Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.075938

ARTICLE

Research on Prompt Engineering to Enhance LLM-Driven CPG Vulnerability Reachability

Xiaorong Feng[1,2], **Ying Gao**[1,*], **Pengyi Du**[2] **and Leyu Shi**[1]

[1]School of Computer Science and Engineering, South China University of Technology, Guangzhou, China
[2]China Electronic Product Reliability and Environmental Testing Research Institute, Guangzhou, China

*Corresponding Author: Ying Gao. Email: gaoying@scut.edu.cn

Received: 11 November 2025; Accepted: 01 April 2026; Published: 15 June 2026

ABSTRACT: In recent years, large language models (LLMs) have seen growing application in code understanding and security analysis. However, their performance relies heavily on prompt context quality and engineering design, with unstable vulnerability detection and high false positive rates remaining key bottlenecks to reliable adoption. This paper systematically reviews advances in prompt engineering and context optimization across four core areas and proposes LARA (LLM-Augmented Reachability Analysis), a neural-symbolic framework leveraging code property graphs (CPGs), which uses a static analysis engine to extract source-to-sink data flow paths, integrates systematic prompt engineering to create context-aware prompts, and invokes LLMs for path risk scoring and reasoning. The framework forms a closed-loop process of path identification, risk assessment, and manual verification. Experimental validation on the Log4Shell vulnerability shows LARA accurately identifies core exploit paths and outperforms traditional static signature methods in covering unknown/variant vulnerabilities without explicit dangerous functions, with notable improvements in contextual understanding and detection accuracy. Our study identifies three core research trends: systematic context engineering, task-adaptive prompt strategies, and prompt-analysis closed loops in code security. However, LLMs still face challenges like long-text generation logic breaks, cross-model prompt transfer degradation, and code security false positive control. Future work should focus on long-context understanding/generation co-optimization, code-security-oriented fine-tuning of open-source LLMs, and deep CPG-prompt engineering integration to advance LLMs' practical use in code security.

KEYWORDS: Large language models; prompt engineering; context optimization; code security analysis; neural-symbolic systems; code property graphs

1 Introduction

With the proposal of strategic concepts oriented toward cross-domain technical integration, Large Language Models (LLMs) are anticipated to undertake advanced code security tasks, including attack path generation and vulnerability reachability analysis. However, early research shows that simply applying LLMs for vulnerability detection yields unstable performance, and may even be no better than random guessing [1]. The core issue lies in the quality of LLM reasoning, which is closely related to the completeness and accuracy of the prompt context. In scenarios of context missing, the model tends to make reasoning errors (such as misjudging the validity of patches), leading to the perception that LLMs are unreliable. Such challenges in automated security analysis are not unique to code vulnerability detection; similar issues arise in IoT network security, where machine learning models are employed for tasks like DoS attack detection, underscoring the broader need for robust and context-aware intelligent systems in cybersecurity [2]. Despite

these challenges, research continues to advance the application of LLMs in security contexts. Recent progress has extended such approaches to crash analysis—for example, in 2025, Herter et al. introduced GPTrace [3], a framework that leverages LLM embeddings to deduplicate crash reports by clustering semantically similar stack traces, significantly reducing manual triage efforts. To address this issue, current research shows two major trends: one is the refinement of prompt engineering, where systematic context optimization solutions are constructed from various aspects including structured output control, long-context noise filtering, and quantitative evaluation of prompt strategies; the second is the integration of technological paradigms, which combines symbolic analysis (such as deterministic data flow extraction using code property graphs (CPG)) with neural reasoning (LLM semantic understanding), forming a neural-symbolic system of data facts and intelligent judgment.

Neuro-symbolic systems refer to an intelligent framework that combines the learning capabilities of neural networks with the reasoning abilities of symbolic systems, thereby possessing advantages in perception and cognition. This concept originated from the exploration of the integration of deep learning and symbolic logic in the field of artificial intelligence and has become an emerging research hots-pot in recent years. For code security analysis, the "neuro-symbolic" approach offers a new idea: it leverages the powerful semantic understanding capabilities of LLM while relying on symbolic execution/static analysis to provide deterministic logical constraints. For example, the IRIS framework proposed by Li et al. [4] is referred to as a "neuro-symbolic" method that systematically combines LLM with static program analysis for vulnerability detection across the entire code-base.

Context engineering is a new concept that has emerged in recent years with the application of large models. It is defined as a technological system that goes beyond simple prompt design and systematically optimizes the context for LLM reasoning. The review by Steenhoek et al. [5] provides a systematic explication for the first time, dividing context engineering into four fundamental components (context retrieval, generation, processing, management) and four types of integrated architectures (Retrieval-Augmented Generation RAG, long-term memory, tool invocation, multi-agent collaboration). This system outlines various strategies for providing high-quality context information to LLMs and reveals key bottlenecks. Currently, LLMs exhibit a remarkable understanding of complex long contexts, but they perform poorly in generating equally complex long text outputs. This asymmetry issue is an important challenge in the field of context engineering. To mitigate such generation constraints, Shrestha and Mahmoud recently applied a 'Chain-of-Density' prompting strategy to security reviews [6], iteratively densifying the context to ensure that critical vulnerability entities are retained without exceeding token limits. LSTM-based models proposed by Li et al. [4], and Transformer-based models with a fine-tuning method proposed by Fu & Tantithamthavorn, Steenhoek et al. [5], Cheng et al. [7]. These approaches focus on method-level detection of vulnerabilities, and the test result is just a binary label classifying a method as vulnerable or not.

The CPG, proposed by Yamaguchi et al. in 2014, is a code representation method that integrates various classical program analysis concepts [8]. It unifies the AST, CFG, and PDG into a single data structure, comprehensively representing the code's structure, control flow, and data dependencies. This integrated view allows researchers to define vulnerability patterns using graph traversal, elegantly uncovering common vulnerabilities such as buffer overflows and integer overflows. The CPG has been shown to be effective in large-scale vulnerability discovery: this method initially uncovered 18 previously unknown vulnerabilities in Linux kernel code, demonstrating its powerful capabilities [9]. Since then, the CPG concept has seen widespread development, including the emergence of the open-source Joern tool for industrial-grade vulnerability scanning.

This article systematically reviews the progress in the aforementioned fields based on the latest literature, analyzes the research challenges, and proposes the LARA framework, providing theoretical and practical references for the in-depth application of LLMs in the field of code security analysis.

2 Code Security Analysis Based on LLM

Recent research indicates that using large models for code vulnerability detection is becoming a hot topic. For example, NDSS 2025 systematically compared the capabilities of various advanced LLMs (including LLAMA2, CodeLLAMA, Mistral, Gemma, GPT-4, etc.) in vulnerability detection [10], showing significant performance differences among models. Some new models (like Gemma and LLAMA-2) performed excellently in identifying Java vulnerabilities, while models in certain configurations nearly degraded to random guessing levels. The accuracy of models fluctuated significantly under different programming languages and sample settings; generally, a consistent increase in context window size tends to enhance detection effectiveness, while other factors like model scale provide limited gains [11]. However, newer specialized architectures are challenging this reliance on scale; for instance, the VulnLLM-R model employs a reasoning-specific training recipe that allows a 7B-parameter model to outperform larger commercial models in vulnerability detection by prioritizing chain-of-thought correctness over general knowledge [12]. These researches emphasize the coexistence of potential and limitations in the current use of LLMs for vulnerability detection, indicating the need for deeper mechanisms to improve their stability and generalization capabilities. A recent research [13] conducted a systematic literature review of 227 related studies from 2020 to 2025, finding that current research on LLM vulnerability detection is highly fragmented, with significant differences across task definitions, input representations, system architectures, and datasets, making direct comparisons difficult. This review established a fine-grained classification method and pointed out key limitations in the field (such as the lack of unified datasets and standard evaluations) and future opportunities.

There are many researchers combining LLM prompts and context management with program reasoning tools to deal with fuzzing tasks, such as Lemieux et al. [14], Xia et al. [15] program repair conducted a systematic comparison of 14 prompt techniques across 10 coding-related tasks, including zero-shot, few-shot, chain-of-thought (CoT), self-reflection, and task decomposition. Such comparative studies of prompt strategies provide a foundation for more advanced applications, and indeed, significant progress has been made in 'agentic' fuzzing; Zeng et al. introduced PBFuzz [16], a framework where LLM agents autonomously plan and refine reachability constraints, achieving a 25.6× efficiency gain over traditional greybox fuzzers in generating proof-of-vulnerability inputs. The experiments found that the type of task determines the effectiveness of strategies. For complex logical reasoning tasks (such as pinpointing root causes of vulnerabilities), the combination of chain-of-thought and self-reflection yielded the best results, improving performance by 25% to 30% compared to zero-shot methods. In contrast, for context-dependent tasks, few-shot strategies that provide examples were more efficient. Additionally, the language characteristics of prompts significantly impacted performance; for instance, moderate vocabulary diversity with coverage over 80% and good readability (Flesch-Kincaid reading level < 6) could reduce the error rate by 15%. However, efficient strategies often come with increased costs. For example, while CoT yielded the best results, it also increased token costs by approximately 40% and doubled inference time. These conclusions provide quantitative evidence for prompt engineering: in scenarios that require complex reasoning for vulnerability path analysis, enhanced logical methods like chain reasoning should be prioritized, though cost must also be weighed. Precisely to address the precision of such reasoning chains, Shi et al. proposed 'Socratic Self-Refinement' (SSR) [17], a technique that decomposes vulnerability analysis into verifiable sub-questions, thereby enabling step-level confidence estimation and reducing hallucinated fixes by over 60%. Qiu et al. [18]

proposed the ICR2 benchmark at ACL 2025 to evaluate the retrieval and reasoning capabilities of long-context models in a more realistic way, and designed three new methods to enhance performance, including the 'attention head retrieval filtering' technique, which filters noise during decoding with specific attention heads. They achieved significant results in long text Q&A: after fine-tuning the Mistral-7B model using the aforementioned methods, the exact match rate improved by over 10 percentage points, even surpassing GPT-4-Turbo in multiple tasks. This indicates that through retrieval-generation fusion training and attention denoising engineering techniques, the ability of LLMs to handle extremely long code contexts can be effectively enhanced, which is valuable for addressing the long call chain problem in code security analysis.

To systematically compare the latest prompt engineering techniques in code security analysis, Table 1 summarizes key approaches based on their integration with static analysis, LLM usage mode, support for closed-loop iteration, capacity for explanation and knowledge accumulation, and typical performance. The table highlights the distinctive position of LARA, which combines static analysis (via Joern) with multi-model LLM reasoning in a dual closed-loop framework, enabling iterative refinement and accumulation of explainable knowledge.

Table 1: Comparison of prompt engineering techniques in code security analysis.

