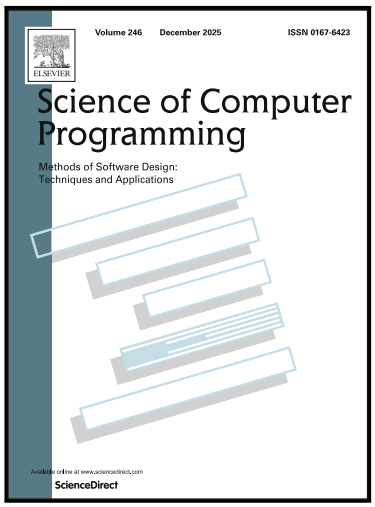

Software Vulnerability Detection via Multimodal Retrieval and Hierarchical Decision Making

Zhoufu Liu, Junhua Chen, Zijie Huang

PII: S0167-6423(26)00057-2
DOI: https://doi.org/10.1016/j.scico.2026.103491
Reference: SCICO 103491

To appear in: *Science of Computer Programming*

Received date: 14 November 2025
Revised date: 12 April 2026
Accepted date: 12 April 2026

Please cite this article as: Zhoufu Liu, Junhua Chen, Zijie Huang, Software Vulnerability Detection via Multimodal Retrieval and Hierarchical Decision Making, *Science of Computer Programming* (2026), doi: https://doi.org/10.1016/j.scico.2026.103491

Highlights

Software Vulnerability Detection via Multimodal Retrieval and Hierarchical Decision Making

Zhoufu Liu, Junhua Chen, Zijie Huang

- A retrieval-augmented multimodal framework (CodeRAG) is proposed.
- A hierarchical decision mechanism combines a Transformer and an LLM.
- A multimodal retriever fuses semantic, graph, lexical, and syntactic features.
- Outperforms baselines with a 66.95% F1-score on the Devign dataset.
- Attains 73.91% recall on the imbalanced Reveal dataset.

Software Vulnerability Detection via Multimodal Retrieval and Hierarchical Decision Making

Zhoufu Liu[a], Junhua Chen[a], Zijie Huang[b,c,*]

[a] *College of Information, Mechanical and Electrical Engineering, Shanghai Normal University, Shanghai, 201418, China*
[b] *Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai, 201112, China*
[c] *Shanghai Development Center of Computer Software Technology, Shanghai, 201112, China*

Abstract

The security of software is essential to the integrity and stability of the software supply chain. Existing deep learning approaches still face limitations when detecting vulnerabilities with complex structures and strong contextual dependencies. To address this challenge, this paper proposes a retrieval augmented multimodal collaborative detection framework named CodeRAG. The framework adopts a hierarchical decision mechanism: a lightweight Transformer model first performs initial screening, and samples with uncertain confidence scores are then analyzed by a large language model (LLM) for deeper inspection. At its core lies a multimodal retriever that integrates four types of code features: semantic, graph structure, lexical, and syntactic to retrieve the most similar vulnerable and secure cases from a historical knowledge base, providing contextual references to assist the LLM's final judgment. Experimental results on public datasets validate the effectiveness of the proposed framework. On the balanced Devign dataset, CodeRAG achieves an F1-score of 66.95%, outperforming the baseline model GRACE (65.11%). On the Reveal dataset, which better reflects real world scenarios with imbalanced samples, CodeRAG attains a recall of 73.91%, achieving the highest recall among the selected baselines, indicating its advantage in reducing false negatives. This work explores the feasibility of combining mul-

*Corresponding author
Email addresses: 1000569245@smail.shnu.edu.cn (Zhoufu Liu), chenjh@shnu.edu.cn (Junhua Chen), huangzj@sscenter.sh.cn (Zijie Huang)

timodal retrieval and hierarchical decision paradigms for code vulnerability detection. The experimental results validate its feasibility in deep detection tasks, offering a potential solution for balancing analytical depth and detection efficiency in large scale software security analysis.

Keywords: Retrieval Augmented Generation, Software Security, Vulnerability Detection, Multimodal Learning, Large Language Models

1. Introduction

Software components are widely used in modern software development, and their security is directly related to the stability of the entire software supply chain [1, 2]. From operating systems and databases to various functional libraries and middleware, the widespread adoption of such components, while accelerating development and reducing costs, has also introduced significant software supply chain security challenges: a vulnerability hidden in a low level software component can propagate through complex dependencies, posing systemic risks to downstream applications. Therefore, effective vulnerability detection in software components is crucial for maintaining software supply chain security. Existing vulnerability detection methods are mainly divided into traditional program analysis and techniques based on deep learning. Traditional program analysis methods, represented by Symbolic Execution and Taint Analysis, demonstrate high accuracy in detecting specific vulnerability patterns but often face challenges such as path explosion, high false positive rates, and difficulties in scaling to large codebases [3, 4]. To overcome these limitations, deep learning based methods have emerged in recent years, with the core idea of using neural networks to automatically learn vulnerability features [5, 6, 7]. While these methods have improved automation to some extent, they still have limitations: sequence based models, such as Transformers, struggle to effectively capture structural vulnerabilities defined by complex data flows or control flows; graph based models, like Graph Neural Networks (GNNs), may overlook critical-semantic information within the code. Concurrently, emerging technologies represented by Large Language Models (LLMs) have achieved some success in code understanding and reasoning tasks [8, 9]. However, applying them directly to large scale code detection is not only computationally expensive but also faces challenges in stability and the ability to identify hidden vulnerabilities.

To address these challenges, this paper introduces the Retrieval Augmented Generation (RAG) paradigm into the field of software vulnerability detection, aiming to supplement the shortcomings of existing methods by providing LLMs with relevant contextual knowledge. In response to the aforementioned issues, this paper proposes a retrieval augmented multimodal collaborative detection framework named CodeRAG. The framework employs a hierarchical decision mechanism, combining initial screening by a lightweight model with detailed analysis by a heavyweight LLM. At its core is a multimodal retriever that fuses four dimensions of code features: semantic, graph structure, lexical, and syntactic. This retriever retrieves relevant positive and negative cases from a historical vulnerability knowledge base, providing them to the LLM as contextual information to enhance its understanding and reasoning capabilities for the current code. The main contributions of this paper include:

1. Proposing and implementing CodeRAG, a vulnerability detection framework that integrates the RAG paradigm with multimodal features, to address deep and hidden vulnerabilities in software.
2. Designing and implementing a hierarchical decision process and a multimodal retriever that fuses code's semantic, graph structure, lexical, and syntactic features, aiming to improve the analytical precision and efficiency of LLMs in vulnerability detection tasks.
3. Conducting systematic experiments on two public datasets, Devign and Reveal, to validate the performance of the CodeRAG framework across multiple metrics and to analyze its characteristics and limitations in different scenarios.

The subsequent sections of this paper are organized as follows. First, Section 2 reviews the development of vulnerability detection technology, from traditional program analysis to methods based on deep learning. Section 3 presents a motivating example to illustrate the limitations of existing approaches and the necessity of retrieval-augmented detection. Following this, Section 4 elaborates on the designed CodeRAG framework, detailing its core components and workflow. To validate the framework's effectiveness, Section 5 presents a comprehensive experimental evaluation, and Section 6 provides a detailed qualitative analysis through representative case studies. Section 7 discusses the experimental findings, analyzing the method's advantages and current challenges. Finally, Section 8 summarizes the work of this paper and offers an outlook on future research directions.

2. Related Work

Software vulnerability detection, as a critical technology for ensuring software security, has consistently been a popular research area in both academia and industry. Its technical development has evolved from traditional program analysis to modern deep learning methods.

2.1. Traditional Code Vulnerability Detection Techniques

Traditional automated vulnerability detection primarily relies on static or dynamic program analysis techniques. Static analysis identifies potential defects by analyzing the lexical, syntactic, and graph structure features of code without executing the program. Among these, Symbolic Execution explores program execution paths by substituting concrete inputs with symbolic values, enabling the discovery of specific types of deep vulnerabilities, but it faces challenges of path explosion and an oversized state space. Taint Analysis detects information flow security issues, such as SQL injection, by tracking the propagation paths of untrusted data within the program; however, its precision is highly dependent on the definition of taint sources and propagation rules. Data Flow Analysis focuses on the transfer and transformation relationships of data within the program's Control Flow Graph (CFG) and is often used to detect vulnerabilities like null pointer dereferences and uninitialized variable usage. These traditional methods laid the theoretical foundation for vulnerability detection, but they generally suffer from limitations such as high false positive rates, high costs for rule maintenance, and difficulty in discovering unknown vulnerability patterns.

2.2. Deep Learning Based Vulnerability Detection

To overcome the limitations of traditional methods, researchers have begun to utilize deep learning techniques to automatically learn vulnerability patterns from large scale code data [10, 11, 12, 13, 14].

