O. Barais, "An Empirical Study on Leveraging LLMs for Metamodels and Code Co-evolution," in European Conference on Modelling Foundations and Applications (ECMFA 2024), 2024.

[6] Z. Sheng, Y. Li, Z. Li, and Z. Liu, "Displacement Measurement Based on Computer Vision," in 2019 International Conference on Sensing, Diagnostics, Prognostics, and Control (SDPC), Aug. 2019, pp. 448-453, doi: 10.1109/SDPC.2019.00087.

[7] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," arXiv:2201.11903, Jan. 2023, doi:10.48550/arXiv.2201.11903.

[8] X. Wang and D. Zhou, "Chain-of-Thought Reasoning Without Prompting," arXiv:2402.10200, May 2024, doi:10.48550/arXiv.2402.10200.

[9] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," arXiv:2005.11401, Apr. 2021, doi:10.48550/arXiv.2005.11401.

[10] L. Xu, J. Liu, H. Zhao, T. Zheng, T. Jiang, and L. Liu, "Autonomous Navigation of Unmanned Vehicle Through Deep Reinforcement Learning," arXiv:2407.18962, Jul. 2024, doi:10.48550/arXiv.2407.18962.

[11] X. Du et al., "Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG," arXiv:2406.11147, Jun. 2024, doi: 10.48550/arXiv.2406.11147.

[12] ScienceDirect. "Confusion Matrix - an overview — ScienceDirect Topics." https://www.sciencedirect.com/topics/engineering/confusion-matrix (accessed).

[13] Y. Cheng et al., "LLM-Enhanced Static Analysis for Precise Identification of Vulnerable OSS Versions," arXiv:2408.07321, Aug. 2024, doi: 10.48550/arXiv.2408.07321.

[14] Y. Sun et al., "LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs' Vulnerability Reasoning," arXiv:2401.16185, Sep. 2024, doi: 10.48550/arXiv.2401.16185.

[15] Q. Xing, C. Yu, S. Huang, Q. Zheng, X. Mu, and M. Sun, "Enhanced Credit Score Prediction Using Ensemble Deep Learning Model," arXiv:2410.00256, Sep. 2024, doi: 10.48550/arXiv.2410.00256.

[16] C. Wu, Z. Yu, and D. Song, "Window views psychological effects on indoor thermal perception: A comparison experiment based on virtual reality environments," E3S Web of Conferences, vol. 546, p. 02003, 2024.

[17] X. Zuo, A. Jiang, and K. Zhou, "Reinforcement prompting for financial synthetic data generation," The Journal of Finance and Data Science, vol. 10, p. 100137, Dec. 2024, doi: 10.1016/j.jfds.2024.100137.

[18] Z. Zhang, J. Zhang, H. Yao, G. Niu, and M. Sugiyama, "On Unsupervised Prompt Learning for Classification with Black-box Language Models," arXiv:2410.03124, Oct. 2024, doi:10.48550/arXiv.2410.03124.