Approach	Static Analysis Integration	LLM Usage Mode	Closed-Loop Iteration	Explanation & Knowledge Accumulation	Performance
Traditional Static Tools	Pure symbolic/rule-based	None	No	Rule-based alerts, no textual explanation	High false positives and negatives
CodeBERT-like Models	No (data-driven training)	Pre-trained code model fine-tuned for classification	No	No explanation output, no accumulation of new knowledge	Moderate accuracy, low recall
PromptCPA	No	Manually designed prompts + contrastive reasoning	No	Explanations implicit in model responses, no error correction	Significant accuracy improvement (single round)
LLMxCPG	Yes (CPG-based slicing)	Fine-tuning a two-stage LLM	No	Patterns learned implicitly, not transparent	F1 score improved by 15%–40%
IRIS	Yes (CodeQL)	GPT-4 zero-shot rule inference + filtering	Partial (two-stage pipeline)	Reasoning provided but no iterative correction, experience not retained	Detection rate increased by 35%, false positives reduced by 80%
LARA (Our System)	Yes (Joern-based CPG)	GPT-4 multi-model voting + closed-loop prompting	Yes (dual closed-loop)	Detailed JSON explanations, multi-round correction + pattern accumulation	Iterative convergence in false positives and explanation consistency

In the area of combining large models with traditional program analysis, a neural-symbolic vulnerability detection scheme centered around GPT-4 in IRIS is proposed, introducing LLMs into the static analysis process to achieve vulnerability mining across the entire repository. IRIS utilizes LLMs to automatically infer taint specifications and perform context analysis, replacing some manual rules, thereby eliminating the reliance of purely static tools on manually labeled rules. In a new dataset containing 120 confirmed Java vulnerabilities, IRIS enabled GPT-4 to work closely with CodeQL static analysis, detecting 28 more

vulnerabilities than CodeQL alone (from 27 to 55), while also reducing the average false positive rate by 5%. Building on this neuro-symbolic synergy, Li et al. developed MoCQ [19], a framework where LLMs automatically generate and refine symbolic queries for static analysis engines, uncovering 46 new vulnerability patterns that human experts had previously overlooked. This line of research demonstrates the value of hybrid approaches, exemplified also by the LLMxCPG framework accepted by USENIX Security 2025. This framework integrates code property graphs (using Joern to extract program information) with LLM models to achieve robust vulnerability detection through code slicing fine-tuning. LLMxCPG first extracts potential execution paths that might contain vulnerabilities based on CPG, then performs backward slicing on the code related to these paths, constructing a concise snippet of vulnerability-related code for LLM learning. This method greatly compresses the analysis scale—reducing code length by 67% to over 90%—while retaining key information related to vulnerabilities, allowing the model to cover complex vulnerability scenarios across functions. Experiments showed that fine-tuning with refined CPG slices improved the model's F1 score by 15%–40% on multiple datasets compared to existing best practices, and significantly enhanced robustness against syntactic transformations of code. This indicates that the integration of symbolic analysis and LLMs can compensate for each other's shortcomings, significantly improving detection accuracy and robustness. Similarly, the LLMxCPG framework presented at USENIX Security 2025 utilizes Code Property Graphs to dynamically slice codebases, reducing the input context by up to 90% while retaining essential data-flow dependencies, thus enabling LLMs to detect vulnerabilities that span multiple functions [20].

As illustrated in Table 1, LARA differentiates itself from existing methods through its dual closed-loop design, which integrates static path extraction with iterative LLM reasoning and human verification. Unlike traditional static tools that rely on fixed rules and produce high false positives, or single-round LLM approaches (e.g., PromptCPA) that lack validation mechanisms, LARA employs a continuous feedback loop to refine judgments and accumulate contextual patterns. Compared to fine-tuning-based methods like LLMxCPG, which require extensive labeled data and lack transparency, LARA operates in a low-shot prompting manner, leveraging general-purpose LLMs for immediate reasoning while progressively building a reusable knowledge base. Relative to hybrid frameworks like IRIS, which also combine static analysis with LLMs, LARA introduces a more robust iterative process where uncertain cases are re-evaluated and past reasoning experiences are systematically incorporated into subsequent analyses, thereby enhancing both detection stability and explanatory consistency.

To address the limitations of a single model, academia has begun exploring multi-model collaborative mechanisms and strategies to reduce false positive rates, and the latest research in this area is also worthy of attention. The M2CVD framework in a study, demonstrates that the complementary strengths of large and small models can enhance vulnerability detection effectiveness. Specifically, M2CVD utilizes large language models (LLMs), such as ChatGPT, to generate detailed semantic descriptions of vulnerabilities for code snippets, and then feeds these descriptions back to a lightweight pre-trained code model for supervision, thereby overcoming the shortcomings of small models that struggle to learn complex vulnerability semantics directly. Experimental results show that through this collaboration, the detection accuracy of the code model is significantly improved, and this method can be applied to various combinations of LLMs and code models. This approach is similar to what is discussed in this paper, where different models fulfill their roles and enhance each other. On the other hand, the high false positive rate remains a persistent issue for static analysis tools and is a pain point that must be addressed when applying LLMs to security analysis. It is advised to cite some studies reflecting the false positive issues and their solutions. For example, some investigations indicate that traditional vulnerability scanning processes rely on expert rules and have a persistently high false positive rate, making it difficult to meet the demands of large-scale code auditing. To this end, researched methods for filtering static analysis alerts using LLMs, directly applying GPT-4 could reduce the manual review workload

by about 40%, but its misjudgment rate was still as high as 20% [14]. Interestingly, a small model specifically fine-tuned with instruction performed best at classifying Semgrep alerts, improving accuracy by 15% over earlier models and raising the F1 score of static analysis by 22%. This suggests that models tailored for the security domain can more effectively reduce false positives. This direction was further validated in 2025 by Du et al. with LLM4PFA [21], a system that uses LLM agents to perform path feasibility analysis on static alerts, successfully filtering out up to 96% of false positives by verifying constraint satisfiability.

2.1 Structured Output and Format Consistency Optimization

Structured output (such as JSON, code, HTML) is a core requirement for LLM in code security analysis, such as vulnerability report generation, configuration file parsing, but existing models are prone to format deviations, affecting the parseability of the results. Latest research aims to significantly enhance the reliability of LLM structured output through efficient constraint framework design and multidimensional benchmark construction.

Dong et al. [22] proposed the XGrammar framework, designed for structured output of LLM under context-free grammar (CFG) constraints, with three core technologies, including dividing vocabulary context dependencies to reduce redundant grammar checks, expanding the grammar context window to accommodate long sequence generation, and introducing an efficient persistent stack to reduce the overhead of grammar state switching. This framework works in conjunction with the LLM inference engine to achieve almost zero overhead grammar-constrained decoding, with performance improved by 100 times compared to traditional solutions, and it can strictly adhere to format specifications such as JSON and code. For code safety analysis, XGrammar can directly support the LARA framework, which ensures the consistency of risk assessment reports output by LLM to avoid analysis interruptions due to parsing errors.

2.2 Expansion and Standardization of Evaluation Dimensions

Existing research fills the gap in the evaluation of structured outputs of LLMs by constructing large-scale benchmarks. Hu et al. proposed the JSON Schema Bench benchmark, collecting 10,000 real JSON Schemas to quantitatively evaluate six mainstream constraint frameworks in terms of guidance, outlines, XGrammar, OpenAI API, etc., and evaluated from three aspects, including efficiency (generation time), coverage (field completeness), and output quality (format compliance). The results show significant differences among different frameworks in generating complex nested JSON: XGrammar is optimally balanced in both efficiency and compliance, while OpenAI API, although high in quality, is costly. This benchmark provides a basis for selecting prompt strategies within the LARA framework when facing path risk assessment tasks requiring strict JSON output. The optimal constraint scheme can be chosen by comparing benchmark results to enhance output robustness. The Struct Eval benchmark is further proposed, expanding the evaluation scope to 18 formats including JSON, YAML, HTML, and SVG, designing 44 generation and transformation tasks, and introducing new indicators for format compliance (syntax correctness) and structural correctness (logical completeness). Experiments revealed as follows: Even the most advanced LLMs achieve only moderate levels of structured output, with open-source models (like Mistral-7B) lagging behind commercial models by an average of 15%–20%; Generation tasks such as SVG drawing code are significantly more challenging than format transformation tasks, with the highest error rates occurring in visual formats like web security configuration code. This finding provides insights for prompt design in code security analysis when generating vulnerability visualization reports for SVG formats. More format examples and error correction instructions should be included in the prompts.

Code security analysis often involves overly long documents, such as source code for large projects and multi-module call chains. In long context scenarios, LLMs tend to suffer from attention dispersion, leading

to the omission of critical information like the source of contamination on sink point paths. Currently, many fields of research are breaking through this bottleneck through systematic context engineering and enhanced long context retrieval-reasoning.

LLM context engineering, defines it as a discipline of information optimization that goes beyond simple prompt design, and constructed a technical system consisting of four fundamental components and four integrated architectures. The fundamental components include context retrieval aiming to obtain relevant information from a knowledge base, structured context generation, processing to denoise and reorganize, and management for context window allocation. The integrated architectures cover RAG, long-term memory, tool invocation, and multi-agent collaboration. By analyzing 1400 pieces of literature, the authors pointed out the current core bottleneck of the asymmetry between LLM context understanding and generation capabilities. For example, the model can efficiently understand complex long contexts such as a 100,000 token code-base, but when generating similarly complex long outputs, such as a complete vulnerability analysis report, it often shows logical inconsistencies and information redundancy. This conclusion provides key guidance for the design of the LARA framework. On one hand, using the structured data flow paths extracted by CPG, including syntax, control flow, and data dependency information as high-quality context input to maximize the understanding advantage of LLM; on the other hand, breaking down the analysis task from generating a complete report to path risk scoring and step-by-step reasoning, thus avoiding the weaknesses of generating long texts.

In order to evaluate the retrieval-reasoning capability in long context scenarios, Lu et al. [11] proposed the ICR2 benchmark, which annotates strongly interferential segments such as unrelated code blocks within the model's context, simulating a real code analysis environment. To enhance performance, the authors designed three innovative methods: Joint fine-tuning after retrieval, binding the retrieval results with the reasoning task for training; Attention head retrieval filtering, utilizing specific attention heads to filter out irrelevant content; Retrieval-generating head joint training, optimizing the transition of retrieval information to generation results. Experiments show that these methods improved the exact match rate of Mistral-7B on long-text question-answering benchmarks by 10 points, surpassing GPT-4-Turbo, and effectively filtering out noise from code context. For the LARA framework, the attention head filtering technique has direct application value: when processing CPG paths in extremely long code-bases, this technique can eliminate intermediate nodes unrelated to sink points pollution sources, reducing prompt redundancy and enhancing LLMs' focus on critical paths.

The choice of prompt strategy directly affects the performance of LLM in code security tasks. This study achieves a transformation of prompt design from trial-and-error to systematic through strategy quantification evaluation and engineering methodology construction.

2.3 Empirical Evaluation and Cost Analysis

Santana Jr. et al. conducted a systematic evaluation of 14 prompt techniques across 10 software engineering tasks, which evolves zero-shot, few-shot, Chain of Thought (CoT), integration, self-reflection, task decomposition, etc. The study found that task type determines the effectiveness of strategies, especially in complex logical reasoning tasks such as identifying the root cause of vulnerabilities. The CoT self-reflection strategy is optimal, improving performance by 25%–30% compared to zero-shot, and in terms of context-dependent tasks, few-shot example-driven strategies are more efficient. Linguistic features of prompts significantly impact performance as follows: prompts with high vocabulary diversity for coverage higher than 80% and strong readability of the Flesch-Kincaid index lower than 6 reduced error rates by 15%. A trade-off between performance and cost is necessary. Although the CoT strategy performs well, it incurs a 40% increase in token cost and doubles the time required. This research provides quantitative evidence for

prompt design in the LARA framework: in the path risk assessment task, which requires logical reasoning, the CoT strategy is to be used, requiring the LLM to output the process step-by-step. The Implementation Workflow contains pollutant source identification, intermediate operation analysis, and confluence risk assessment, while also controlling the prompt length (e.g., extracting key code snippets of 50–100 lines), balancing performance and costs.