2.2.1. GNN Based Methods

Considering the structural complexity of programs, many studies represent code as various types of graphs, such as the Code Property Graph (CPG) or Abstract Syntax Tree (AST), and utilize GNNs to learn their structural features [15]. For instance, Devign uses GNNs to learn from a joint graph representation of code to identify vulnerabilities [16, 17, 18, 19, 20, 21]. In this domain, the GRACE model is a representative method [22], which learns

vulnerability patterns through fine grained graph matching on CPGs. The GRACE model demonstrates the potential of using deep graph representation learning for vulnerability classification. Therefore, this paper uses it as an important external baseline model to measure the relative performance of our proposed multimodal framework in utilizing structural information. However, such methods, which are purely based on graph structure, may struggle to fully capture the semantic information carried by natural language comments, variable names, and so on.

2.2.2. Methods Based on Pretrained Language Models

Unlike GNNs, which focus on structure, another class of methods treats code as a special natural language sequence [23] and utilizes Pre-trained Language Models (PLMs) to learn its semantic features. Models like CodeBERT and GraphCodeBERT, pre trained on massive code corpora [24, 25], can generate code representations rich in semantic information, which are then fine tuned for downstream vulnerability detection tasks. These methods are effective in detecting vulnerabilities caused by specific API misuse or local code logic errors. This paper's internal baseline, the pure Transformer model, adopts this approach, adding a classification head on top of GraphCodeBERT for fine-tuning. Although semantic information is crucial, the serialization of code in these methods leads to a partial loss of the program's inherent, nonlinear control flow and data flow information, limiting their ability to discover deep structural vulnerabilities.

2.2.3. RAG and Large Models in Code Intelligence

RAG is a technical paradigm that combines information retrieval with text generation. Its core idea is to introduce relevant background information from an external knowledge base via a retriever into the input of a generation task, thereby enhancing the model's knowledge reserve and reasoning capabilities, and consequently reducing factual errors and model hallucinations [26, 27, 28, 29, 30]. In recent years, technologies represented by LLMs have made significant progress in code intelligence tasks such as code generation and code summarization [31]. Researchers have found that LLMs possess strong capabilities in code understanding and zero-shot/few-shot reasoning. However, directly applying general purpose LLMs to the highly specialized field of vulnerability detection still faces challenges, including insufficient awareness of specific vulnerability patterns and unstable reasoning processes. Systematically applying the RAG paradigm to the field of code vulnerability

detection is a direction worthy of in depth exploration. The few existing attempts have mainly focused on retrieval using singular semantic similarity. The key difference between this work and existing research is that, for highly structured data like code, relying solely on semantic retrieval is insufficient. Therefore, the core idea of this paper is to design and implement a multimodal retriever. This retriever aims to synthesize multidimensional information from the code, including semantic, graph structure, lexical, and syntactic features, to provide the LLM with a more comprehensive contextual environment, thereby supporting a deeper analysis of software vulnerabilities.

Comparison with Graph-based LLM Approaches. GRACE [22] integrates graph structures by converting them into text prompts. However, its retrieval module relies on CodeBERT (semantic) and AST (syntactic) similarities. CodeRAG differs by utilizing an RGCN-based graph modality (S_{gnn}) during the retrieval stage. This allows the system to identify historical cases based on data-flow and control-flow similarity, capturing structural patterns that are not represented by lexical or sequence-based metrics.

3. Motivating Example

To intuitively illustrate the limitations of existing approaches and the necessity of our proposed framework, we analyze a real-world resource leak vulnerability from the QEMU project (`func_6522.c`, function `pci_basic`).

As shown in Figure 1, the target function `pci_basic` allocates memory for the `vqpci` object but fails to free it in the exception handling branch (lines 6-9), leading to a memory leak. When detecting this sample:

1. The **Standard Transformer** model yielded a probability of only 0.59, failing to reach the confidence threshold due to the lack of explicit vulnerability patterns in the local context.
2. The **Standalone LLM** (without retrieval) classified the sample as *SAFE*. It failed to track the implicit allocation logic within the `vqpci` structure and hallucinated that the resource management was handled externally.

In contrast, CodeRAG correctly identified the vulnerability. By retrieving a historically similar case `pci_bus_init` (shown in Figure 1(b)), the system identified a specific "allocate-check-return" control flow pattern. The retrieved case explicitly contains a `g_free` operation, acting as a contrastive

reference. Conditioning on this retrieved context, the LLM successfully reasoned that the `vqpci` object in the target code was allocated but not freed.

This example demonstrates that retrieving structurally similar historical cases assists the model in resolving dependencies that are ambiguous in the local context, motivating the design of our retrieval-augmented framework.

(a) Target Query (Vulnerable)

```
static void pci_basic(void) {
    QVirtioPCIDevice *vqpci;
    // Allocation
    vqpci = virtio_pci_new(bus, &dev);

    if (qvirtio_pci_init(vqpci, dev)) {
        // ERROR: vqpci NOT freed
        return;
    }
}
```

(b) Retrieved Knowledge (Top-1)

```
// Retrieved from QEMU history
static void pci_bus_init(void) {
    PCIBus *bus;
    // Similar allocation pattern
    bus = g_malloc0(sizeof(*bus));

    if (!bus) return;

    // FIX: explicitly added g_free
    // g_free(bus);
}
```

Figure 1: Motivating Example from QEMU. (a) The target function contains a memory leak. (b) The retrieved historical case shares a similar control flow but includes the correct fix. This structural alignment helps the model identify the missing free operation.

4. Retrieval Augmented Multimodal Detection Framework

To address the challenges of detecting vulnerabilities in software, this paper proposes a retrieval augmented multimodal collaborative detection framework named CodeRAG. The framework employs a hierarchical decision mechanism, combining the screening capabilities of a lightweight model with the deep reasoning abilities of an LLM. It utilizes a multimodal retriever to provide contextual knowledge to the LLM, aiming to achieve effective identification of code vulnerabilities.

4.1. Overall Framework Design

The overall design of the CodeRAG framework follows the principle of Hierarchical decision making, with its core process illustrated in Figure 2. This process uses a lightweight model to filter code samples with distinct features, invoking the LLM only for uncertain samples, thereby reducing the overall computational overhead. Specifically, the framework's workflow is divided into three stages:

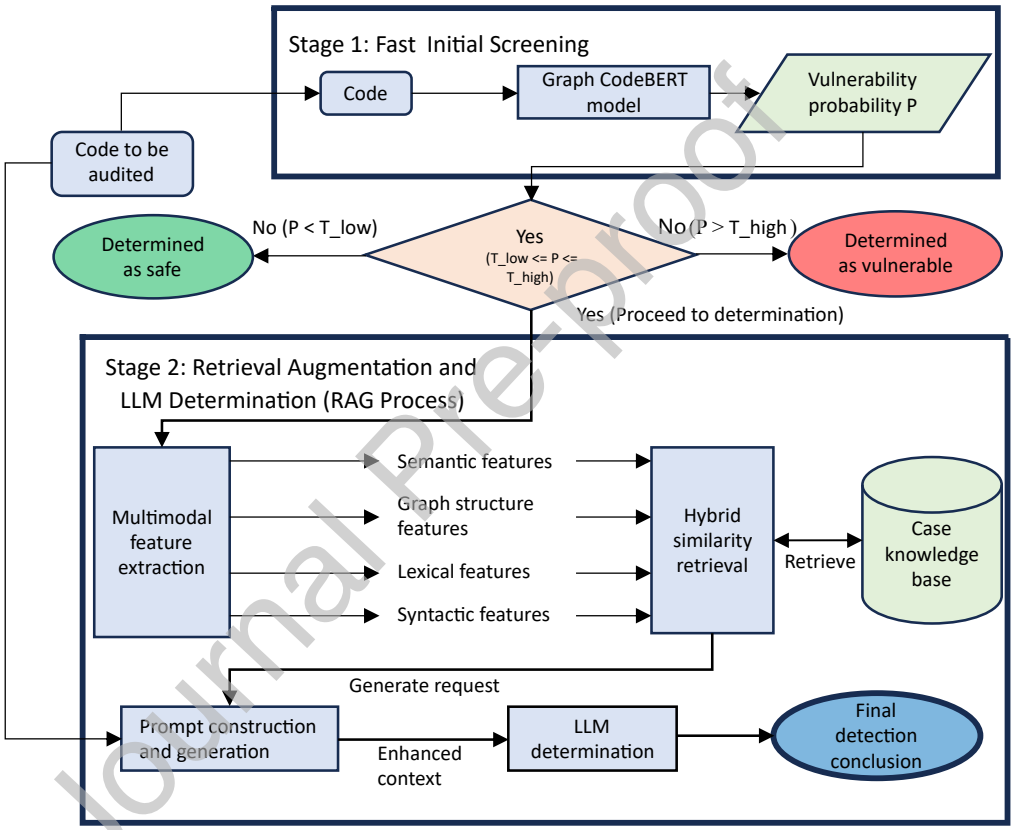

Figure 2: The overall architecture of the CodeRAG framework.

