Detecting Code Vulnerabilities with DeepSeek and FAISS

Sarfraz Farooq1, a), Shih Yin Ooi1, 2, b), Yee Jian Chew1, 2, c)

*1Faculty of Information Science and Technology (FIST), Multimedia University, Jalan Ayer Keroh Lama, Melaka, 75450, Malaysia*

*2Centre for Advanced Analytics (CAA), COE for Artificial Intelligence, Multimedia University, Jalan Ayer Keroh Lama, Melaka, 75450, Malaysia*

*b) Corresponding author: syooi@mmu.edu.my*

*a) sarfraz.farooq@student.mmu.edu.my*

*c)chewyeejian@mmu.edu.my*

**Abstract.** Open-source software is increasingly being created and updated using generative AI, leading to increased exploits that compromise system integrity. Although traditional vulnerability detection methods such as static analysis exist, they lack contextual understanding, are less efficient, and not scalable. This research introduces a hybrid approach to software vulnerability detection, combining FAISS and static code analysis to boost the accuracy of detecting code related vulnerabilities. The study begins by formatting publicly available datasets such as the Juliet Test Suite, DiverseVul, and NVD for our use. The dataset is also processed to map Common Vulnerabilities and Exposures (CVEs) to Common Weakness Enumerations (CWEs), providing structured insights into past vulnerabilities as an additional benefit. These entries are then embedded and indexed in FAISS (Facebook AI Similarity Search) to enable efficient retrieval of vulnerabilities relevant to the provided code snippet. The framework utilizes DeepSeek-R1-Distill-Llama-8B, an AI-powered code analysis model, in conjunction with LangChain for contextual processing and interacting with the user. When a code file is provided to the system, the system first conducts a FAISS query to identify similar past vulnerabilities, which in turn are used as contextual references to guide the AI model in analyzing the code. This approach, also known as retrieval-augmented generation (RAG) helps reduce false positives, improves accuracy, and helps reduce AI hallucination. Through a lot of testing on Python and C++ code, the system was able to identify critical vulnerabilities such as SQL Injection, buffer overflows, and improper input validation.

# Introduction

The millions of lines of code that make up modern software systems are very complex and rely on extensive third-party dependencies. This complexity raises the chances of security vulnerabilities to appear and complicates their detection through manual means. Traditionally, static analysis tools were the answer to this problem. They use manually engineered rules or signature-based pattern matching but have shown limitations in both their accuracy and comprehension. They frequently generate false positives and fail to detect newer and more context-sensitive vulnerabilities [1].

To address these issues, researchers explored machine learning and deep learning solutions, especially Large Language Models (LLMs). These showed promise in interpreting, understanding, and even detecting vulnerabilities in source code. However, the cost of hosting and running these models remains a problem. Most state-of-the-art LLMs require a considerable amount of memory to load, not just standard RAM, but specifically the high-bandwidth VRAM found in modern GPUs. On the other hand, when attempting to compress or quantize these models so that they fit within limited resources, they often degrade in performance or have trouble with generating proper output.

Moreover, the model’s raw answers often include mistakes or made-up details and must be converted into a clear, uniform report showing each flaw’s type, identifier, severity, location, description, and fix. Finally, the model’s built-in knowledge can be out of date, leading it to invent vulnerabilities when it lacks up-to-date context, i.e., hallucinate [2].

# related work

Utilizing artificial intelligence to aid in the tasks of cybersecurity, such as finding software vulnerabilities, has been explored greatly in recent studies [3]. Large Language Models (LLMs) have even been used in hybrid analysis techniques to detect and fix software vulnerabilities [4]. However, despite using LLMs together with Retrieval-Augmented Generation (RAG) techniques and static analysis tools, many of these tools still face technical and practical challenges. Current static analysis tools have been known to produce a large number of false positives [1] since they are not able to understand the context of the code they deem vulnerable, thus needing manual triaging [5]. This problem persists even when these static analysis tools are paired with LLMs to provide a contextual and reasoning element. The system fails to understand complex vulnerabilities and the role of specific code snippets across an entire repository, as can be seen in IRIS [5]. In another application, called LLM4Vuln, the researchers show that adding more context to the model does not always lead to better results [6]. The context may confuse the model and lead it to generate incorrect results. Moreover, in an application named Vul-RAG, the researchers report in their experiments that the application had nearly 48% false positives caused by the RAG system retrieving unrelated vulnerability examples for the model [7]. This weakened the reliability of their RAG method. Furthermore, other systems like LProtector found that over-reporting from the GPT-4o model that they used caused it to classify almost all inputs as security vulnerabilities [8]. In addition to that, the researchers behind SafeRAG demonstrated that malicious information injected into a RAG pipeline can easily bypass filters and mislead the LLMs, thereby compromising reliability and safety of the system [9]. Last but not least, large models like LLaMA 65B require extreme amounts of GPU memory to finetune the model. Even in 16-bit precision, the GPU’s RAM requirement will be over 780 GB, while using 4-bit quantization results in around 48 GB of RAM requirement [10]. This makes it very costly and impractical for researchers and developers to explore these models on consumer-grade GPUs. Finally, there is the challenge of properly formatting the LLM’s output. The Promptfoo guide highlights this issue and warns users that LLMs are unreliable in providing JSON formats of their results and may even provide malformed or incomplete JSON output if forced [11].