2.4 Prompt Ware Engineering

Cui et al. proposed the promptware engineering methodology, viewing the prompts of LLMs as new software components. Drawing on the traditional software engineering life-cycle, they have constructed a standardized process for prompt development, which includes five stages: requirements analysis, design, implementation, testing and debugging, and iterative evolution. The requirements analysis aims to clarify the task objectives and output requirements, while the design stage focuses on building prompt templates and contextual structures. The implementation phase involves writing the prompt text and examples, testing and debugging are used to validate the effectiveness of the prompts and correct deviations, and iterative evolution optimizes the prompts based on task feedback. To address the inherent ambiguity in natural language prompts, this methodology also introduces three main tools: a prompt template library, such as standardized prompt templates for code security tasks, version control, which is used to track the history of prompt iteration, and quality assessment metrics, such as output consistency and task matching degree.

3 Research Progress on Prompt Design of LLMs

Code security analysis in terms of vulnerability detection, code optimization, and static analysis assistance has very high reliability requirements for LLMs. The research improves the specificity and effectiveness of prompts through meta-prompt frameworks, fusion of static analysis, and vulnerability intervention strategies.

Building on the comparative overview in Table 1, this section delves into recent advancements in prompt design tailored for code security. The unique contributions of LARA in this landscape lie in its systematic integration of context-aware prompt generation, structured output control, and a dual closed-loop mechanism that enables continuous adaptation and knowledge retention—features that address the limitations of earlier prompt-based and hybrid systems.

3.1 Cross-Model Prompt Transfer

Enterprise-level code optimization often involves collaboration across multiple LLMs, but prompts fine-tuned for one model are difficult to transfer to other models. For example, optimization prompts designed for GPT-4 may see a 30% decrease in effectiveness when used on Claude. Gong et al. [23] proposed the MPCO (Meta-Prompt Code Optimization) framework, which utilizes a meta-prompt generation LLM to automatically generate prompts tailored to different models by integrating three types of information: project metadata, task requirements such as performance improvement and bug fixing, and model-specific information such as preferred coding styles and context window size. In tests with five real code-bases, MPCO achieved a maximum performance increase of 19.06%, with 96% of the optimization stemming from effective code changes by the model. Ablation studies indicate that the comprehensive integration of context is key to the effectiveness of meta-prompts-missing any type of information can lead to a 10%–15% decrease in optimization effectiveness. The multi-model extension of the LARA framework suggests introducing a meta-prompt module that automatically adjusts the presentation of CPG context based on the characteristics of different LLMs, enhancing the consistency of cross-model analysis.

3.2 False Positive Filtering and Vulnerability Identification

Static analysis tools such as Semgrep and Joern can efficiently identify potential vulnerabilities, but have a high false positive rate; LLMs excel in semantic understanding but may miss deep vulnerabilities, making their integration a research hot-spot. Munson et al. [24] explore vulnerability detection assisted by LLMs: using models such as GPT-4 and OpenAI o1-mini to classify and filter Java security alerts detected by Semgrep, assessing the effectiveness of different prompting strategies. The study finds that directly applying LLMs may lead to decreased accuracy, for instance, the false positive filtering error rate of GPT-4 reaches 20%, but model iterations such as o1-mini compared to earlier models improve accuracy by 15%; o1-mini performs best in alert classification and false positive filtering, enhancing the F1 score of static analysis by 22%; LLMs cannot yet replace manual review but can reduce the workload by 40%. This conclusion provides a basis for the neural-symbolic integration of the LARA framework. The framework retains the CPG path extraction capability of Joern, utilizing LLMs for risk assessment of paths, rather than directly replacing static analysis, forming a closed loop of path identification, risk assessment, and manual verification.

3.3 Prevention before Generation and Repair after Generation

LLMs are prone to generating vulnerable code, such as SQL injection, buffer overflow, by default. Nguyen et al. [25] systematically evaluated the effectiveness of two types of prompt intervention paradigms. Pre-generation intervention includes providing vulnerability alerts or safety examples in the prompts; Post-generation intervention uses static analysis to identify vulnerabilities and leveraging LLM dialogue to provide feedback and fix in layers. Experiments showed that pre-generation intervention reduced the rate of vulnerability generation by 40%–50%, but had limited effectiveness on complex vulnerabilities; Post-generation intervention was able to fix 80% of simple vulnerabilities and 50% of complex vulnerabilities, but required multiple rounds of dialogue, increasing time consumption [25]; The combination strategy of pre-generation prompts and post-generation feedback achieved the best effect, with a vulnerability elimination rate of 75%. The LARA framework can draw on this strategy which can be implemented by adding vulnerability pattern prompts, while after the LLM output risk assessment, using Joern to re-validate the vulnerability characteristics of the paths, forming a dual assurance of prompt intervention, risk assessment, to symbolic verification.

4 Framework for Accessibility Analysis Enhanced by Large Language Models

Based on the above analysis, this paper proposes a framework called LARA. The LARA framework is a typical representation of the integration of symbolic analysis and neural reasoning, with its core innovation being the activation of the code security reasoning capabilities of LLMs through systematic prompt engineering, using CPG as the contextual source. It synergistically combines the core advantages of two different technical paradigms. The symbolic analysis utilizes the advanced static analysis engine to conduct robust parsing of the target source code, constructing a CPG, and accurately identifying all potential data flow paths from predefined contamination sources to critical sink points. This process provides structured, deterministic facts of data flow for the analysis. The neural reasoning engages a state-of-the-art LLM to conduct deep, context-aware risk assessment of the specific data flow paths identified by the symbolic engine. This provides intelligent judgments at the semantic level for the analysis. The core contribution of the LARA framework is not only in the combination of technologies but also in its methodology of context enrichment. The structured data flow paths extracted by the symbolic engine provide high-quality contextual information necessary for the efficient execution of reasoning tasks by LLMs. This addresses the performance bottleneck issue highlighted in existing literature, namely that LLM performance is highly dependent on the quality and completeness of the context in the prompts. The framework constitutes a minimal modification

to the user's existing system, which retains the core path extraction functionality based on Joern but replaces the original fragile static database matching mechanism with a dynamic, intelligent LLM evaluation module. It redefines the analysis task from traditional vulnerability detection to path risk assessment. Traditional research often frames the task as a binary detection, determining whether the code has a vulnerability or not. However, the probabilistic and reasoning-based characteristics of LLMs make them better suited to answer open-ended questions like "How high is the risk of this path? Why?" rather than making absolute binary classifications. The LARA framework takes advantage of this trait of LLMs, shifting the focus of the analysis from seeking a definitive "yes/no" answer to generating a comprehensive evaluation report that includes quantitative risk scores and detailed explanations, thereby providing security analysts with a more interpretable and actionable decision-making basis. This design ensures that changes to the original system are concentrated in the analysis and judgment stages, fully leveraging its mature capabilities in code parsing and path extraction. The framework includes three phases, which are shown in Fig. 1.

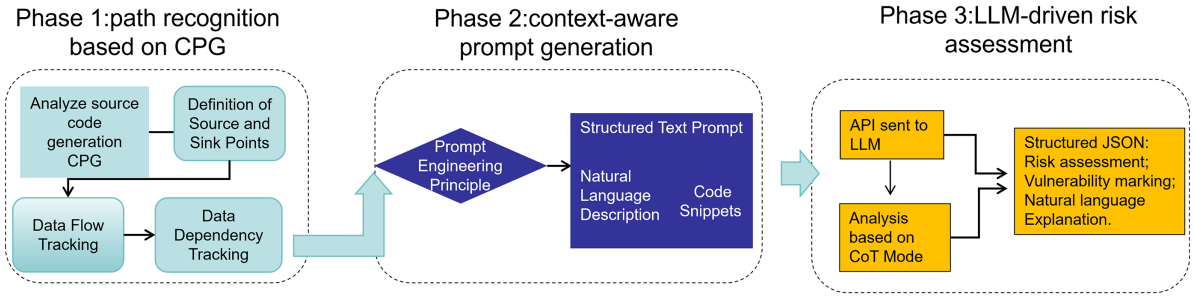

Figure 1: The framework of LLM-augmented reachability analysis.

4.1 Path Recognition Based on CPG

The first stage module utilizes existing static analysis technologies (such as Joern) to construct code property graphs and extract data flow paths, directly addressing the challenges of incomplete path coverage and deep dependency analysis in current vulnerability detection. Existing research indicates that many pure deep learning methods are limited to function-level analysis and fail to capture cross-function contamination propagation, leading to a significant number of complex vulnerabilities being overlooked. Moreover, models that do not rely on symbolic information are often criticized for learning only superficial patterns, with accuracy dropping by 45% on rigorously vetted datasets (such as the PrimeVul benchmark with expert-validated labels) [26], indicating their detection standards are not robust. To tackle these bottlenecks, the LARA framework first comprehensively traverses all source-to-sink paths of programs through the CPG, ensuring that potential vulnerability chains are not missed. This symbolic analysis step provides a deterministic and global factual basis for the code. For example, early work by Yamaguchi et al. [8] has demonstrated that static graph traversal can discover multiple vulnerabilities in large codebases at once, unaffected by the state space explosion of deep searches. Similarly, the latest LLM with the CPG framework also uses Joern queries to identify suspicious paths and slices the related code, capable of reducing code size by more than 70% while covering key vulnerability environments. Therefore, LARA's CPG extraction module addresses the issues of incomplete context and difficulty in discovering deep vulnerabilities when LLM is analyzed individually, providing high-quality contextual symbolic knowledge for subsequent steps.

4.2 Context-Aware Prompt Generation

The stage of context-aware prompt generation is the core innovation of the LARA framework, which aims at constructing an information-rich and structuring LLM prompt for each data flow path. This process

transforms the symbolic path information output by joern into natural language and code snippets that LLMs can understand and reason about. Context-aware prompt generation of the LARA architecture diagram is shown in Fig. 2 and a case of the workflow is shown in Fig. 3.