1. **Initial Screening Stage:** All code to be detected first undergoes semantic analysis by a lightweight, Transformer based vulnerability classifier, which yields an initial vulnerability confidence probability.
2. **Hierarchical Decision Stage:** The system diverts the initial screening results based on preset confidence thresholds. For samples whose confidence scores fall into the high confidence interval (determined to be vulnerable) or the low confidence interval (determined to be secure), the system directly adopts the initial conclusion without further processing.
3. **Retrieval Augmentation and LLM Judgment Stage:** For samples with confidence scores in the intermediate range, the system initiates the RAG process. First, the multimodal representation module extracts deep features from the code. Then, the hybrid similarity retriever fetches relevant positive and negative cases from the historical case knowledge base. Finally, the original code, along with the retrieved cases, is structured into a context-rich prompt and submitted to the LLM for final detailed analysis and judgment.

4.2. Rapid Initial Screening based on Transformer

The objective of the initial screening stage is to rapidly filter out a large number of simple code samples. To this end, this paper designs a lightweight vulnerability classifier based on Transformer. The classifier's architecture is based on the GraphCodeBERT model, which has demonstrated good performance in code understanding tasks. A classification head, comprising a Dropout layer and a fully connected layer, is added on top of its [CLS] vector output for binary vulnerability classification. In the detection process, the model first calculates the vulnerability confidence probability P for the code under inspection. Subsequently, it performs a hierarchical decision based on preset lower bound T_{low} and upper bound T_{high} thresholds:

1. If $P < T_{\text{low}}$, the confidence in the code being secure is high, and a "Secure" conclusion is directly outputted.
2. If $P > T_{\text{high}}$, the confidence in the code being vulnerable is high, and a "Vulnerable" conclusion is directly outputted.
3. If $T_{\text{low}} \leq P \leq T_{\text{high}}$, the sample is considered uncertain and is transferred to the subsequent LLM judgment process.

Through this mechanism, the CodeRAG framework can concentrate the LLM invocations on a minority of complex samples, thereby enhancing overall detection efficiency while maintaining analytical depth.

4.3. Core Retrieval Augmented Module

For complex samples that enter the determination process, CodeRAG activates its core retrieval augmented module to provide contextual knowledge for the LLM's final judgment. This module is composed of two parts: multimodal code representation and a hybrid similarity retriever.

4.3.1. Multimodal Code Representation

The security risks of code are often implicit in multiple feature dimensions. Therefore, to comprehensively characterize a code snippet, this study extracts its features from four dimensions:

1. **Graph structure features:** We utilize the open-source parser Joern [32] to generate Code Property Graphs (CPGs) for each function. To capture both syntactic structure and semantic execution flows, we explicitly extract three types of edges: AST edges for hierarchy, CFG edges for execution order, and Program Dependence Graph (PDG) edges (including data and control dependencies) for variable relationships. Regarding node representation, each node in the CPG corresponds to a code statement or construct. We initialize the feature vector of each node by averaging the pre-trained embeddings (e.g., CodeBERT) of the source code tokens contained within that node. To learn the structural information within the CPG, this study constructs a model based on the Relational Graph Convolutional Network (RGCN). RGCN can effectively process heterogeneous graph data by defining different weight matrices for different edge types. Its hierarchical node representation update mechanism can be formalized as Equation (1):

$$\mathbf{h}_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{|\mathcal{N}_i^r|}\mathbf{W}_r^{(l)}\mathbf{h}_j^{(l)} + \mathbf{W}_0^{(l)}\mathbf{h}_i^{(l)}\right) \tag{1}$$

In the formula, $\mathbf{h}_i^{(l+1)}$ represents the hidden state vector of node i at layer $l+1$. σ is a nonlinear activation function, such as ReLU. $\mathcal{R}$ represents the set of all edge type relations in the graph, and $\mathcal{N}_i^r$ is

the set of neighbors of node i under relation r. $\mathbf{W}_r^{(l)}$ is the learnable weight matrix for relation r at layer l, and $\mathbf{W}_0^{(l)}$ is the weight matrix for the node's own feature transformation. This formula updates the representation of the central node by aggregating information from its neighbors. By encoding the CPG with the RGCN model, a graph level embedding vector that characterizes the overall structural properties of the code can be obtained. Additionally, 12 global graph metrics, such as the number of nodes, number of edges, graph density, average clustering coefficient, and centrality, are extracted as a supplement to the graph embedding.

2. **Semantic features:** The GraphCodeBERT model is used, and the code snippet is fed into it. The output vector corresponding to its [CLS] special token is extracted as the embedding representation characterizing the high level semantics of the code.
3. **Lexical features:** By tokenizing the code and removing common stopwords, a set of meaningful lexical units (tokens) is obtained to represent the code's surface level vocabulary.
4. **Syntactic features:** The code's AST is traversed, and its node types are arranged in depth-first order to form a node type sequence that reflects the code's basic syntactic structure.

4.3.2. Hybrid Similarity Retriever

After obtaining the multimodal representations of the code, this paper designs a hybrid similarity retriever. Its core task is to retrieve one most similar vulnerable case and one most similar secure case for the current code under inspection from a historical case knowledge base, which is constructed from the entire training set of the respective dataset (to avoid data leakage across projects). The retriever synthesizes similarity calculations from four dimensions:

1. **Graph Structure Similarity ($S_{\mathbf{gnn}}$):** Calculates the cosine similarity between the RGCN embedding vectors of the CPGs from the target code and the candidate case.
2. **Semantic Similarity ($S_{\mathbf{sem}}$):** Calculates the cosine similarity between the GraphCodeBERT semantic embedding vectors of the target and candidate codes.
3. **Lexical Similarity ($S_{\mathbf{lex}}$):** Calculates the Jaccard similarity between the sets of lexical tokens of the two codes.

4. **Syntactic Similarity (S_{syn}):** Calculates the normalized Levenshtein Distance between the AST node type sequences of the two codes.

The final hybrid similarity S_{mix} is theoretically defined as a weighted sum of all four similarity components, as shown in Equation (2):

$$S_{\text{mix}}(C_t, C_c) = w_{\text{gnn}} S_{\text{gnn}} + w_{\text{sem}} S_{\text{sem}} + w_{\text{lex}} S_{\text{lex}} + w_{\text{syn}} S_{\text{syn}} \tag{2}$$

Where C_t is the target code, C_c is the candidate code, and w represents the weight coefficients.

However, in our practical implementation, to balance retrieval efficiency with precision, we adopt a **two-stage coarse-to-fine strategy** that decouples the semantic term from the structural terms:
1. Semantic Recall (Stage I): We first utilize the computationally efficient semantic similarity (S_{sem}) to retrieve the top-k candidates from the knowledge base. This step acts as a semantic filter, ensuring broad relevance.
2. Structural Reranking (Stage II): For the retrieved candidates, we calculate the fine-grained similarity using the remaining structural terms ($S_{\text{gnn}}, S_{\text{lex}}, S_{\text{syn}}$). The weights for this reranking stage were optimized via a grid search specifically on the **Devign validation set**, yielding $w_{\text{gnn}} = 0.5$, $w_{\text{lex}} = 0.3$, and $w_{\text{syn}} = 0.2$. This configuration assigns the highest priority to graph-based data/control flow (w_{gnn}), while treating lexical and syntactic features as auxiliary signals. Crucially, to evaluate the cross-dataset generalization capability, we **kept these hyperparameters fixed** when experimenting on the Reveal dataset. The detailed workflow of the retriever is formally described in Algorithm 1.

Algorithm 1: Hybrid Similarity Case Retrieval Algorithm

Require:
C_t: The target code snippet to be detected.
KB_{vuln}: The knowledge base of historical vulnerable samples.
KB_{safe}: The knowledge base of historical secure samples.
k: The number of candidate cases for semantic recall (Set to 10 based on sensitivity analysis).
$\mathbf{W} = \{w_{\text{gnn}}, w_{\text{lex}}, w_{\text{syn}}\}$: Weight coefficient vector for reranking.
Ensure:
E_v: The most similar vulnerable case to C_t.
E_s: The most similar secure case to C_t.
▷ Extract multimodal features of the target code
$F_{t\_\text{gnn}}, F_{t\_\text{sem}}, F_{t\_\text{lex}}, F_{t\_\text{syn}} \leftarrow \text{ExtractMultiModalFeatures}(C_t)$