To address these gaps, our system was built to offer a GPU-efficient and context-aware vulnerability detection system. It runs a 4-bit quantized and distilled version of DeepSeek-R1 fully on a single RTX 3070 Ti GPU with only 8 GB of memory. All thanks to the advanced quantization methods like NormalFloat and double quantization [10]. It also uses a carefully curated offline FAISS index of known CWE and CVE examples, instead of relying on online retrieval or open-ended databases that are prone to poisoning attacks and increased noise [12] [9]. In addition to that, during the detection phase, the RAG system retrieves the most relevant or “top known” vulnerability examples based on their similarity, which helps us avoid the irrelevant retrieval problem described in Vul-RAG [7] and reduces the chances of the model hallucinating. Finally, the system also formats the model’s response in a manner that is similar to vulnerability reports, while also keeping the raw response available for the user. It states the vulnerability type, CWE ID, severity, location, and description. If the model fails in generating a proper JSON output, the program’s fallback mechanism handles the broken JSON output and extracts relevant data [11].

# Methodology

Our implementation for the program code vulnerability detection tool combines advanced search with AI reasoning. Using flask as the backend framework, it allows users to paste code snippets or upload files directly through a web interface. In the first stage of the two-stage analysis, the system uses FAISS to retrieve similar known vulnerabilities that may exist in the provided code. These results are then fed into our chosen LLM, DeepSeek-R1-Distill\_LLaMA-8B with 4-bit quantization, along with the user's code for analysis. In the second stage, the system uses the AI generated response to re-match them with official CWE definitions and classifications. This stage is mainly for the reporting and post-processing of the CWE classifications, so they are accurately shown to the user. The reason we chose this methodology is because it prioritizes GPU efficiency while handling multiple requests at the same time.

## Retrieval-Augmented Generation (RAG) with FAISS

To tackle the issue of inaccurate results and hallucination, we have decided to use Retrieval-Augmented Generation (RAG). The RAG system will help by providing the AI with context so that it does not deviate from its course when faced with a challenging code. When the user submits their code to the program, it first converts it into a numbered vector using a transformer model, named sentence-transformers/all-MiniLM-L6-v2, from Hugging Face. These vectors are then compared against our curated dataset of known vulnerabilities using FAISS, which finds the examples that are most similar to the user’s code. Giving the AI real-world examples will help it provide more relevant and accurate analysis. As the AI is not trained on updated CWE classifications, it will often give a random CWE number in in analysis while still accurately describing the vulnerability. To improve the accuracy of the mapping, the system then takes the descriptions generated by the model in the first stage and embeds them again using the same transformer model. These are mapped against a different FAISS index that contains the official vectorized CWE classifications and definitions. This allows the system to replace the initial labels using the correct descriptions. For the time being, these searches are executed on the CPU as we are experiencing problems downloading FAISS-GPU on our system. However, this works to our benefit as the GPU is left for the more intensive task of AI analysis.

## Choice of Model: DeepSeek-R1-Distill-LLaMA-8B

We have chosen to use a distilled version of the DeepSeek-R1 model to analyze the vulnerable code. The model itself is not built to specifically analyze code and answer in a cybersecurity structured manner, but its nature of text generation and conversation will help us provide more compressive reports. Since it is a distilled version, it is smaller and faster than the LLaMA models it is based on but still understands technical language well. Moreover, it is a reasoning model, so it is particularly good at following instructions and thinking through complex problems. This really helps when the task is to analyze code for security vulnerabilities. When using the distilled model without providing it context, it may have some trouble pointing out every vulnerability in the provided code. However, combining information from the FAISS database with its analysis of new code aids it in recognizing previously missed vulnerabilities. Furthermore, the program is built to run on standard commercial-grade GPU like an RTX 3070 Ti with 8 gigabytes of VRAM, which we have been using. Even though the specified GPU is not inadequate, we still need to quantize it. The model uses 4-bit quantization through the "bitsandbytes" library, thus compressing the model even further to use less memory while retaining most of the accuracy. Specifically, it uses the Normal Float 4 (NF4) format to make the best use of the hardware available.