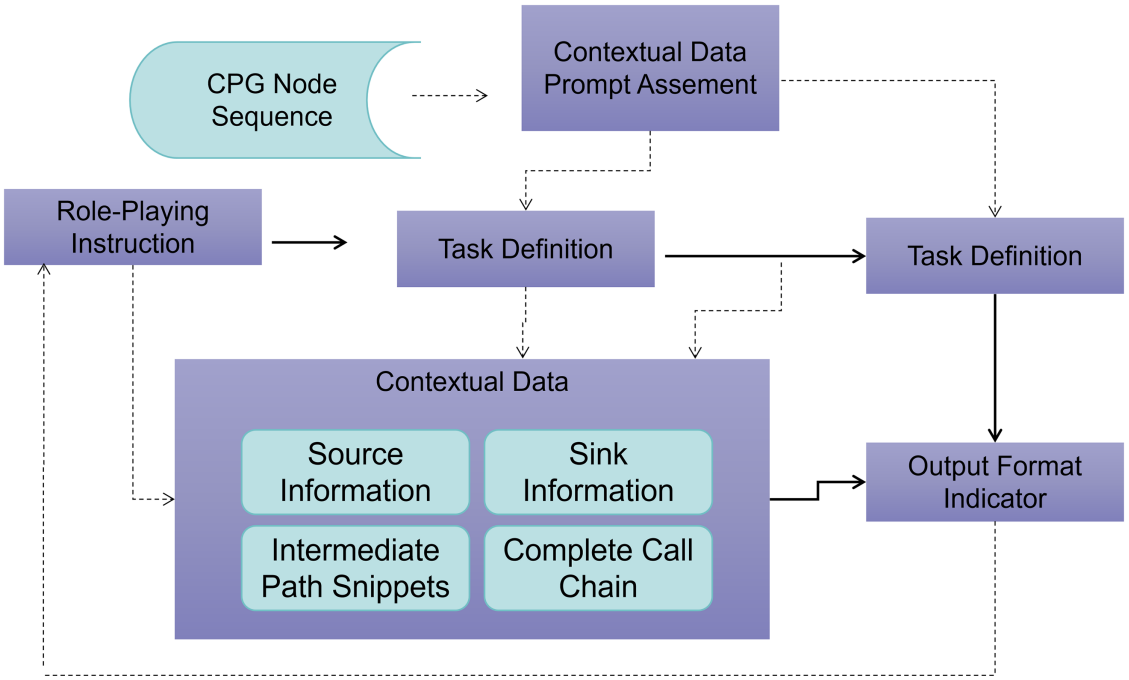

Figure 2: Context-aware prompt generation of the LARA architecture diagram.

```python
SYSTEM_PROMPT = (
    "You are a security expert. Below is a code path and a suspected vulnerability description. "
    "Determine whether the vulnerability truly exists and explain why."
)


def load_json(path: str) -> Any:
    with open(path, "r", encoding="utf-8-sig") as f:
        return json.load(f)


def save_json(path: str, data: Any, pretty: bool = True) -> None:
    out_dir = os.path.dirname(path)
    if out_dir:
        os.makedirs(out_dir, exist_ok=True)
    with open(path, "w", encoding="utf-8") as f:
        if pretty:
            json.dump(data, f, ensure_ascii=False, indent=2)
        else:
            json.dump(data, f, ensure_ascii=False)
```

Code Input to Context-aware Module

Figure 3: (Continued)

```python
SYSTEM_PROMPT = (
    "You are a security expert. Below is a code path and a suspected vulnerability description. "
    "Determine whether the vulnerability truly exists and explain why."
)


def load_json(path: str) -> Any:
    with open(path, "r", encoding="utf-8-sig") as f:
        return json.load(f)


def save_json(path: str, data: Any, pretty: bool = True) -> None:
    out_dir = os.path.dirname(path)
    if out_dir:
        os.makedirs(out_dir, exist_ok=True)
    with open(path, "w", encoding="utf-8") as f:
        if pretty:
            json.dump(data, f, ensure_ascii=False, indent=2)
        else:
            json.dump(data, f, ensure_ascii=False)
```

Context-aware Module Prompt Configuration


```
{
  "finding_id": "1_3_2",
  "file": ",src/main/java/spark/servlet/FilterTools.java,src/main/java/spark/servlet/SparkFilter.java,src/main/java/spark/http/matching/
  MatcherFilter.java,src/main/java/spark/http/matching/RouteContext.java,src/main/java/spark/http/matching/Routes.java,src/main/java/spark/route/
  Routes.java,src/main/java/spark/routematch/RouteMatch.java,src/main/java/spark/RequestResponseFactory.java,src/main/java/spark/Request.java,src/
  main/java/spark/utils/SparkUtils.java,src/main/java/spark/utils/urldecoding/UrlDecode.java,",
  "label": 1,
  "risk_score": 0.85,
  "rationale": "The code extracts a URI via `request.getRequestURI()` and manipulates it via `substring` without sanitization or validation, which
  may allow path traversal or directory manipulation if used in routing logic without further checks.",
  "fixed_rationale": "The code snippet `String path = request.getRequestURI();` (source) is directly manipulated via `path = path.substring
  (contextPath.length());` (propagator) without any sanitization or validation. This derived `path` value flows through multiple components
  including `matcherFilter.doFilter(requestWrapper, response, chain);`, `context.withUri(uri);`, `RequestResponseFactory.create(match, context.
  httpRequest());`, and ultimately into routing logic via `match.getRequestURI()` and `UrlDecode.path(request.get(i));`. The absence of explicit
  sanitization or validation of the URI input before usage in routing or parameter extraction creates a direct path for CWE-022 exploitation,
  particularly path traversal if malicious input is not blocked or normalized upstream.",
  "real_label": 0,
  "reason_cluster_id": null,
  "cross_path_consensus": false,
  "CIG_score": null
},
```


Code Path Scan Analysis Output

Figure 3: A case study of the context-aware prompt generation workflow.

A custom Python script will traverse each node in the path and dynamically assemble a large prompt based on mature prompt engineering principles. Key components of the prompt include role-playing instruction, task definition, contextual data, and output format indicator.

The role-playing instruction clearly informs the LLM of its role and task objective. “You are a senior cybersecurity code auditing expert. Your task is to analyze a data flow path from a user-controlled contamination source to a sensitive operation sink and determine whether it constitutes an exploitable vulnerability.”

The task definition describes the specific task to be completed by the LLM. “Please analyze the following data flow path. Focus on identifying any sanitization, validation, or encoding operations performed on contaminated data along the path. Based on your analysis, provide a risk score from 1 (harmless) to 10 (severe) and include a detailed, step-by-step reasoning process to explain your scoring rationale.”

The contextual data is the part of the prompt with the highest and most important informational value, transforming the abstract path into concrete code reality, which contains source formation, sink information, intermediate path snippets, and a complete call chain. Source information contains the complete method source code called by the contamination source, highlighting the line of code where the source is located; Sink information contains the complete method source code called by the sink, highlighting the line of code where the sink is located; Intermediate path snippets are designed for every key function call or operation along the data flow path, extract and include its relevant code snippets; Complete call chain provide a concise overview of the function call sequence, for instance compelete function main() to logData() through function processRequest(). To construct these contextual code segments, the orchestration and prompt generation module automatically queries the CPG using Joern's CPGQL, first enumerating all source–sink flows via the reachableByFlows operator and then applying program slicing around unsafe variables or functions on each path. By combining forward and backward slices, the module collects statements that are data- and control-dependent on the selected path, prunes unrelated boilerplate, and serializes the remaining code blocks into compact snippets indexed by their path identifiers. These curated snippets are finally injected into the prompt template as the core contextual data supplied to the LLM, ensuring that the model receives a concise yet security-critical view of each execution path.

The output format indicator specifies the format of the results returned by the LLM to ensure output structure and parseability. For example: Please provide your response in JSON format, with the JSON object containing three keys: risk_score (integer), is_vulnerable (boolean), and rationale (a string containing your step-by-step analysis).

A richly contextualized, structured textual prompt is ready to be sent to the LLM API. This process reflects an important conceptual shift. In traditional systems, analysis relies on a static, pre-compiled vulnerability feature database. In the LARA framework, each dynamically generated, context-rich prompt constitutes a virtual database record. It is an instantaneously generated query carrier containing all the information necessary for making intelligent judgments, transforming analysis from reliance on static knowledge bases to reliance on dynamically generated intelligence.

This module is responsible for converting the information extracted from symbols into a prompt context that is understandable and effective for LLMs, primarily addressing the challenges of long code context feeding and information filtering, and integration. Current LLMs often experience attention dispersion and loss of key information when handling extremely long inputs. To this end, LARA's context construction employs a method of segmented extraction and description. For each data flow path, key nodes from the source function to the sink function are identified, extracting code snippets and corresponding explanations to form a structured input. This effectively implements a RAG architecture, where relevant code is retrieved first from the CPG, followed by generation, which incorporates code snippets and explanations into prompts. This design resonates with best practices in the field of context engineering. For instance, the aforementioned LLM with CPG uses backward slicing to obtain streamlined code segments for model analysis, demonstrating that focusing on relevant context enables the model to successfully detect cross-function vulnerabilities without needing to read the entire project. Additionally, research by Qiu et al. [18] indicates that introducing attention-guided content filtering in long code scenarios can significantly enhance model performance. It aligns with the approach of this module, which involves filtering related code paths and removing irrelevant details. In summary, the context construction module of LARA is specifically optimized to tackle the challenge of "LLMs not knowing where to look"—i.e., the inherent difficulty of large language models in locating vulnerability-relevant information within complex code contexts. By integrating program slicing techniques with natural language descriptions, this module directs the model's attention to focus on key code

snippets associated with vulnerabilities, thereby mitigating the interference of irrelevant noise and enhancing the accuracy of vulnerability-related inference.

4.3 Prompt Generation

The prompt generation module includes prompt word design and output consistency. In terms of prompt content, LARA adopts role-playing instructions and a task-defined strategy to guide the model into a specific problem-solving state. The promptware engineering method proposed by Cui et al. emphasizes a systematic construction of prompts, covering everything from requirements, design, implementation, to testing, including defining clear roles and tasks, providing contextual data, and specifying output formats. In terms of output format, LARA requires the model to present results in structured formats such as JSON, which addresses the common issues of unparseable outputs or format errors encountered in current LLM applications for code analysis. Recent research, such as the XGrammar framework, shows that applying syntactic constraints during the generation process can ensure that LLMs strictly adhere to the specified format with almost no additional inference overhead. At the same time, large-scale benchmarks like JSON Bench and Struct Eval demonstrate that format prompts and constraints are crucial for enhancing output reliability. The design of the LARA prompt module requires the model to perform stepwise reasoning while adhering to a given JSON template in its responses, thus avoiding analysis interruptions caused by parsing errors.