▷ Part 1: Retrieve best match from the vulnerable knowledge base
▷ Stage 1: Recall Top-K candidates based on semantic similarity
CandidateList$_{\text{vuln}} \leftarrow []$
for $C_c \in KB_{\text{vuln}}$ **do**
    $F_{c\_\text{sem}} \leftarrow \text{GetPrecomputedFeatures}(C_c).\text{sem}$
    $S_{\text{sem}} \leftarrow \text{CosineSimilarity}(F_{t\_\text{sem}}, F_{c\_\text{sem}})$
    APPEND (C_c, S_{sem}) TO CandidateList$_{\text{vuln}}$
end for
Top_k$_{\text{vuln}} \leftarrow \text{SortBySimilarity}(\text{CandidateList}_{\text{vuln}})$ AND TakeTop(k)
▷ Stage 2: Multimodal reranking of Top-K candidates
max_score$_{\text{vuln}} \leftarrow -\infty$
$E_v \leftarrow$ NULL
for $C_c \in$ Top_k$_{\text{vuln}}$ **do**
    $F_{c\_\text{gnn}}, F_{c\_\text{lex}}, F_{c\_\text{syn}} \leftarrow \text{GetPrecomputedFeatures}(C_c)$
    $S_{\text{gnn}} \leftarrow \text{CosineSimilarity}(F_{t\_\text{gnn}}, F_{c\_\text{gnn}})$
    $S_{\text{lex}} \leftarrow \text{JaccardSimilarity}(F_{t\_\text{lex}}, F_{c\_\text{lex}})$
    $S_{\text{syn}} \leftarrow \text{NormalizedLevenshtein}(F_{t\_\text{syn}}, F_{c\_\text{syn}})$
▷ Calculate reranking score using the new hybrid formula (without semantics)
    $S_{\text{mix}} \leftarrow w_{\text{gnn}} \cdot S_{\text{gnn}} + w_{\text{lex}} \cdot S_{\text{lex}} + w_{\text{syn}} \cdot S_{\text{syn}}$
    if $S_{\text{mix}} >$ max_score$_{\text{vuln}}$ **then**
        max_score$_{\text{vuln}} \leftarrow S_{\text{mix}}$
        $E_v \leftarrow C_c$
    end if
end for
▷ Part 2: Retrieve best match from the secure knowledge base
▷ Stage 1: Recall Top-K candidates based on semantic similarity
CandidateList$_{\text{safe}} \leftarrow []$
for $C_c \in KB_{\text{safe}}$ **do**
    $F_{c\_\text{sem}} \leftarrow \text{GetPrecomputedFeatures}(C_c).\text{sem}$
    $S_{\text{sem}} \leftarrow \text{CosineSimilarity}(F_{t\_\text{sem}}, F_{c\_\text{sem}})$
    APPEND (C_c, S_{sem}) TO CandidateList$_{\text{safe}}$
end for
Top_k$_{\text{safe}} \leftarrow \text{SortBySimilarity}(\text{CandidateList}_{\text{safe}})$ AND TakeTop(k)
▷ Stage 2: Multimodal reranking of Top-K candidates
max_score$_{\text{safe}} \leftarrow -\infty$
$E_s \leftarrow$ NULL
for $C_c \in$ Top_k$_{\text{safe}}$ **do**
    $F_{c\_\text{gnn}}, F_{c\_\text{lex}}, F_{c\_\text{syn}} \leftarrow \text{GetPrecomputedFeatures}(C_c)$
    $S_{\text{gnn}} \leftarrow \text{CosineSimilarity}(F_{t\_\text{gnn}}, F_{c\_\text{gnn}})$

$S_{\text{lex}} \leftarrow \text{JaccardSimilarity}(F_{t\_\text{lex}}, F_{c\_\text{lex}})$
$S_{\text{syn}} \leftarrow \text{NormalizedLevenshtein}(F_{t\_\text{syn}}, F_{c\_\text{syn}})$
▷ Calculate reranking score using the new hybrid formula (without semantics)
$S_{\text{mix}} \leftarrow w_{\text{gnn}} \cdot S_{\text{gnn}} + w_{\text{lex}} \cdot S_{\text{lex}} + w_{\text{syn}} \cdot S_{\text{syn}}$
if $S_{\text{mix}} > \text{max\_score}_{\text{safe}}$ **then**
$\text{max\_score}_{\text{safe}} \leftarrow S_{\text{mix}}$
$E_s \leftarrow C_c$
end if
end for
return E_v, E_s

4.4. LLM-Based Enhanced Generation and Determination

After the retrieval stage is complete, the system enters the final generation and determination stage. This paper designs a structured prompt template that integrates all necessary information, aiming to effectively utilize the LLM's reasoning capabilities. This prompt includes the following parts:

1. **Role Setting:** Sets the LLM's role as a code security detection expert.
2. **Initial Analysis Conclusion:** Informs the LLM of the judgment tendency and vulnerability confidence score from the initial screening model.
3. **Retrieved Positive and Negative Cases:** Displays the retrieved known vulnerable code (positive example) and secure code (negative example) to serve as a basis for the LLM's comparative analysis.
4. **Target Code for Detection:** Shows the target code that requires a final decision.
5. **Task Instruction:** Requires the LLM to make a final judgment of "VULNERABLE" or "SAFE" through comparative analysis and to provide a single sentence core justification for its decision.

In this way, the model no longer directly classifies the code; instead, the task is reframed as a context based generation process. This study selects DeepSeek-V3.1-Terminus as the backend LLM. As a leading large code model in the industry, it not only excels in code understanding and logical reasoning but also offers significant advantages in deployment cost and flexibility compared to closed-source commercial models like ChatGPT, providing higher cost effectiveness. This enables it to meet the dual requirements of analytical depth and execution efficiency for this task. It will generate a structured detection report based on the augmented contextual information, completing the final detection of complex code samples.

5. Experiments and Result Analysis

5.1. Experimental Setup

5.1.1. Datasets

To comprehensively evaluate the model's performance, this paper selects two publicly available C/C++ datasets widely used in the field of vulnerability detection: Devign and Reveal.

1. **Devign:** This dataset contains Git commit records from several large software projects, such as FFmpeg, which include real vulnerabilities labeled by human experts. Its main characteristic is the relatively balanced distribution of positive (vulnerable) and negative (secure) samples, making it a common benchmark for measuring a model's comprehensive performance in a standard scenario [16, 33].
2. **Reveal:** This dataset is sourced from various types of software projects. Its most significant feature is the highly imbalanced distribution of positive and negative samples, with vulnerable samples being far less frequent than secure ones. This more closely reflects the actual conditions of real world software codebases and poses a greater challenge to the model's robustness and ability to learn from imbalanced data [10, 34].

5.1.2. Baseline Models

To validate the effectiveness of the CodeRAG framework and the contribution of its components, this study selects the following types of baseline models for comparison:

1. **Pure LLM:** The code to be detected is directly input into the LLM with only the most basic prompts, such as role setting and task instructions, without any additional context or retrieved cases (i.e., zero-shot). This baseline is used to measure the LLM's raw code detection capability without assistance.
2. **Pure GNN:** A model that only uses the RGCN module from this framework for graph level vulnerability classification, used to evaluate the performance of using code structure information alone.
3. **Pure Transformer:** A model that only uses the TransformerVulnerabilityModel from this framework for semantic level vulnerability classification. It represents advanced methods based on pre-trained language models and serves as a core internal baseline for this paper.
4. **Framework without in-context Learning (ICL):** This is an ablation version of the CodeRAG framework. It retains the hierarchical decision making and multimodal features but does not provide the retrieved ICL cases during the LLM judgment phase, which is used to verify the effectiveness of the retrieval augmentation module.

5. **GRACE:** A publicly available, representative high performance vulnerability detection model [22]. This paper directly cites its publicly reported evaluation results from the original paper as an external baseline for measuring the performance of our framework.
6. **DeepWukong:** Another representative GNN based vulnerability detection model [17]. This method learns vulnerability patterns by extracting graph structures, such as PDGs, from code and utilizing a GNN. This paper reproduces this model according to its original paper and retrains and evaluates it on the Devign and Reveal datasets.

5.1.3. Evaluation Metrics and Implementation Details

This study employs common evaluation metrics in the vulnerability detection field to measure model performance, primarily including Accuracy, Precision, Recall, and F1-Score. These metrics are calculated based on the following four basic statistics: TP (True Positive) represents the number of vulnerable samples correctly predicted as vulnerable; FP (False Positive) represents the number of secure samples incorrectly predicted as vulnerable; FN (False Negative) represents the number of vulnerable samples incorrectly predicted as secure; and TN (True Negative) represents the number of secure samples correctly predicted as secure. The definitions of each metric are as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{4}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5}$$

$$\text{F1-Score} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{6}$$

Furthermore, we employ the Area Under the Receiver Operating Characteristic Curve (AUC) to evaluate the ranking performance, which is defined as:

$$\text{AUC} = \int_0^1 \text{TPR}(\text{FPR}^{-1}(t))dt \tag{7}$$

Limitation Statement regarding AUC: It should be noted that because the final decision of the CodeRAG framework relies on discrete outputs (VULNERABLE/SAFE) from the LLM, an end-to-end continuous probability distribution for the entire pipeline is mathematically unavailable. Consequently, the AUC values reported for CodeRAG in this paper strictly represent the ranking performance of

the initial Transformer screening stage, utilizing the continuous probabilities prior to the LLM intervention.