## Dataset and FAISS Index Construction

To make sure that we have an accurate vulnerability detection system, we used the DiverseVul dataset [13], the Juliet Test Suite (C/C++ v1.2) [14], and the official CWE-699 taxonomy (699.csv) [15] from the CWE website. The first two datasets are originally in JSON format. However, since they are two different datasets with different JSON formatting, we had to preprocess them into a unified database. The CWEs that had matching CVEs using the NVD dataset were labeled accordingly, whereas the ones that only have one of the two identifiers were labeled with “Unknown” for the missing label. This approach made sure that the dataset remained usable without inaccurate identifiers. In addition to this, we created a second FAISS index for the second stage using the official CWE entries from MITRE Corporation. This dataset is used to enhance and correct the CWE classifications between the model-generated descriptions and official CWE definitions.

## Program Flow

Our system uses Flask as the backend framework to analyze the code for vulnerabilities. First, after the user submits code for analysis, the program checks if its main components are loaded and ready. These components are the FAISS index for similarity search and the DeepSeek-R1 model for analysis. If they are not loaded, the program loads the FAISS index using the CPU, whereas the 4-bit quantized LLM is loaded on the GPU. Next, the program checks the type of upload provided by the user, whether it is a single snippet, file, or directory. Should it be a directory that is uploaded, then the program begins to scan multiple files at once using parallel threads. Each piece of code that is uploaded gets analyzed by finding the top-known similar vulnerabilities inside the FAISS index. Afterwards, a detailed prompt will be created by using these examples as context, which is then processed by the LLM to find potential vulnerabilities. The results are checked for JSON formatting and parsed. If the output from the LLM is not structured properly, the program falls back to regex and pattern parsing. Additionally, the system does a second FAISS search by using the LLM-generated descriptions and comparing them to the vectorized CWE definitions to correct the CWE labels assigned by the LLM. An overall diagram of this entire process is visualized in Figure 1.

# Findings

We evaluated our system across 15 different CWE categories, using 5 alternative test cases per class, for a total of 75 test cases. These samples were made to be different in their syntax and logic so that we can assess the system's generalization ability. As shown in Table 1, our AI + RAG pipeline outperformed our chosen baseline static analysis tool bandit, in detection coverage. It was successfully able to identify vulnerabilities in all test cases for 13 out of the 15 categories. The main reason why Bandit had a lower detraction rate was because of its inability to understand the code's context. Since, many vulnerabilities could only be recognized by analyzing control flow, function usage, or the broader logic rather than just syntax. In contrast, the LLM, being a reasoning model, was able to reason about code behavior using both the input and retrieved logic.

A diagram of a flowchart

AI-generated content may be incorrect.

**FIGURE 1.** Workflow diagram of the vulnerability detection pipeline

While detection was consistently strong, CWE classification was a major issue in the initial stage. Some vulnerabilities, such as SQL Injection, Code Injection, and Deserialization were correctly mapped for most of the test cases. However, other vulnerabilities like CSRF and Authorization failures were difficult for the LLM to precisely classify. The system addressed this in the second stage by re-matching them against the FAISS index built from the official CWE software vulnerabilities dataset. This second stage correction step helped improve the overall labeling precision, especially for vulnerabilities that are well defined. It did still try to overfit some CWE classifications from the LLM's own reasoning, but using prompt engineering for CWE-Top-25 vulnerabilities made sure that the top retrieved classification will be the correct in this experiment even if the model overfits more. These results show how our system addresses the shortcomings found in the existing tools, such as context misinterpretation, irrelevant retrievals, and formatting failures that were highlighted in previous works.

# CONCLUSION

This study shows the feasibility and effectiveness of combining similarity search with large language models for the purpose of detecting program code vulnerabilities. By integrating FAISS-based retrieval with a lightweight reasoning model DeepSeek-R1-Distill-LLaMA-8B, our system is able to detect complex vulnerabilities in code such as the ones in CWE top 25. It can do that even when these are diverse or hidden deep in the context of the code. The RAG system provides relevant examples to the LLM as context that help guide it towards more accurate detection.

Further work on this system will focus on fixing the issues it currently faces as well as exploring other methods to improve accuracy. The first will be to test a model that has been designed specifically for coding, like the DeepSeek-Coder. This should allow the model to generate a proper fix to mitigate the vulnerability in code rather than just an explanation we receive in most cases. DeepSeek-Coder also has a larger context window which will allow us to provide it with more context and scan larger directories. Moreover, we will explore larger and more recent datasets to integrate into the FAISS index. These will be isolated according to programming language instead of merging into one large dataset. This will reduce the risk of false positives that can occur when the AI overfits a vulnerability from one language to another. Finally, we will research ways to optimize the system further and increase the speed for each scan so that larger source codes can be scanned for vulnerabilities in a short time.