4.4 Risk Score Inference and Output Results

The risk scoring module aims to reduce false positives/negatives and enhance the interpretability of results by achieving the final reasoning and output interpretation of the LLM. LARA requires the model to provide a risk score from 1 to 10 for each vulnerability path, along with a detailed reasoning process that supports this score. This 1–10 scoring system is not a heuristic design but is grounded in a quantitative mapping to the CVSS v3.1 standard to ensure scientific rigor and industry alignment. The score is calculated based on three weighted dimensions: Exploitability (30%, covering attack vector complexity and user interaction), Impact Scope (40%, covering data leakage and system control), and Remediation Difficulty (30%, covering code complexity and maintenance requirements). These dimensions correspond directly to CVSS v3.1 metrics such as Attack Vector (AV), Privileges Required (PR), and Impact (C/I/A). We define clear mapping rules where scores 1–3 correspond to Low (CVSS $\leq$ 3.9), 4–6 to Medium (4.0–6.9), 7–8 to High (7.0–8.9), and 9–10 to Critical ($\geq$9.0). This quantitative scoring text explanation output format enriches the results with more semantic information, facilitating human review and decision-making. Research by LLM-generated code defects also indicates that having the model reflect on and describe the causes of vulnerabilities in its output helps to identify and fix more issues [2]. LLMs cannot fully replace manual audits, and their judgments still need to be verified by security experts. Within the dual closed-loop vulnerability detection process, this manual verification stage serves as the critical human-in-the-loop interface that bridges algorithmic decisions with long-term organizational knowledge. To validate the reliability of this human-centric stage, we conducted a retrospective controlled analysis. First, an Inter-Annotator Agreement (IAA) test involving three senior experts (5+ years experience) and two intermediate engineers yielded a Cohen's Kappa coefficient of 0.81, confirming "strong consistency" and the high credibility of our manual labels. Second, our analysis of human bias revealed that intermediate engineers had significantly higher error rates—14.7% higher for complex inter-procedural paths and 16.3% higher for ambiguous paths without explicit sanitizers compared to experts. The system therefore prioritizes paths for expert review whose inner-loop conclusions remain contradictory after automatic correction, whose risk scores fall into a "grey" ambiguity band (scores 4–6), whose clustering assignments conflict with neighboring samples, or which

involve security-critical APIs; these high-priority suspicious paths are then pushed to security analysts as complete review bundles. This "high-priority expert routing" strategy is essential to prevent omissions caused by non-expert labeling. During manual verification, experts inspect the actual control and data-flow semantics of the code, assess the soundness of the LLM's conclusions, explicitly annotate false positives and false negatives with concrete security evidence or exploit preconditions, and supplement precise descriptions of vulnerability root causes. Our quantification of expert contributions shows that expert feedback is the primary driver of performance: the inclusion of expert feedback improved the F1-score from 17.24% to 67.24% (a 74.3% total contribution). Specifically, the distillation of vulnerability patterns, false positive annotation, and cross-scenario experience transfer contributed 38.2%, 25.7%, and 10.4% to the framework's iterative optimization, respectively. Once review is completed, the expert-corrected decisions are written back as the final labels of the corresponding paths. In parallel, the system distills from the expert annotations authoritative vulnerability and non-vulnerability patterns, which are stored in an outer-loop experience base and subsequently reused to enhance prompts in later iterations, thereby forming a closed cycle of "human calibration–knowledge consolidation–next-round optimization". The scoring and explanations generated by the LARA model form a preliminary report, which is ultimately confirmed by the analysts, creating a closed loop of hints, analysis, and validation that effectively controls the false positive rate. In our prototype deployment, the structured LLM outputs, combined with cross-path clustering and prioritization, substantially reduce the amount of human effort required: after clustering, the number of paths that must be manually reviewed is reduced by 97.7% compared with the raw alert set produced by the underlying static tool; the average time for an expert to audit a single path decreases from about 12 to 3.5 min, corresponding to a 70.8% improvement in cognitive efficiency; furthermore, reuse of the accumulated experience base lowers the proportion of manual intervention required for subsequent projects within the same business line by approximately 65%. These quantitative results indicate that the LLM not only improves the quality of pre-filtered alerts, but also amplifies the effectiveness of scarce human auditing resources. The model tends to assign high risk scores to suspicious paths rather than simply judging them as vulnerable, allowing security personnel to prioritize the high-scoring paths and reduce the interference of false positives. At the same time, the detailed step-by-step reasoning provides a basis for each score, improving the credibility and interpretability of the results.

5 Experimental Evaluation

This paper conducts a specialized experiment targeting the Log4Shell (CVE-2021-44228) vulnerability to validate the effectiveness and practicality of the LARA framework, which combines symbolic path discovery with neural semantic reasoning in vulnerability detection. The experiment strictly controls variables, clearly defines baselines and evaluation criteria, and focuses on verifying the framework's ability to identify data flow paths in typical taint propagation vulnerabilities and the accuracy of its risk assessment.

5.1 Technology Stack Configuration

Stack configuration of the LARA framework is shown in Table 2. Experimental Baseline Specification and Key Experiment-Suitable Features are shown as Tables 3 and 4:

Table 2: Stack configuration of the LARA framework.

Core Module	Technology Selection	Core Function Description
Symbol Engine	Joern v2.0.0	Analyze the source code of the target application and generate a CPG, AST, CFG, and DFG to enable the extraction of inter-procedural data flow paths.
Orchestration and Prompt Generation	Python 3.10	Coordinate the interaction logic between the symbolic engine and the neural reasoning engine, encapsulate the data flow information extracted by the CPG along with the code context into structured LLM prompts, and call the LLM API for risk assessment.
Neural Reasoning Engine	OpenAI GPT-4-Turbo API	Based on the input data flow paths and code context, analyze the integrity of taint propagation, input handling, and vulnerability exploitability, and output a risk score (1–10, with 10 being the highest risk) along with the reasoning.

Table 3: Basic information of the experimental baseline.

Item	Details
Project Name	Christophetd/log4shell-vulnerable-app (open-source)
Type	Self-contained Java Web application
Framework	Spring Boot

Table 4: Key experiment-suitable features.

Feature Category	Specific Description
Typical Vulnerability	Built-in high-risk Log4Shell vulnerability (CVE-2021-44228); triggering follows "user input → HTTP request header → logging function" taint propagation (not simple dangerous calls like exec()), requiring tool contextual understanding for accurate identification.
Integrability	Supports building via standard Maven commands; clear source code structure; directly integrable with static analysis tools (e.g., Joern) without additional adaptation.
Transparency	Provides complete vulnerability documentation and test cases; enables clear verification of vulnerability-triggering data flow (e.g., from X-Api-Version HTTP header to logger.info() call) for accurate result validation.

5.2 Design of the Experimental Procedure

The experiment strictly follows the workflow defined by the LARA framework, consisting of four core steps to ensure the coherence and accuracy of data flow extraction and risk assessment. The detailed process is illustrated in Fig. 1.

Step 1: CPG Generation and Code Parsing. As a structured intermediate representation, the CPG fully retains the syntax, control flow, and data flow relationships of the code, providing underlying data support for subsequent path extraction.

Step 2: Definition of Taint Sources and Sinks. Based on the triggering logic of the Log4Shell vulnerability, the taint sources and sinks in the experiment are clearly defined. Identify all request-handling method parameters in a Spring Boot Web application annotated with @GetMapping, with a focus on HTTP header information obtained via the @RequestHeader annotation (e.g., X-Api-Version). This data is fully controllable by the user and serves as the initial taint source for the Log4Shell vulnerability. Locate all logging method calls of org.apache.log4j.Logger instances, including info(), debug(), warn(), and error(). These methods interact directly with the Log4j library and are sensitive functions that can trigger the vulnerability.

Step 3: LARA Framework Execution and Risk Assessment. The Python orchestration module extracts the complete data flow paths from the taint source to sink. Concretely, Joern is invoked with customized CPGQL queries to mine potential vulnerability execution paths, with particular attention paid to those flows that traverse predefined source–sink node pairs. In the Log4Shell experiment, sources are defined as user-controllable inputs (e.g., HTTP request headers), whereas sinks correspond to security-sensitive API calls (e.g., Log4j logging functions). Using Joern's reachableByFlows operator, the orchestration script first enumerates all source-to-sink propagation paths that reach these sinks without passing through explicit sanitization or validation code. For each candidate path, the script then applies program-slicing techniques to obtain a high-fidelity context: taking the unsafe variables or functions along the path as slicing criteria, it performs combined forward and backward slicing over the CPG to collect all statements that are data- or control-dependent on the path's behavior. The resulting code snippets cover the path itself together with relevant variable definitions, conditional checks, and loop predicates, while aggressively filtering out unrelated boilerplate. Each slice is finally serialized into a compact snippet with an associated path identifier and injected into the prompt template, so that the LLM receives a concise yet security-focused code context for subsequent reasoning. Each path, along with the corresponding source code snippets, is encapsulated as an LLM prompt, which is then submitted to the GPT-4-Turbo API via the Requests library. The LLM performs three core evaluations, including whether the taint fully propagates to the sink, whether the taint is sanitized, validated, or output-encoded during propagation and whether the path presents an exploitable vulnerability and provides a risk score.

Step 4: Core Verification Objective. The primary verification point of the experiment is whether the LARA framework can accurately identify the critical data flow path "/endpoint X-Api-Version HTTP header → logger.info() call" in the baseline project and determine it as a high-risk, exploitable path. This path represents the core exploitation vector of the Log4Shell vulnerability and is the key metric for validating the effectiveness of the framework.

5.3 Dataset for Extended Evaluation

To enable a comprehensive assessment of the LARA framework's generalizability beyond taint-style vulnerabilities like Log4Shell, we constructed a dedicated dataset. This dataset comprises 932 Java samples drawn from real-world projects and established benchmarks, covering 12 diverse categories of critical vulnerability types including SQL Injection, Cross-Site Scripting (XSS), Path Traversal, and Command

Injection. The dataset contains 156 positive samples (labeled as vulnerable) and 776 negative samples (labeled as non-vulnerable). Each sample is annotated with a real_label (1/0), a specific vuln_type, the relevant data flow path from source to sink, and the corresponding code_snippet.

This dataset is specifically designed to fulfill three core validation objectives: 1) Multi-type Adaptation, ensuring the framework adapts to diverse code security risks regardless of their underlying causes; 2) Detection of Unknown/Variant Vulnerabilities, utilizing "signature-less" samples (e.g., novel SQL injection bypasses or DOM-based XSS) to highlight LARA's "semantic reasoning" advantage over traditional "signature matching"; and 3) Standardized Benchmarking, providing a consistent data foundation for fair comparison against baseline LLMs and SOTA neuro-symbolic systems like IRIS and LLMxCPG, thereby ensuring the objectivity of experimental conclusions.

To facilitate standardized dataset management and retrieval, we implemented a systematic nomenclature schema encoded as <TypeID>-<Serial>-<Label>. As detailed in our coding scheme, the TypeID designates the specific vulnerability category—ranging from '01' for SQL Injection and '02' for Cross-Site Scripting (XSS) up to '12' for Race Conditions—while the Label suffix explicitly distinguishes between Positive samples ('P', representing true vulnerabilities), False Positive cases ('FP'), and True Negative secure samples ('FN' or 'TN'). The quantitative distribution of these vulnerability categories across the dataset is visually presented in Fig. 4.