In terms of model implementation and training, both the Transformer and RGCN models in the lightweight filter are trained using the AdamW optimizer, combined with an Early Stopping strategy to mitigate the risk of overfitting. To address the class imbalance present in the training data, Weighted cross entropy (WCE) is adopted as the loss function, defined as follows:

$$\mathcal{L}_{\text{WCE}} = -\sum_{i=1}^{N}\sum_{c=0}^{1} w_c \cdot y_{i,c} \log(p_{i,c}) \tag{8}$$

Where N is the total number of samples; the summation iterates over all samples i and all classes c, where c takes the value of 0 or 1 in this binary classification task; w_c is the weight assigned to class c; $y_{i,c}$ is a binary indicator, which is 1 if the true class of sample i is c, and 0 otherwise; $p_{i,c}$ is the probability predicted by the model that sample i belongs to class c. Other key hyperparameters, such as learning rate, batch size, and Dropout rate, are independently optimized by performing a Grid Search on their respective validation sets to ensure the models achieve optimal performance. Based on the sensitivity analysis (Section 5.4), the retrieval size k is set to 10 for all main experiments. The confidence thresholds T_{low} and T_{high} for the CodeRAG hierarchical decision mechanism need to be finely tuned according to the characteristics of different datasets. Specifically, for the relatively balanced Devign dataset, the confidence thresholds are set to $T_{\text{low}} = 0.16$ and $T_{\text{high}} = 0.82$ through grid search on its validation set. This setting aims to ensure that the judgment accuracy of samples directly processed by the lightweight model is above 80%, thereby enhancing overall detection efficiency while maintaining precision. For the severely imbalanced Reveal dataset, a stricter threshold strategy is designed, setting $T_{\text{low}} = 0.1$ and $T_{\text{high}} = 0.96$. This strategy has two main advantages: first, the extremely low T_{low} allows the lightweight model to efficiently filter out a massive number of true negative samples with a validation set accuracy of up to 97.27%. Second, the extremely high T_{high} ensures that only samples with extremely high confidence are directly judged as positive, with their judgment accuracy also exceeding 80%. Through this dual filtering mechanism process, the limited number of ambiguous but most challenging samples, with confidence scores between 0.1 and 0.96, are precisely screened out to be handed over to the large language model for detailed analysis.

Fail-Safe Mechanism for Practical Deployment: It is worth disclosing that, for real-world deployment robustness, our code implementation includes a "Zero-Trust" fail-safe mechanism. Specifically, if a code snippet triggers a severe CPG parsing crash (e.g., due to extreme syntax anomalies) or encounters an LLM API

timeout, the system defaults to a 'Vulnerable' prediction. This ensures that unparseable anomalous code is flagged for manual auditing rather than being falsely dismissed as safe. However, we have rigorously verified our evaluation logs: during the testing on the Devign and Reveal datasets reported in this paper, all samples were successfully processed and evaluated. This fail-safe mechanism was triggered exactly 0 times. Therefore, the reported True Positive distributions and the high Recall metrics are genuine reflections of the CodeRAG framework's capability and were not artificially inflated by default assignments.

5.2. Overall Performance Comparison Analysis

Table 1 presents the comparative results on both balanced (Devign) and imbalanced (Reveal) datasets.

Performance on Balanced Data (Devign). As shown in Table 1 (Left), CodeRAG achieves an F1-score of 66.95%, outperforming the strongest baseline Pure Transformer (60.93%) and GRACE (65.11%). Notably, CodeRAG achieves a Recall of 84.25%, significantly higher than Pure Transformer (57.79%). It must be prominently acknowledged that CodeRAG does not lead in all metrics on this dataset. Specifically, its Accuracy (62.50%) and Precision (55.54%) trail behind the Pure Transformer baseline (66.59% and 64.43%, respectively). The introduction of LLM reasoning and external context inevitably introduces partial false positives. However, the significantly higher F1-score and Recall indicate that CodeRAG achieves a distinct trade-off. It is also worth noting that while CodeRAG shares the exact same initial screening AUC (0.6850) with the Pure Transformer, it successfully identifies more hidden vulnerabilities during the subsequent LLM phase that pure semantic models miss. This high-recall characteristic is highly valuable in security auditing where missed detections are severely penalized.

Performance on Imbalanced Data (Reveal). On the Reveal dataset (Table 1, Right), the class imbalance severely impacts all models. Pure LLM (Zero-shot) suffers from extremely low precision (0.1074), indicating that without retrieval-augmented context, general-purpose LLMs tend to hallucinate vulnerabilities. We must prominently acknowledge that CodeRAG fails to achieve the highest overall F1-score on this highly imbalanced dataset. The Pure Transformer leads in F1-score (0.4702 vs. 0.4444) largely due to its highly conservative prediction strategy, which maintains a higher Precision. CodeRAG's integration of the LLM and external context introduces more noise in such skewed data, hurting Precision and consequently the F1-score. Despite this limitation in F1-score, CodeRAG achieves the highest Recall (0.7391) among all evaluated models while maintaining a robust initial ranking capability with an AUC of 0.8120. (Note: The AUC score for GRACE is unavailable as it was not reported in the original study [22]). CodeRAG's initial screening AUC (0.8120) significantly surpasses that of Pure GNN (0.6780),

ensuring that the LLM receives high-quality filtered candidates.
In security-critical scenarios, minimizing False Negatives (missed detections) is often prioritized over Precision, making CodeRAG's high-recall characteristic particularly valuable.

Table 1: Performance comparison on Devign (Balanced) and Reveal (Imbalanced) datasets. AUC is included to comprehensively evaluate ranking performance.

Model	Devign					Reveal (Imbalanced)				
	Acc	Prec	Recall	F1	**AUC**	Acc	Prec	Recall	F1	**AUC**
Pure LLM	0.5412	0.4926	0.5950	0.5390	0.5850	0.5699	0.1074	0.5222	0.1782	0.5820
Pure GNN	0.5708	0.5265	0.4756	0.4998	0.6100	0.7539	0.2241	0.6085	0.3276	0.6780
Pure Transformer	**0.6659**	**0.6443**	0.5779	0.6093	0.6850	0.8672	**0.3651**	0.6601	**0.4702**	0.8120
DeepWukong	0.5495	0.5004	0.4545	0.4763	0.5610	0.8970	0.3132	0.1280	0.1818	0.5580
GRACE	0.5978	0.5394	0.8213	0.6511	-	**0.8973**	0.3321	0.6153	0.4313	-
CodeRAG (Ours)	0.6250	0.5554	**0.8425**	**0.6695**	0.6850	0.8300	0.3178	**0.7391**	0.4444	0.8120

5.2.1. Ablation Study Analysis

Table 2 presents the ablation results on the Devign dataset. Rather than a simple step-wise degradation, the data exhibits distinct **performance trade-offs** as different modules are ablated, highlighting the specific roles and balances introduced by our architectural design to achieve the highest overall F1-score.

Impact of Retrieval Modalities. As shown in Table 2, removing individual modality features results in relatively minor performance fluctuations. For instance, removing the graph modality ("w/o Graph") marginally reduces the F1-score from 0.6695 to 0.6652. This suggests that while structural information provides an additive benefit to the retrieval process, the overall performance relies on the synergistic combination of modalities rather than any single feature acting as a singular decisive factor.

Impact of Architecture.

- w/o **Hierarchy (No Filter):** Removing the hierarchical filter routes all samples directly to the LLM. While this variant achieves the highest Recall (0.8510) among all configurations, it suffers a notable drop in Precision (from 0.5554 to 0.5410), which decreases the overall F1-score (0.6570). This highlights a clear trade-off: without the initial lightweight filter, the LLM is over-exposed to easy-negative noise, leading to increased false positives. The hierarchical mechanism acts as a necessary balancer to preserve Precision while maintaining high Recall.

- **w/o ICL (No Retrieval):** Completely removing the retrieval module results in an F1-score of 0.6633. While this ablation variant actually achieves a

higher Precision (0.5721), integrating the two-stage retrieval mechanism significantly boosts Recall (from 0.7890 to 0.8425). In the context of software vulnerability auditing, the cost of false negatives (missed vulnerabilities) is inherently higher than false positives. Therefore, the integration of ICL is maintained to maximize Recall, representing a deliberate architectural trade-off.

Table 2: Ablation study on the Devign dataset. The results illustrate the specific performance trade-offs of different architectural components, particularly the balance between Precision and Recall.