**TABLE 1.** Detection results across CWE types using Bandit and AI + RAG

|  |  |  |  |
| --- | --- | --- | --- |
| **Vulnerability** | **Bandit Detection** | **AI + RAG Detection** | **AI CWE Accuracy** |
| CWE-787: Out-of-bounds Write | 2/5 | 5/5 | 3/5 |
| CWE-89: SQL Injection | 4/5 | 5/5 | 5/5 |
| CWE-352: Cross-Site Request Forgery (CSRF) | 0/5 | 1/5 | 0/5 |
| CWE-22: Path Traversal | 0/5 | 5/5 | 5/5 |
| CWE-78: OS Command Injection | 2/5 | 2/5 | 2/5 |
| CWE-862: Missing Authorization | 0/5 | 5/5 | 0/5 |
| CWE-434: Unrestricted File Upload | 0/5 | 5/5 | 3/5 |
| CWE-94: Code Injection | 4/5 | 5/5 | 5/5 |
| CWE-20: Improper Input Validation | 0/5 | 5/5 | 1/5 |
| CWE-287: Improper Authentication | 0/5 | 5/5 | 0/5 |
| CWE-269: Improper Privilege | 0/5 | 4/5 | 0/5 |
| CWE-502: Deserialization of Untrusted Data | 4/5 | 5/5 | 5/5 |
| CWE-200: Exposure of Sensitive Information to an Unauthorized Actor | 0/5 | 4/5 | 0/5 |
| CWE-918: Server-Side Request Forgery (SSRF) | 0/5 | 5/5 | 2/5 |
| CWE-476: NULL Pointer Dereference | 0/5 | 5/5 | 5/5 |

# References

1. F. Hashmat, Z.A. Aljaali, M. Shen, and A. Machiry, “Insights from Running 24 Static Analysis Tools on Open Source Software Repositories,” (Springer Nature Switzerland, 2025), pp. 225–245.
2. N. Jiang, Q. Li, L. Tan, and T. Zhang, “Collu-Bench: A Benchmark for Predicting Language Model Hallucinations in Code,” (2024).
3. E. Katsadouros, C.Z. Patrikakis, and G. Hurlburt, “Can Large Language Models Better Predict Software Vulnerability?,” IT Prof **25**(3), 4–8 (2023).
4. A.A. Mahyari, “Harnessing the Power of LLMs in Source Code Vulnerability Detection,” in *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*, (IEEE, 2024), pp. 251–256.
5. Z. Li, S. Dutta, and M. Naik, “IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities,” (2024).
6. Y. Sun, D. Wu, Y. Xue, H. Liu, W. Ma, L. Zhang, Y. Liu, and Y. Li, “LLM4Vuln: A Unified Evaluation Framework for Decoupling and Enhancing LLMs’ Vulnerability Reasoning,” (2024).
7. X. Du, G. Zheng, K. Wang, J. Feng, W. Deng, M. Liu, B. Chen, X. Peng, T. Ma, and Y. Lou, “Vul-RAG: Enhancing LLM-based Vulnerability Detection via Knowledge-level RAG,” (2024).
8. Z. Sheng, F. Wu, X. Zuo, C. Li, Y. Qiao, and L. Hang, “LProtector: An LLM-driven Vulnerability Detection System,” (2024).
9. X. Liang, S. Niu, Z. Li, S. Zhang, H. Wang, F. Xiong, J.Z. Fan, B. Tang, S. Song, M. Wang, and J. Yang, “SafeRAG: Benchmarking Security in Retrieval-Augmented Generation of Large Language Model,” (2025).
10. T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, “QLoRA: Efficient Finetuning of Quantized LLMs,” (2023).
11. I. Webster, “LLM evaluation techniques for JSON outputs,” Promptfoo, (n.d.).
12. M. Ibiyo, T. Louangdy, P.T. Nguyen, C. Di Sipio, and D. Di Ruscio, “Detecting Malicious Source Code in PyPI Packages with LLMs: Does RAG Come in Handy?,” (2025).
13. Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, “DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection,” in *ACM International Conference Proceeding Series*, (Association for Computing Machinery, 2023), pp. 654–668.
14. NSA Center for Assured Software, “Juliet C/C++ 1.2,” National Institute of Standard and Technology, (2013).
15. MITRE Corporation, “CWE-699: Software Development, Common Weakness Enumeration,” Common Weakness Enumeration (CWETM), (2025).