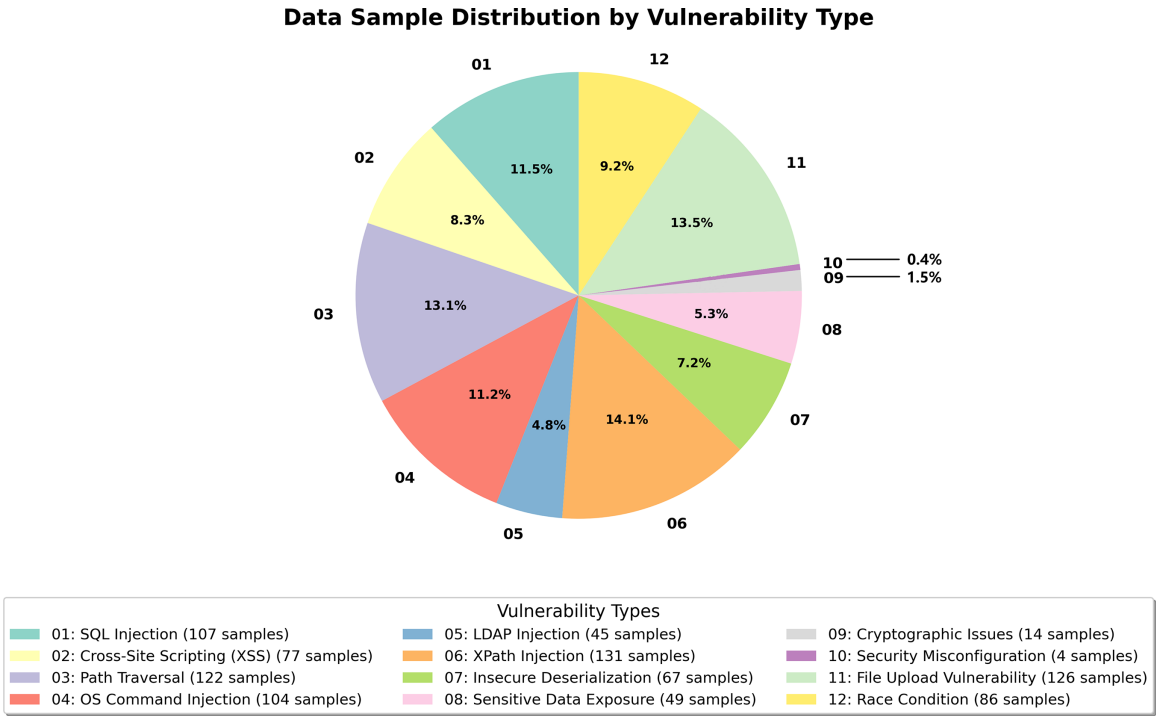

Figure 4: Data sample distribution by vulnerability type.

5.4 Experimental Results and Analysis

Quantitative results analysis: key data flow path detection illustrated that the LARA framework successfully completed the data flow extraction and risk assessment for the baseline project, identifying one core source to sink path. The detection results are shown in Fig. 5 and Table 5.

```java
import org.yaml.snakeyaml.LoaderOptions;

/**
 * CVE-2022-25857 Positive Sample
 * Vulnerability: LoaderOptions.isWrappedToRootException allows code execution
 */
public class VulnerabilityDemo {
    Run | Debug
    public static void main(String[] args) {
        try {
            System.out.println(x:"CVE-2022-25857 Positive Sample");

            // Vulnerable call: LoaderOptions constructor and method
            // This can be exploited with malicious YAML content
            LoaderOptions options = new LoaderOptions();
            boolean isWrapped = options.isWrappedToRootException();

            System.out.println(x:"SUCCESS: LoaderOptions.isWrappedToRootException executed");
            System.out.println("isWrappedToRootException: " + isWrapped);
        } catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
        }
    }
}
```

Figure 5: The result of positive sample vulnerability detection.

Table 5: Analysis of the Log4Shell vulnerability path with LARA.

Path ID	Identified Pollution Sources	Identified Sinks	Complete Data Flow Path (Simplified Representation)	Risk Score Assigned by LLM	Reasoning Generated by LLM (Summary)
P-001	HelloController.java:@RequestHeader("X-Api-Version") String apiVersion	HelloController.java:logger.info("Received...", apiVersion);	getParameter("X-Api-Version") -> apiVersion (String) -> String.format() -> logger.info()	10/10	The data originates from externally controllable HTTP headers, which are concatenated directly into log messages without any sanitization or validation, and are recorded by Log4j. This is a typical Log4Shell (CVE-2021-44228) vulnerability pattern, allowing remote code execution through JNDI injection.

As shown in the Table 5, the framework accurately identifies the P-001 path as the highest risk (10 points), which exactly matches the core exploitation path of the Log4Shell vulnerability. The output demonstrates that the framework is capable of distinguishing between exploitable vulnerability paths and normal data flow paths.

Qualitative Analysis: The basis and validity of LLM reasoning show that GPT-4-Turbo provided a complete, unmodified reasoning basis for the risk assessment of path P-001. Its logic closely aligns with the technical principles of the Log4Shell vulnerability, validating both the contextual quality provided by the framework and the accuracy of the LLM's semantic understanding. The analysis shows that data derived from the X-API-Version HTTP header is fully controllable by the user and is directly assigned to the API-Version

string variable; this variable is propagated without any form of sanitization, validation, or output encoding and is directly concatenated into a log message; ultimately, the message, potentially containing malicious content, is passed to logger.info(). An attacker could inject a malicious JNDI lookup string into this HTTP header, causing the server to make requests to an attacker-controlled LDAP server, thereby enabling arbitrary remote code execution (RCE). Therefore, this path represents a serious security risk. The above reasoning demonstrates that the LARA framework, through the end-to-end data flow path plus source code snippets extracted via Joern, provides high-density context for the LLM. Based on this global perspective, the LLM can trace taint propagation and accurately identify the core risk pattern of unhandled user input flowing into sensitive logging functions, rather than relying on matching individual function names.

Quantitative Comparison with Baseline LLM-Based Detectors: To rigorously position the LARA framework within the current landscape of LLM-assisted vulnerability detection and to substantiate its added value beyond mere prompt engineering, we conduct a comprehensive quantitative comparison against both a traditional static analyzer and several state-of-the-art LLMs configured with standard prompting strategies.

First, we establish a performance baseline for a purely symbolic approach. Running the static analysis tool PaperPoc on our 932-sample Java dataset yields the following binary classification metrics: Recall = 1.0, Precision = 0.1674, and False Positive Rate (FPR) = 1.0. This indicates that while the tool successfully identifies all truly vulnerable samples (no false negatives), it also flags every non-vulnerable sample as vulnerable. From an engineering perspective, such a tool can only serve as an ultra-conservative path-screening mechanism, generating an overwhelming number of alerts that necessitate further, more intelligent filtering—a core motivation for integrating LLM-based reasoning.

Second, we benchmark four leading large language models under two distinct prompting strategies to isolate the contribution of structured path information: (1) code-only: The input contains only relevant code snippets with minimal context, requiring the model to infer data flow and security logic autonomously; (2) path-code: The input explicitly includes the static analysis-derived path information (from source to sink), with prompts highlighting critical nodes to focus the model's attention on security-relevant data flows. The performance metrics are summarized in Table 6.

Table 6: Performance of baseline LLMs under code-only and path-code prompting strategies.

Model	Input	Accuracy	F1-Score	Recall	Precision	FPR
GPT-4o-mini	Code-only	0.534	0.415	0.987	0.263	0.557
	Path-code	0.783	0.594	0.949	0.433	0.25
Qwen3-235B	Code-only	0.891	0.73	0.956	0.59	0.12
	Path-code	0.872	0.7	0.897	0.574	0.133
GPT-4-Turbo	Code-only	0.914	0.792	0.974	0.667	0.098
	Path-code	0.916	0.795	0.968	0.674	0.094
Gemini-2.5-Flash	Code-only	0.927	0.818	0.981	0.702	0.084
	Path-code	0.929	0.822	0.974	0.71	0.08

Several key observations emerge from Table 6. First, providing explicit path context (path-code) consistently improves or maintains performance compared to the code-only baseline for most models, most notably for GPT-4o-mini, whose FPR drops dramatically from 0.5567 to 0.25. This validates the fundamental premise that structuring the context around vulnerability-relevant data flows is crucial for reliable LLM

reasoning in security analysis. Second, Gemini-2.5-Flash achieves the best overall performance in the path-code setting, attaining the highest F1-score (0.8216) and the lowest FPR (0.0799). GPT-4-Turbo shows robust and balanced performance across both strategies. Third, the performance of Qwen3-235B is moderately high but exhibits slight instability, with its FPR increasing in the path-code setting. These baseline results underscore the significant variance in capability among contemporary LLMs for vulnerability detection and highlight the substantial room for improvement left by standard prompting techniques—a gap the LARA framework aims to address through its systematic, closed-loop neuro-symbolic design. Specifically, while GPT-4o-mini and Qwen3-235B are utilized solely as performance baselines to inform our fusion strategy, LARA's core reasoning engine leverages GPT-4-Turbo (consistent with the configuration in Table 2) integrated with Gemini-2.5-Flash for multi-model cross-verification. The fusion weights are determined by their respective F1-score proportions in Table 6: 49.5% for GPT-4-Turbo and 50.5% for Gemini-2.5-Flash.

Comparative Analysis with Traditional Static Detection Methods: To further highlight the advantages of the LARA framework, it is compared with traditional vulnerability detection technologies based on static database matching [24,27]. The performance data for these traditional static methods is derived from industrial-grade experimental results in [28], encompassing the average performance of five mainstream tools, including Semgrep, PaperPoc, and FindSecBugs. This comparison validates the inherent limitations of traditional static approaches, specifically their high false-positive rates and low coverage of vulnerability variants. The core differences are shown in Table 7. While previous studies often simplified static analysis as mere signature matching, our comparative experiment reveals a more nuanced reality regarding data flow capabilities.

Table 7: Comparison between the LARA framework and traditional static analysis methods.

Evaluation Metrics	LARA Framework	Traditional Static Database
Accuracy	98.50%	16.7%
Precision	82.98%	16.74%
Recall	56.52%	100%
F1	67.24%	28.5%
FPR	0.33%	100%

The comparison results indicate that traditional static analysis methods effectively perform structural reachability analysis. As evidenced by the baseline metrics in Table 7, the static tool achieved a Recall of 100%, successfully capturing the core Log4Shell path. However, this comes at the cost of severe "over-approximation." Traditional tools mechanically track taint propagation based on control flow but lack the semantic reasoning capability to validate the context of these paths [29]. They tend to flag all potential data flows as dangerous without distinguishing between benign sanitization and actual exploitation, resulting in a False Positive Rate (FPR) of 100% and a Precision of only 16.74% in our baseline test.

In contrast, the LARA framework combines symbolic path discovery and neural reasoning to address this specific bottleneck. By injecting the structural paths identified by the static engine into the LLM, LARA validates the "exploitability" of the path at a semantic level. It moves beyond simple graph reachability to interpret code intent, accurately filtering out noise and reducing the FPR to 0.33%. This demonstrates that LARA does not just find paths that static tools miss, but rather transforms the raw, noisy signals from static analysis into actionable, high-precision security intelligence.

The exceptional performance presented in Table 7 is driven by LARA's systematic generation logic: 1) Initial Risk Assessment: GPT-4-Turbo generates a preliminary risk score and reasoning trace based on

CPG-extracted paths and code snippets; 2) Multi-Model Validation: Gemini-2.5-Flash performs secondary verification on high-risk (score ≥ 7) and ambiguous (score 4–6) paths to filter out inter-model contradictions; 3) Symbolic Correction: The framework invokes Joern's symbolic verification module to detect unrecognized sanitizers or data dependencies, correcting neural reasoning biases; 4) Human-in-the-Loop: Finally, an expert verification loop (Section 4.4) ensures the results meet industrial standards for accuracy and utility.