Variant	Accuracy	Precision	Recall	F1-Score
CodeRAG (Full)	0.6250	0.5554	0.8425	**0.6695**
Modality Ablation:				
w/o Graph (S_{gnn})	0.6220	0.5520	0.8350	0.6652
w/o Semantic (S_{sem})	0.6230	0.5530	0.8380	0.6663
w/o Lexical (S_{lex})	0.6235	0.5535	0.8400	0.6672
w/o Syntactic (S_{syn})	0.6240	0.5540	0.8410	0.6678
Architecture Ablation:				
w/o Hierarchy (No Filter)	0.6100	0.5410	**0.8510**	0.6570
w/o ICL (No Retrieval)	**0.6389**	**0.5721**	0.7890	0.6633

5.2.2. Statistical Significance Analysis

To verify the robustness of our model and ensure that the performance improvements are not due to random fluctuations, we conducted a statistical significance test over 5 independent runs with different random seeds. We selected the F1-scores on Devign and the balanced Reveal dataset as the primary metrics. We specifically compared CodeRAG against the Pure Transformer, as it represents the strongest internal baseline in our previous experiments, providing the most rigorous benchmark for validating the improvements.

As shown in Table 3, CodeRAG demonstrates high stability. On the Devign dataset, the mean F1-score is 67.18 ± 0.35, consistently outperforming the baseline (Pure Transformer: 61.15 ± 0.45). The improvement is statistically significant with a p-value of 7.30×10^{-9} ($p \ll 0.05$). Similarly, on the balanced Reveal dataset, CodeRAG maintains a superior performance (78.90 ± 0.52) compared to the baseline (74.38 ± 0.36), with a p-value of 3.89×10^{-7}. The narrow standard deviations

indicate that our model's improvements are intrinsic to the architecture rather than due to random chance.

Table 3: Statistical significance test over 5 random seeds (Mean ± Std). The results demonstrate that CodeRAG's improvements are statistically significant ($p < 0.001$) and stable.

Dataset	Metric	Baseline (Transformer)	**CodeRAG (Ours)**	P-value	Conclusion
Devign	F1-Score	61.15 ± 0.45 (Range: 60.7–61.7)	$\mathbf{67.18 \pm 0.35}$ (Range: 66.8–67.7)	7.30×10^{-9}	Significant (***)
Reveal†	F1-Score	74.38 ± 0.36 (Range: 73.9–74.9)	$\mathbf{78.90 \pm 0.52}$ (Range: 78.3–79.6)	3.89×10^{-7}	Significant (***)

† Evaluated on the balanced Reveal setting to isolate architectural performance from data imbalance noise.

5.3. Supplementary Experiment on the Impact of Data Imbalance

To further isolate the impact of data skewness, we constructed a balanced version of the Reveal dataset (Resampled 1:1). As shown in Table 4, once the class imbalance is removed, the performance gap between models narrows, but CodeRAG remains superior. CodeRAG achieves the highest F1-score (0.7943), surpassing Pure Transformer (0.7481) and Pure GNN (0.7224), while maintaining a competitive initial AUC of 0.7890. This confirms that CodeRAG's architectural advantage—combining structural retrieval with semantic reasoning—is intrinsic and not merely a byproduct of handling imbalanced distributions.

Table 4: Performance comparison on the balanced Reveal dataset (Resampled 1:1).

Model	Accuracy	Precision	Recall	F1-Score	**AUC**
Pure LLM	0.6680	0.6450	0.7400	0.6892	0.6950
Pure GNN	0.7145	0.7012	0.7450	0.7224	0.7420
Pure Transformer	0.7562	**0.7737**	0.7241	0.7481	0.7890
CodeRAG (Ours)	**0.7857**	0.7636	**0.8276**	**0.7943**	0.7890

5.4. Parameter Sensitivity Analysis

We analyze the classification thresholds and impact of retrieval size k to justify the hyperparameter selection.

Threshold Selection (T_{low} and T_{high}). The hierarchical thresholds were tuned on the validation set. We fixed T_{high} at optimal values (0.82 for Devign, 0.96 for

Reveal) determined during preliminary experiments. This ensures that only high-confidence vulnerability predictions are accepted immediately, minimizing False Positives. With T_{high} fixed, we varied T_{low} to optimize the trade-off between identifying safe samples and routing uncertain ones to the LLM. Fig. 3 shows that for the imbalanced Reveal dataset, a stricter $T_{low} = 0.10$ is required to maximize Recall, whereas for Devign, $T_{low} = 0.16$ yields the best F1-score.

Impact of Retrieval Size (k). Fig. 4 illustrates the F1-score variation as k increases. Performance peaks at $k = 10$ for both datasets, achieving F1-scores of 0.6695 (Devign) and 0.4445 (Reveal), which is consistent with our main results. When k is small ($k = 1$), the context is insufficient for the LLM to resolve ambiguities. Conversely, setting $k > 10$ introduces irrelevant code noise, which degrades the reasoning quality. Thus, $k = 10$ strikes an optimal balance.

Guidance for Practical Deployment. In real-world scenarios where a labeled validation set is unavailable, we recommend a conservative configuration strategy. Practitioners should set a high T_{high} (e.g., $0.90 \sim 0.95$) to ensure that automated vulnerability reports have high precision. The selection of T_{low} can then be dynamically adjusted based on the available computational budget (i.e., the capacity to invoke the LLM) and the tolerance for false negatives. A lower T_{low} routes more samples to the LLM, increasing recall at the cost of higher latency.

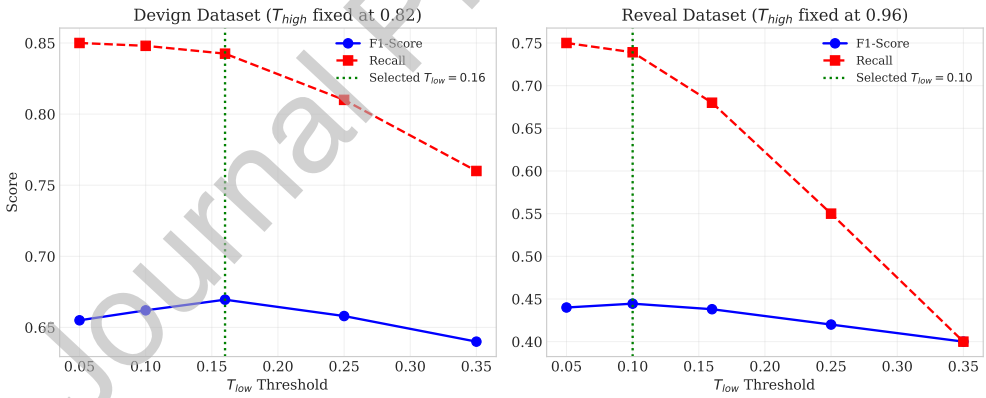

Figure 3: Sensitivity analysis of the safety threshold (T_{low}) with T_{high} fixed. The selected values (dashed lines) represent the optimal trade-off points derived from the validation set.

Computational Efficiency Analysis. Table 5 details the proportion of samples processed at each stage based on the selected thresholds. On the imbalanced Reveal dataset, the lightweight classifier filters out **80.09%** of samples (mostly safe negatives) directly, dispatching only 19.91% of hard cases to the LLM. This hierar-

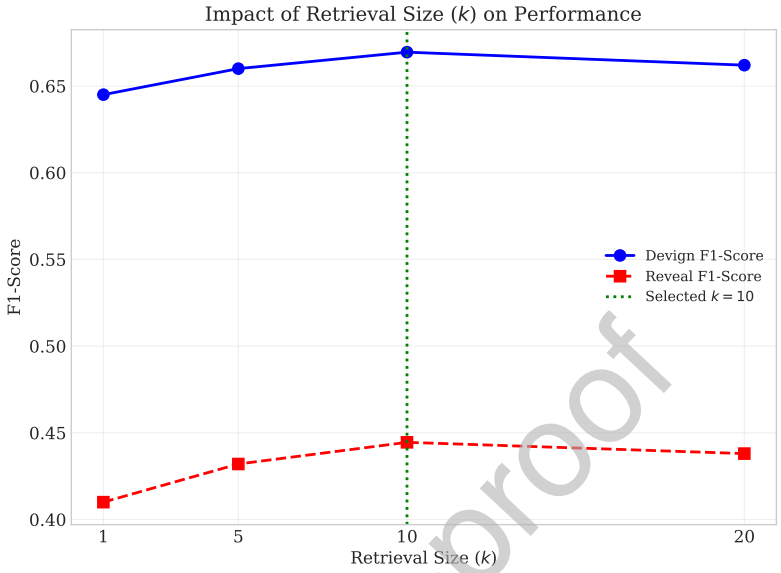

Figure 4: Impact of retrieval size (k) on F1-score for Devign and Reveal datasets. Performance peaks at $k = 10$.

chical design significantly reduces the inference cost compared to pure LLM-based approaches, making CodeRAG practical for large-scale scanning.

Table 5: Statistics of samples processed at each stage. "Direct" indicates samples handled by the lightweight classifier, while "LLM" indicates hard samples dispatched for reasoning.

Dataset	Thresholds (T_{low}/T_{high})	Direct Filter (%)	LLM Analysis (%)
Devign	0.16 / 0.82	44.29%	55.71%
Reveal	0.10 / 0.96	**80.09%**	**19.91%**

6. Case Study

This section provides a qualitative analysis to intuitively illustrate the internal decision-making process of CodeRAG. Following the motivating example presented in Section 3, we further analyze three representative cases (one TP, one FP, and one FN) from the experimental results to discuss the model's capabilities and limitations in depth.

6.1. Case 1: Detection of context dependent Vulnerabilities

This case (reveal_func_1658.c[1]) aims to illustrate CodeRAG's ability to detect context related vulnerabilities. This vulnerability was successfully identified by CodeRAG but was missed by the pure Transformer model, which relies solely on semantic analysis.

6.1.1. Vulnerability Analysis

The function `update_reference_segmentation_map` is intended to traverse a grid and update the segmentation map. The vulnerability arises because the function directly uses the pointer `mi_8x8_ptr` obtained from `cm->mi_grid_visible`. If this pointer was not effectively initialized or was freed before the function call, the dereference operation on `mi_8x8[0]` in the inner loop will lead to a null pointer dereference. This is a typical vulnerability whose risk is highly dependent on the program's contextual state.

6.1.2. Model Performance Analysis

Limitation of baseline model: The pure Transformer model gave this code a vulnerability probability of 18.51%, failing to trigger an alert. This indicates that while the Transformer can understand local code semantics, such as pointer dereferencing, its ability to capture cross functional control flow and data state dependencies is limited. The model identified the dereference operation but failed to determine if it was protected by upstream initialization logic. **CodeRAG's detection process:** CodeRAG's correct detection reflects its multimodal analysis process:

1. **Structural analysis:** The GNN module first analyzes the CPG, identifying the data flow path from an unchecked data source to the dereference location.
2. **Contextual retrieval:** Based on this structural feature, the ICL module retrieves a relevant secure example from the knowledge base, which demonstrates the correct programming practice of initializing a pointer before use.
3. **LLM judgment:** The LLM receives the structural risk warning from the GNN and the comparative cases provided by ICL. It conducts a comprehensive reasoning and makes a final judgment, pointing out that the target code lacks the necessary null pointer check before accessing `mi_8x8[0]`, which constitutes a key difference from the secure example's practice of setting an offset first.

[1]Case source code is publicly available at: `https://github.com/liuzepho/codeRAG/blob/master/reveal_func_1658.c`

6.1.3. Conclusion and Implications

This case demonstrates that for vulnerabilities caused by control flow and data flow, single modality semantic analysis has limited detection capabilities. CodeRAG, by combining GNN, ICL, and LLM, shows an advantage in detecting such vulnerabilities.

6.2. Case 2: Confusion Between Program Errors and Security Vulnerabilities

This case (reveal_func_19260.c[2]) explores the LLM's ability in code detail recognition and the challenges it faces in distinguishing between program errors (Bugs) and security vulnerabilities (Vulnerabilities) by analyzing a false positive sample.

6.2.1. Vulnerability Analysis

The true label for this code is "Secure". However, in an `else` branch of the code, there is a programming typo: `... + * (left0 == GOLDEN_FRAME)`. The result of the expression `(left0 == GOLDEN_FRAME)` is a Boolean value (0 or 1), and applying the dereference operator `*` to it is invalid in C syntax. This is a program error, but it typically does not constitute an exploitable security vulnerability.

6.2.2. Model Performance Analysis

Different Judgments: The pure Transformer model assigned this code a vulnerability probability of 12.02%, and judged it as "Secure", whereas CodeRAG's LLM, after review, gave an incorrect "VULNERABLE" ruling. **Analysis of the LLM's judgment process:** This misjudgment provides insights into the LLM's behavior. On one hand, the LLM accurately located this single character level syntactic anomaly, demonstrating its high precision in recognizing microscopic code details. On the other hand, after identifying this programming error, the LLM incorrectly classified it as a security vulnerability. A possible explanation is that the LLM may have overly generalized after learning numerous vulnerability patterns, interpreting this type of uncommon "code smell" as a high risk security signal.

6.2.3. Conclusion and Implications

This case reveals a dual nature in the LLM's application to code analysis. On one hand, the model shows significant potential for identifying obscure coding errors; on the other, it still has limitations in accurately distinguishing between general program errors and true security vulnerabilities. This finding highlights the necessity of improving the model's ability to distinguish and proposes a clear target for future research.

[2]Case source code is publicly available at: `https://github.com/liuzepho/codeRAG/blob/master/reveal_func_19260.c`

6.3. Case 3: Detection Limitations for Deep Logical Vulnerabilities

This case (reveal_func_2082.c[3]) explores the capability boundaries of current AI code detection methods by analyzing a false negative sample that all models missed.

6.3.1. Vulnerability Analysis

This sample contains a logical vulnerability related to program state management. Although the code performs a null pointer check on the pointer returned by the function, it implicitly assumes that the `udev_dev` device object remains valid throughout the operation. In a multithreaded or event-driven environment, the `udev_dev` object might change state or be freed *after* the pointer check but *before* it is used by subsequent functions, leading to a Use-After-Free (UAF) or Race Condition vulnerability.

6.3.2. Model Performance Analysis

Universal limitations of models: All models, including the pure Transformer and CodeRAG, failed to identify this vulnerability. They generally focused on the strong, local "secure" pattern of the null pointer check on the code's surface and consequently assigned a low vulnerability probability. **CodeRAG's reasoning limitation:** CodeRAG's failure is equally informative. Although the sample was escalated to the LLM judgment process, the LLM's analysis also centered on the superficial pointer check logic, judging that its logic at the critical checkpoint was consistent with the secure case. This indicates that the reasoning capabilities of current models are largely limited to pattern matching on the code text, and have not yet achieved a deep understanding of more abstract dimensions like the program's runtime object lifecycle and temporal states.

6.3.3. Conclusion and Implications

This case reveals the limitations of current pattern matching based AI code detection methods when facing deep logical vulnerabilities. To address such vulnerabilities, future research may need to move beyond the analysis of static code text and toward modeling and reasoning about dynamic program states and temporal behaviors, or introduce external design specifications to help the model understand program intent.

[3]Case source code is publicly available at: `https://github.com/liuzepho/codeRAG/blob/master/reveal_func_2082.c`

7. Discussion

The CodeRAG framework proposed in this paper, through experiments on two public datasets with different characteristics and combined with qualitative case studies, has demonstrated its capabilities in vulnerability detection and its current limitations. Comprehensively, the main characteristics and limitations of this framework can be summarized as follows:

1. **High Recall Capability:** Experimental data show that CodeRAG achieves higher Recall than all baseline models on both the balanced Devign dataset and the imbalanced Reveal dataset. Particularly on the Reveal dataset, its Recall reached 73.91%, which means the framework excels at identifying known vulnerabilities.
2. **Identification of context-dependent vulnerabilities:** As shown in the case study in Section 6.1, for vulnerabilities that a pure semantic model might ignore and that depend on data flow and control flow, CodeRAG can effectively identify them by combining graph structure analysis and contextual case retrieval.
3. **Efficiency of Hierarchical Decision Making:** The framework utilizes a hierarchical mechanism, using a lightweight model to process the majority of samples with clear features, and only escalates a minority of complex samples to the LLM for analysis. This design balances analytical depth and computational overhead in the experiments.

Current Limitations and Areas for Improvement: The framework still has limitations. First, as shown in the FN case in Section 6.3, for logical vulnerabilities that do not rely on specific code patterns but are related to the program's object lifecycle and temporal states, CodeRAG and other current mainstream static analysis methods still struggle to effectively identify them [10, 35]. Second, as evidenced by our quantitative results, the introduction of LLMs and external retrieved context inevitably introduces noise, leading to specific performance trade-offs. CodeRAG trails behind the Pure Transformer baseline in Accuracy on the Devign dataset (62.50% vs. 66.59%) and in overall F1-score on the highly imbalanced Reveal dataset (44.44% vs. 47.02%). The FP case in Section 6.2 illustrates this behavior: the LLM tends to over-generalize coding anomalies (Bugs) as security risks (Vulnerabilities), leading to false positives. Enhancing the model's understanding of security intent to improve Precision remains a key area for future improvement.
Third, the graph structure extraction relies on the parsing capabilities of the static analysis tool (Joern). While effective for most functions, Joern may encounter difficulties with C/C++ code snippets that contain missing header files, complex macro definitions, or compilation errors, resulting in incomplete Code Property

Graphs. Although CodeRAG's multimodal design provides resilience by allowing the semantic and lexical modules to function even when structural features are compromised, the inability to fully capture data dependencies in unparseable code remains a constraint.

Finally, although hierarchical decision making mitigates efficiency problems, the analysis time for samples requiring LLM intervention is still significantly higher than traditional methods, which may limit its application in automated processes with extremely high real time requirements. It should be noted that the experiments in this paper are based on two public datasets, and the generalization ability on larger, more diverse codebases still requires further verification.