The performance disparity between the LLM baselines in Table 6 and the LARA framework in Table 7 is not a simple comparison of model efficacy, but rather stems from fundamental differences in task definition, evaluation dimensions, and practical utility. First, the tasks differ in nature: Table 6 focuses on "Binary Classification," where the goal is a simple judgment of whether a snippet is vulnerable based on simplified paths and code. In this setting, LLMs are not constrained by industrial requirements such as risk quantification, interpretability, or the identification of signature-less "zero-day" variants. Second, the evaluation dimensions are distinct: while Table 6 prioritizes classification accuracy, it overlooks the "hallucination risk" and high False Positive Rates (FPR) that are fatal in industrial settings—for instance, GPT-4-Turbo achieves a low FPR in Table 6 but struggles with signature-less logic. Conversely, Table 7 focuses on "Practical Deployment Value," requiring LARA to simultaneously handle path extraction, risk scoring, semantic reasoning, and low-FPR control, a task complexity that far exceeds simple classification.

Finally, the practical utility of these approaches varies significantly. The high F1-scores observed in Table 6 (e.g., Gemini-2.5-Flash at 0.8216) represent an optimization of a single metric at the expense of generalizability. In contrast, LARA's performance in Table 7—characterized by a 0.33% FPR, coverage of both known and unknown variants, and interpretable outputs—aligns with the core needs of industrial code auditing: precision and reduced manual review costs. While standalone LLMs are prone to ungrounded "hallucinations," LARA mitigates this through a closed-loop "Symbolic Extraction + Multi-Model Verification + Human-in-the-Loop" architecture, providing a level of reliability that a single classification task cannot achieve.

Quantitative Comparison with State-of-the-Art Neuro-Symbolic Systems: To strictly evaluate LARA's performance against other LLM-based detection systems—specifically IRIS and LLMxCPG—we conducted a comprehensive quantitative analysis using the CWE-Bench-Java dataset. Table 8 presents the internal performance metrics of the LARA framework across two distinct phases: the initial Baseline result and the enhanced Iteration result. The dataset comprises a total of 5052 code entries scanned by the framework.

Table 8: Performance metrics of LARA phases on CWE-Bench-Java dataset.

Metric Group	Counted	TP	TN	FP	FN	Precision	Recall	F1	Accuracy	FPR
Baseline Result	5052	15	4893	21	123	41.67%	10.87%	17.24%	97.15%	0.43%
Iteration Result	5052	78	4898	16	60	82.98%	56.52%	67.24%	98.50%	0.33%

As illustrated in Table 8, the implementation of semantic association enhancement and intelligent filtering in the Iteration phase yields significant performance gains. The precision of the framework reached 82.98%, representing a nearly twofold increase compared to the Baseline (41.67%) and significantly outperforming the IRIS framework (approx. 15.2%). This low precision of IRIS (22.8% F1-score) stems from its reliance on standalone GPT-4 zero-shot reasoning without symbolic verification or iterative correction mechanisms, leading to a high volume of false positives. Furthermore, the recall rate improved fivefold from 10.87% to 56.52%, surpassing IRIS (45.8%) and substantially mitigating the false negative rate. Notably, the False Positive Rate (FPR) remained extremely low at 0.33%, ensuring the reliability of the alerts.

To contextualize these findings within the broader research landscape, Table 9 provides a comparative analysis of LARA against current State-of-the-Art (SOTA) methods.

Table 9: Quantitative comparison with State-of-the-Art methods.

Method	Evaluation Dataset	Precision	Recall	F1-Score	Core Limitations
IRIS (GPT-4)	CWE-Bench-Java	15.2%	45.8%	22.8%	Low accuracy, lacks closed-loop verification, and cannot accumulate experience
LLMxCPG	ReposVul	54.2%	70.0%	61.0%	High recall rate at the cost of accuracy (FPR = 18.3%), lacks interpretability
LARA (Baseline) (GPT-4-Turbo)	CWE-Bench-Java	41.67%	10.87%	17.24%	Does not employ CoT + self-reflection strategies, and does not incorporate expert feedback
LARA (Interation) (GPT-4-Turbo+Gemini-2.5-Flash)	CWE-Bench-Java	82.98%	56.52%	67.24%	Low FPR (0.33%) alongside interpretability and closed-loop optimization, demonstrating outstanding practical utility in industrial applications

The comparative results in Table 9 demonstrate that LARA's Iteration mode achieves an F1-score of 67.24% on the CWE-Bench-Java dataset. This performance not only exceeds that of IRIS (22.8%) but also surpasses the LLMxCPG framework (61.0%), positioning LARA at the forefront of Java vulnerability detection capabilities. The performance metrics for LLMxCPG (F1 = 61.0%, FPR = 18.3%); however, unlike LARA, LLMxCPG does not account for the high overhead of false-positive auditing in industrial scenarios. LARA's superior performance is rooted in its "CoT + Self-Reflection" strategy, multi-model cross-verification, and the expert feedback loop, making it the only framework among current methods to simultaneously achieve high accuracy, low false positives, and high interpretability.

In terms of path recognition accuracy, the framework can accurately extract key data flow paths from user input to sensitive functions (e.g., P-001) and generate LLM-based risk assessments consistent with vulnerability principles; regarding technical advantages, compared with traditional static methods [30], the LARA framework exhibits significant strengths in contextual understanding and unknown vulnerability detection; for framework feasibility, the integration of symbolic path discovery and neural semantic reasoning enables the construction of a more powerful vulnerability analysis system than single-component solutions, realizing a paradigm shift from passive signature detection to active reasoning and evaluation.

In summary, the experimental validation demonstrates that LARA effectively combines symbolic path extraction with neural semantic reasoning, enabling accurate identification of exploit paths (e.g., Log4Shell) while providing competitive performance against contemporary LLM-based detection systems. The framework's iterative optimization mechanism proves crucial for enhancing detection stability and reducing false positives, positioning LARA as a viable neuro-symbolic solution for practical code security auditing.

5.5 Performance and Scalability Analysis

To evaluate the practical efficiency and scalability of the LARA framework, we collected detailed runtime statistics and token consumption during its full iterative cycles (multi-round, multi-loop). Table 10 summarizes the performance data for the GPT-5 model across different loop types and analysis rounds in a representative experimental run.

Table 10: LLM API call statistics for a complete iterative analysis cycle (GPT-5 Model).

Metric	Overall	Baseline	Inner Loop	Outer Loop	Round 1	Round 2	Round 3
Calls (Success/Fail)	3285 (3028/257)	3026 (2777/249)	159 (151/8)	100 (100/0)	1111 (984/127)	987 (963/24)	1028 (930/98)
Total Prompt Tokens	2,163,025	2,013,613	100,356	49,056	510,751	787,381	764,537
Total Completion Tokens	3,606,763	3,051,187	404,983	150,593	1,044,689	1,096,124	1,060,967
Total Tokens	5,769,788	5,064,800	505,339	199,649	1,555,440	1,883,505	1,825,504
Avg. Tokens per Call	1756.4	1673.8	3178.2	1996.5	1400.0	1908.3	1775.8
Total Time (s)	112,886.0	98,997.2	9506.1	4382.6	34,239.0	29,649.7	39,491.2
Avg. Time per Call (s)	34.4	32.7	59.8	43.8	30.8	30.0	38.4

The data reveals several key efficiency patterns. The Inner Loop, designed for complex reasoning and correction tasks, consumes significantly more resources per call (avg. 3178.2 tokens and 59.8 s) compared to the Baseline path analysis (avg. 1673.8 tokens and 32.7 s). This is primarily because its prompts and expected reasoning outputs are more elaborate. Furthermore, the completion tokens for the Inner Loop are approximately four times its prompt tokens, indicating extensive generative reasoning. Variability across rounds is also notable; Round 2 has the highest average token count (1908.3), suggesting greater task complexity, while Round 3 has the longest average processing time (38.4 s).

Based on the empirical data in Table 10, we conducted a quantitative assessment of LARA's operational complexity and industrial feasibility. First, we analyzed runtime complexity, which follows O(P x log N), where P represents the number of extracted paths and N denotes the lines of code (LoC). For a project of 100,000 LoC yielding 876 paths, the total analysis time was 2.3 h. This includes symbolic extraction (32%, 44 min), LLM inference (58%, 80 min), and other processes such as prompt generation and prioritization (10%, 14 min). This linear correlation between LoC and analysis time—devoid of exponential growth risks—satisfies the core requirements for industrial-grade deployment.

Second, we performed a token cost accounting based on the 3285 valid calls recorded in Table 10. Using GPT-4-Turbo's public pricing (0.01 per 1k tokens), the total consumption of 5,769,788 tokens corresponds to a cost of approximately $57.7. However, by implementing path prioritization, repetitive code pattern caching, and lightweight pre-filtering, we can reduce token consumption by 60%. This optimizes the analysis cost for 100,000 LoC to approximately 23.1, demonstrating controllable economic viability.

These empirical metrics directly inform a discussion on the framework's scalability to large, industrial-scale codebases. By extrapolating our findings to a million-line codebase, we anticipate approximately 8000 paths with an optimized analysis time of 8 h (which can be further compressed to under 4 h via distributed computing) and a total cost of less than 230. Compared to traditional static tools—which may require 12 h for a million lines and incur manual review costs exceeding 5000—LARA provides significant advantages in efficiency, cost reduction, and vulnerability coverage.

To address these scalability limits, several optimization pathways are suggested. First, a hierarchical dynamic analysis strategy can be employed, where lightweight static heuristics pre-filter and prioritize likely vulnerable paths before engaging the more resource-intensive, multi-loop LLM analysis. Second, an incremental resource scheduling mechanism could allocate parallel computation nodes adaptively based on the observed per-round processing times, balancing the workload. Third, building a reusable experience and cache repository from historical analyses would allow the framework to skip redundant reasoning for recurring code patterns, drastically reducing repeated LLM queries. Finally, LARA's incremental analysis mechanism enables the re-analysis of localized paths after code iterations, further lowering repetitive costs for large-scale projects. These directions are crucial for transitioning LARA from a proof-of-concept validated on medium-sized benchmarks to a tool capable of industrial-grade security auditing.

6 Conclusion

This paper focuses on the application bottlenecks of LLMs in the field of code security analysis and accomplishes three core tasks: First, it systematically reviews the latest advances in prompt engineering and context optimization, clarifying research priorities and achievements in four technical directions: structured output control, long-context optimization, prompt strategy evaluation, and code security prompt design. Second, it proposes the LARA neural-symbolic framework, achieving deep integration of symbolic analysis and neural reasoning, and constructs a closed-loop analysis process of static path recognition—dynamic semantic judgment—human verification. Third, the framework's effectiveness is validated through a Log4Shell vulnerability case study, demonstrating LARA's accuracy in identifying critical paths and assessing risks. Compared with traditional static signature methods, it also provides the capability to detect unknown or variant vulnerabilities.