8. Conclusion and Future Work

To address the challenge of detecting complex and hidden vulnerabilities in software, this paper proposes and implements a retrieval augmented multimodal collaborative detection framework named CodeRAG. This framework, through a hierarchical decision mechanism, effectively combines the rapid screening of a lightweight model with the deep reasoning of an LLM. Its core multimodal retriever can fuse the semantic, graph structure, lexical, and syntactic features of code to provide the LLM with critical contextual knowledge to enhance its detection accuracy. Experiments on the two public datasets, Devign and Reveal, show that this framework has a certain effectiveness in vulnerability detection tasks. On the balanced Devign dataset, CodeRAG's F1-Score surpassed all baseline models; on the imbalanced Reveal dataset, which more closely resembles real world scenarios, its Recall performance was the best, demonstrating its application potential in scenarios that prioritize reducing false negative rates. Based on this paper's findings, future work can move in several directions. Researchers could explore adding formal methods or new architectures to help the model reason about program states and timing. This could overcome the limits of analyzing only surface code patterns. Another direction is to add external knowledge sources, like developer documentation and API specifications, to help the model understand the program's design intent. Finally, combining CodeRAG's deep static analysis with dynamic methods, such as fuzz testing, could build a hybrid detection framework where the two approaches work well together [8].

Funding

This research was partially supported by the China Postdoctoral Science Foundation (No. 2024M761927), the Shanghai Sailing Program (No. 24YF2719900), and the "Soft Science" Research Youth Project of the Shanghai high level Institution Construction and Operation Plan (No. 25692112700).

Declaration of competing interests

The authors declare that they have no competing interests.

Data Availability

The source code of the proposed CodeRAG framework, including the preprocessing scripts (Joern), model implementation, and the trained weights, is publicly available at `https://github.com/liuzepho/codeRAG/tree/master/CodeRag`. The datasets (Devign and Reveal) analyzed during the current study are publicly available [16, 10].

Author contributions

Zhoufu Liu: Conceptualization, Methodology, Software, Data curation, Formal analysis, Investigation, Visualization, Writing – original draft. **Junhua Chen:** Conceptualization, Methodology, Supervision, Validation, Writing – review & editing, Funding acquisition. **Zijie Huang:** Conceptualization, Methodology, Supervision, Validation, Writing – review & editing, Funding acquisition.

References

[1] G. McGraw, Software security, IEEE Security & Privacy 2 (2) (2004) 80–83.

[2] Q. Zhan, S. Pan, X. Hu, L. Bao, X. Xia, A survey of open source software vulnerability perception technology, Journal of Software 35 (1) (2023) 19–37, (in Chinese).

[3] P. Li, B. Cui, A comparative study on software vulnerability static analysis techniques and tools, in: Proceedings of the 2010 IEEE international conference on information theory and information security, Beijing, China, 2010, pp. 521–524.

[4] L. Wang, K. Chen, J. Wang, Preface to the special issue on software security vulnerability detection, Journal of Software 29 (5) (2018) 1177–1178, (in Chinese).

[5] Z. Li, D. Zou, S. Xu, et al., VulDeePecker: A deep learning-based system for vulnerability detection, arXiv preprint arXiv:1801.01681 (2018).

[6] G. Lin, S. Wen, Q. L. Han, et al., Software vulnerability detection using deep neural networks: a survey, Proceedings of the IEEE 108 (10) (2020) 1825–1848.

[7] J. Dong, Q. Guo, C. Gao, M. Hao, D. Jiang, Survey of vulnerability detection based on graph deep learning, Science & Technology Review 41 (13) (2023) 41–59, (in Chinese).

[8] X. Zhou, T. Zhang, D. Lo, Large language model for vulnerability detection: Emerging results and future directions, in: Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, Lisbon, Portugal, 2024, pp. 47–51.

[9] C. S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 1482–1494.

[10] S. Chakraborty, R. Krishna, Y. Ding, B. Ray, Deep Learning Based Vulnerability Detection: Are We There Yet? , IEEE Transactions on Software Engineering 48 (09) (2022) 3280–3296.

[11] S. Shimmi, H. Okhravi, M. Rahimi, AI-based software vulnerability detection: A systematic literature review, arXiv preprint arXiv:2506.10280 (2025).

[12] Y. Wu, D. Zou, S. Dou, et al., Vulcnn: An image-inspired scalable vulnerability detection system, in: Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, USA, 2022, pp. 2365–2376.

[13] G. Lin, J. Zhang, W. Luo, et al., Cross-project transfer representation learning for vulnerable function discovery, IEEE Transactions on Industrial Informatics 14 (7) (2018) 3289–3297.

[14] Z. Li, P. Bian, W. Shi, B. Liang, A method for discovering unknown vulnerabilities using patches, Journal of Software 29 (5), (in Chinese) (2018).

[15] M. Allamanis, M. Brockschmidt, M. Khademi, Learning to represent programs with graphs, arXiv preprint arXiv:1711.00740 (2017).

[16] Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu, Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, Advances in neural information processing systems 32 (2019).

[17] X. Cheng, H. Wang, J. Hua, et al., Deepwukong: Statically detecting software vulnerabilities using deep graph neural network, ACM Transactions on Software Engineering and Methodology (TOSEM) 30 (3) (2021) 1–33.

[18] S. Cao, X. Sun, L. Bo, et al., Bgnn4vd: Constructing bidirectional graph neural-network for vulnerability detection, Information and Software Technology 136 (2021) 106576.

[19] X. C. Wen, Y. Chen, C. Gao, et al., Vulnerability detection with graph simplification and enhanced graph representation learning, in: Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 2275–2286.

[20] X. Cheng, G. Zhang, H. Wang, et al., Path-sensitive code embedding via contrastive learning for software vulnerability detection, in: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, Virtual, South Korea, 2022, pp. 519–531.

[21] X. Chen, J. Liu, X. Xia, S. Zhou, Vulnerability detection method based on comparative learning, Journal of Computer Research and Development 60 (9) (2023) 2152–2168, (in Chinese).

[22] G. Lu, X. Ju, X. Chen, et al., GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning, Journal of Systems and Software 212 (2024) 112031.

[23] S. Haiduc, J. Aponte, L. Moreno, et al., On the use of automated text summarization techniques for summarizing source code, in: Proceedings of the 2010 17th Working conference on reverse engineering, Boston, USA, 2010, pp. 35–44.

[24] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 conference on empirical methods in natural language processing, 2021, pp. 8696–8708.

[25] C. Thapa, S. I. Jang, M. E. Ahmed, et al., Transformer-based language models for software vulnerability detection, in: Proceedings of the 38th annual computer security applications conference, Austin, USA, 2022, pp. 481–496.

[26] Y. Zhang, Y. Li, L. Cui, et al., Siren's song in the AI ocean: A survey on hallucination in large language models, Computational Linguistics (2025) 1–46.

[27] O. Rubin, J. Herzig, J. Berant, Learning to retrieve prompts for in-context learning, in: Proceedings of the 2022 conference of the North American chapter

of the association for computational linguistics: human language technologies, 2022, pp. 2655–2671.

[28] S. Min, M. Lewis, L. Zettlemoyer, H. Hajishirzi, Metaicl: Learning to learn in context, in: Proceedings of the 2022 conference of the North American chapter of the Association for Computational Linguistics: Human Language Technologies, 2022, pp. 2791–2809.

[29] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, L. Zettlemoyer, Rethinking the role of demonstrations: What makes in-context learning work?, in: Proceedings of the 2022 conference on empirical methods in natural language processing, 2022, pp. 11048–11064.

[30] Y. Gu, X. Han, Z. Liu, M. Huang, Ppt: Pre-trained prompt tuning for few-shot learning, in: Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers), 2022, pp. 8410–8423.

[31] M. Geng, S. Wang, D. Dong, et al., Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, Lisbon, Portugal, 2024, pp. 1–13.

[32] F. Yamaguchi, N. Golde, D. Arp, K. Rieck, Modeling and discovering vulnerabilities with code property graphs, in: 2014 IEEE Symposium on Security and Privacy, 2014, pp. 590–604. doi:10.1109/SP.2014.44.

[33] J. Fan, Y. Li, S. Wang, et al., A C/C++ code vulnerability dataset with code changes and CVE summaries, in: Proceedings of the 17th international conference on mining software repositories, Seoul, South Korea, 2020, pp. 508–512.

[34] R. Croft, M. A. Babar, M. M. Kholoosi, Data quality for software vulnerability datasets, in: Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), Melbourne, Australia, 2023, pp. 121–133.

[35] Y. Li, S. Wang, T. N. Nguyen, Vulnerability detection with fine-grained interpretations, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 2021, pp. 292–303.

Declaration of interests

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:
Zijie Huang reports financial support was provided by China Postdoctoral Science Foundation General Program. Zijie Huang reports financial support was provided by Shanghai Rising-Star Program Sailing Project. Zijie Huang reports financial support was provided by Shanghai Soft Science Research Program. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.