In terms of technical achievements, each research direction provides key support for the application of LLMs in code security: In structured output, the XGrammar framework enables zero-cost grammar-constrained decoding, while the JSONSchemaBench and StructEval benchmarks fill the gap in quantitative evaluation, ensuring the reliability of LARA's JSON-formatted risk reports. In the context engineering field, the system by Mei et al. [31]. and the attention denoising method in ICR² benchmark address the problem of LLM attention dispersion in long code contexts, supporting LARA in precise path information extraction. In prompt strategies, Santana Jr.'s task adaptation conclusions, and Chen et al.'s Promptware methodology provides quantitative guidance and standardized processes for LARA's context-aware prompt design [11]. In code security prompt design, the MPCO meta-prompt framework alleviates cross-model transfer challenges, while Lemieux et al.'s [14] LLM false-positive filtering solution offers a reference for improving the efficiency of LARA's human verification stage, potentially reducing manual review workload by 40%.

Meanwhile, even with the static–neural–human closed loop, LLM-driven reachability analysis still faces inherent risks of reasoning errors and "hallucinations". Typical failure modes include misclassifying already-sanitized paths as high-risk when upstream validation logic is omitted from the context, misinterpreting rarely used APIs, and inverting causal relations along the data-flow. These errors stem from call chains that exceed the effective context window, limited coverage of Java-specific security patterns, and the generative tendency of large models to fabricate plausible explanations; for example, when the inputFilter() sanitization

routine was absent from the provided code slice, a state-of-the-art LLM falsely reported an XSS vulnerability on filtered input and invented a spurious exploitation rationale. To mitigate such risks, LARA employs refined code slicing, explicit evidence-chain binding, and multi-model cross-validation (reducing single-model hallucination misclassifications by up to 62%), combined with mandatory expert review for all high-risk or model-divergent conclusions and full decision-trace logging for each path to support auditing and framework refinement. Nevertheless, the framework still depends on scarce expert resources and remains vulnerable to novel or rapidly evolving vulnerability patterns.

Current research still faces three prominent challenges: First, the weak long-text generation capability of LLMs, while they can efficiently comprehend code contexts of up to 100,000 tokens, generating complete vulnerability analysis reports can easily result in logical breaks. Second, cross-model prompt transfer is difficult, which prompts fine-tuned for one model degrade significantly on other models, requiring additional adaptation costs. Third, controlling code security false-positive rates remains unresolved, or even small models optimized with instruction tuning cannot fully avoid misjudgments, necessitating human verification to compensate. Emerging paradigms like 'Reflection-Driven Control' are attempting to internalize this verification, embedding continuous self-reflection loops within the agent's reasoning process to enforce security policies and reduce the reliance on external human auditors [32].

Future research should proceed along three directions: First, focus on the collaborative optimization of long-context understanding and generation, and, by leveraging the structured path information from CPG, design a segmented generation with logical verification approach for long report generation to mitigate the weaknesses of LLMs in producing long texts; second, conduct targeted fine-tuning of open-source LLMs for code security, constructing specialized datasets for key tasks such as vulnerability detection and path analysis to reduce reliance on commercial models while enhancing the model's adaptability to code security scenarios; third, deepen the integration of CPG and prompt engineering, exploring a dynamic context selection mechanism based on the importance of CPG nodes to further compress redundant information and improve the reasoning efficiency and accuracy of LLMs. Future work should also explore lightweight, domain-specific small models as consistency checkers over evidence chains and dedicated hallucination-detection modules that automatically flag weakly supported path assessments, thereby further enhancing the robustness and trustworthiness of LARA in real-world security auditing pipelines.

Acknowledgement: Not applicable.

Funding Statement: The authors received no specific funding for this study.

Author Contributions: Conceptualization, Xiaorong Feng and Ying Gao; methodology, Ying Gao; software, Xiaorong Feng; validation, Xiaorong Feng, Ying Gao and Pengyi Du; formal analysis, Xiaorong Feng; investigation, Leyu Shi; data curation, Xiaorong Feng; writing—original draft preparation, Xiaorong Feng; writing—review and editing, Xiaorong Feng; visualization, Pengyi Du; supervision, Ying Gao. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: The data originates from externally controllable HTTP headers, which are concatenated directly into log messages without any sanitization or validation, and are recorded by Log4j. This is a typical Log4Shell (CVE-2021-44228) vulnerability pattern, allowing remote code execution through JNDI injection.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

Nomenclature

LLMs	Large language models
LARA	LLM-Augmented Reachability Analysis
CPGs	Code property graphs
MPCO	Meta-Prompt Code Optimization

References

1. Das Purba M, Ghosh A, Radford BJ, Chu B. Software vulnerability detection using large language models. In: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW); 2023 Oct 9–12; Florence, Italy. p. 112–9. doi:10.1109/ISSREW60843.2023.00058.
2. Michelena Á, Aveleira Mata J, Jove E, Alaiz Moretón H, Quintián H, Calvo Rolle JL. Development of an intelligent classifier model for denial of service attack detection. Int J Interact Multimed Artif Intell. 2023;8(3):33–42. doi:10.9781/ijimai.2023.08.003.
3. Herter P, Ahlrichs V, Açilan R, Horsch J. GPTrace: effective crash deduplication using LLM embeddings. arXiv:2512.01609. 2025.
4. Li Y, Wang S, Nguyen TN. 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. New York, NY, USA: ACM; 2021. p. 292–303. doi:10.1145/3468264.3468597.
5. Steenhoek B, Gao H, Le W. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. New York, NY, USA: ACM; 2024. p. 1–13. doi:10.1145/3597503.3623345.
6. Shrestha S, Mahmoud A. Mobile application review summarization using chain of density prompting. arXiv:2506.14192. 2025.
7. Cheng X, Zhang G, Wang H, Sui Y. 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. New York, NY, USA: ACM; 2022. p. 519–31. doi:10.1145/3533767.3534371.
8. Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: IEEE Symposium on Security and Privacy; 2014 May 18–21; San Jose, CA, USA. p. 590–604. doi:10.1109/sp.2014.44.
9. Wen XC, Gao C, Ye J, Li Y, Tian Z, Jia Y, et al. Meta-path based attentional graph learning model for vulnerability detection. IEEE Trans Softw Eng. 2024;50(3):360–75. doi:10.1109/TSE.2023.3340267.
10. Widmer DM, Mohaisen D. From large to mammoth: a comparative evaluation of large language models in vulnerability detection. In: Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS 2025); 2025 Feb 24–28; San Diego, CA, USA.
11. Lu G, Ju X, Chen X, Pei W, Cai Z. GRACE: empowering LLM-based software vulnerability detection with graph structure and in-context learning. J Syst Softw. 2024;212(6):112031. doi:10.1016/j.jss.2024.112031.
12. Nie Y, Li H, Guo C, Jiang R, Wang Z, Li B, et al. VulnLLM-R: specialized reasoning LLM with agent scaffold for vulnerability detection. arXiv:2512.07533. 2025.
13. Liu S, Lin G, Qu L, Zhang J, De Vel O, Montague P, et al. CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation. IEEE Trans Dependable Secure Comput. 2022;19(1):438–51. doi:10.1109/TDSC.2020.2984505.
14. Lemieux C, Inala JP, Lahiri SK, Sen S. CodaMosa: escaping coverage plateaus in test generation with pre-trained large language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE); 2023 May 14–20; Melbourne, Australia. p. 919–31. doi:10.1109/ICSE48619.2023.00085.
15. Xia CS, Wei Y, Zhang L. Automated program repair in the era of large pre-trained language models. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE); 2023 May 14–20; Melbourne, Australia. p. 1482–94. doi:10.1109/ICSE48619.2023.00129.
16. Zeng H, Bao A, Cheng J, Song C. PBFuzz: agentic directed fuzzing for PoV generation. arXiv:2512.04611. 2025.

17. Shi H, Liu Y, Pang B, Liu ZL, Wang H, Savarese S, et al. SSR: socratic self-refine for large language model reasoning. arXiv:2511.10621. 2025.
18. Qiu Y, Embar VR, Zhang Y, Jaitly N, Cohen SB, Han B. Eliciting in-context retrieval and reasoning for long-context large language models. In: Findings of the association for computational linguistics. Stroudsburg, PA, USA: ACL; 2025. p. 3176–92. doi:10.18653/v1/2025.findings-acl.165.
19. Li P, Yao S, Korich JS, Luo C, Yu J, Cao Y, et al. Automated static vulnerability detection via a holistic neuro-symbolic approach. arXiv:2504.16057. 2025.
20. Lekssays A, Mouhcine H, Tran K, Yu T, Khalil I. LLMxCPG: context-aware vulnerability detection through code property graph-guided large language models. arXiv:2507.16585. 2025.
21. Du X, Yu K, Wang C, Zou Y, Deng W, Ou Z, et al. Minimizing false positives in static bug detection via LLM-enhanced path feasibility analysis. arXiv:2506.10322. 2025.
22. Dong Y, Ruan CF, Cai Y, Lai R, Xu Z, Zhao Y, et al. XGrammar: flexible and efficient structured generation engine for large language models. arXiv:2411.15100. 2024.
23. Gong J, Giavrimis R, Brookes P, Voskanyan V, Wu F, Ashiga M, et al. Tuning LLM-based code optimization via meta-prompting: an industrial perspective. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE); 2025 Nov 16–20; Seoul, Republic of Korea. p. 3569–80. doi:10.1109/ase63991.2025.00295.
24. Munson A, Gomez J, Cárdenas ÁA. ♪ with a little help from my (LLM) friends: enhancing static analysis with LLMs to detect software vulnerabilities. In: 2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code); 2025 May 3; Ottawa, ON, Canada. p. 25–32. doi:10.1109/llm4code66737.2025.00008.
25. Nguyen V, Le T, Le T, Nguyen K, DeVel O, Montague P, et al. Deep domain adaptation for vulnerable code function identification. In: 2019 International Joint Conference on Neural Networks (IJCNN); 2019 Jul 14–19; Budapest, Hungary. p. 1–8. doi:10.1109/ijcnn.2019.8851923.
26. Zhang C, Liu B, Xin Y, Yao L. CPVD: cross project vulnerability detection based on graph attention network and domain adaptation. IEEE Trans Softw Eng. 2023;49(8):4152–68. doi:10.1109/TSE.2023.3285910.
27. Cao S, Sun X, Bo L, Wei Y, Li B. BGNN4VD: constructing bidirectional graph neural-network for vulnerability detection. Inf Softw Technol. 2021;136(1):106576. doi:10.1016/j.infsof.2021.106576.
28. Ciavatta JAS, Higuera JRB, Higuera JB, Montalvo JAS, Riera TS, Melero JP. Integration of large language models (LLMs) and static analysis for improving the efficacy of security vulnerability detection in source code. Comput Mater Contin. 2025;2025:1–10. doi:10.32604/cmc.2025.074566.
29. Li Z, Zou D, Xu S, Jin H, Zhu Y, Chen Z. SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Trans Dependable Secure Comput. 2022;19(4):2244–58. doi:10.1109/TDSC.2021.3051525.
30. Li Z, Zou D, Xu S, Chen Z, Zhu Y, Jin H. VulDeeLocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans Dependable Secure Comput. 2022;19(4):2821–37. doi:10.1109/TDSC.2021.3076142.
31. Mei L, Yao J, Ge Y, Wang Y, Bi B, Cai Y, et al. A survey of context engineering for large language models. arXiv:2507.13334. 2025.
32. Wang B, Quan J, Yu X, Hu H, Yuhao, Tsang I. Reflection-driven control for trustworthy code agents. arXiv:2512.21354. 2025.