CRITICAL: Collaborative Review Involving Thoughtful Intelligent Code Analysis using LLM

Chun-Kai Yang
b10901027@ntu.edu.tw
National Taiwan University
Taipei, Taiwan

Ping-Chieh Chou
b09902036@ntu.edu.tw
National Taiwan University
Taipei, Taiwan

Ting-Yu Yeh
b10901080@ntu.edu.tw
National Taiwan University
Taipei, Taiwan

Hung-Yu Liu
b10901173@ntu.edu.tw
National Taiwan University
Taipei, Taiwan

ABSTRACT

Open-source software is an essential part of modern software development, relying on the collaboration of numerous contributors to build this supply chain. However, supply chain attacks pose a severe issue. Adversaries can insert malicious code into commonly used software packages, careless developers may unintentionally introduce vulnerabilities, and tired reviewers may recklessly grant access to these vulnerabilities into production. This can lead to security problems for thousands of direct or indirect client applications.

To mitigate this attack, we proposed CRITICAL, a new approach utilizing the collaborative framework of the Large Language Model (LLM) to assist the code review process. Three main LLM agents allow us to process and comprehend source code efficiently. Upon evaluating several datasets and cross-referencing the outcomes with CodeQL's alerts and known vulnerabilities, our method effectively detects every component that could potentially be harmful or dangerous. Furthermore, in comparison to CodeQL, our method covers a wider range of concerns and is quicker and simpler to use. Demonstrating our approach has the potential to serve as a proactive warning system for review usage, aiding developers in identifying and mitigating potential security risks more efficiently. You can the source code of our method at https://github.com/YCK1130/CRITICAL.

1 INTRODUCTION

Modern software applications heavily depend on third-party open-source software. However, supply chain attacks have always been an important concern [3, 7, 11]. The recent security issues with XZ Utils highlight the vulnerability of open-source projects to supply chain attacks, emphasizing the need for robust security measures. Supply chain attacks occur when attackers compromise software by injecting malicious code into third-party libraries or tools, or when careless developers and tired reviewers unintentionally introduce vulnerabilities into open-source packages that software developers depend on.

To mitigate this attack, recent studies have developed various approaches to detect and mitigate malicious activities in software. Techniques like codeQL are used to identify malicious versions or flows in codebases with high accuracy and low false alarm rates, though they depend heavily on existing datasets [1, 4]. Rule-based systems have been proposed for detecting anomalous commits on platforms like GitHub [6]. Additionally, research on large language models (LLMs) for software vulnerability detection highlights their potential and limitations. While LLMs can recognize common vulnerable code patterns, they often perform worse than fine-tuned smaller models and are affected by the quality of prompts used [5, 9, 10, 13, 16]. Enhanced techniques like prompt engineering, self-reflection, and chain-of-thought show promise but struggle with complex tasks. Collaborative frameworks and multi-agent systems offer new directions for improving LLM performance in code review [2, 15]. These insights guide our efforts to enhance code review effectiveness using LLMs.

To resolve the difficulties from the previous research, we propose CRITICAL, a proactive approach for detecting malicious code injections or vulnerability exposure based on the new commit messages and codes. This approach utilizes multiple Large Language Model (LLM) agents' collaboration to identify anomalies and risky code segments. By this approach, we try to answer the following research questions:

(1) Is our approach able to detect malicious package updates?
(2) Can our method avoid being triggered by insignificant minor errors?
(3) Can our method be adapted to real-world scenarios?

2 PROBLEM DEFINITION

2.1 Attacker Model

We consider scenarios where harmful code may be intentionally or unintentionally introduced into an open-source project. These commits or pull requests typically require human approval. However, maintainers may lack the time or capacity to thoroughly review each commit, increasing the risk of overlooking malicious code. In our attacker model, we specify three characters that may introduce harmful code.

(1) **Adversaries** insert malicious code surreptitiously.
(2) **Careless developers** unintentionally introduce vulnerabilities while developing.
(3) **Tired reviewers** may recklessly grant access to these vulnerabilities into production.

2.2 Security Properties

Our system is designed to provide more accurate warnings to maintainers when malicious commits are detected. By pointing out issues, it reduces the chances of maintainers overlooking important details. This also decreases the time required for code reviews, allowing them to focus on more critical aspects of the project.

2.3 Assumptions

We assume that the submission form of each commit is in the source code. Consequently, our system does not address the detection of malicious binary files, but it will alert developers to pay special attention to the source and accuracy of these files.

3 APPROACH

We propose a novel collaborative framework utilizing Large Language Models (LLMs) to assist developers in preventing the accidental deployment of harmful or sensitive code into production environments, as illustrated in Figure 1. In this framework, the LLMs assume three distinct roles, Each role performs successive layers of examination through discussion, debate, and reflection to review a commit for potential issues meticulously.

3.1 Preprocessing

Firstly, our system conducts a separate analysis for each commit. During preprocessing, the differential data between commits stored in git is divided according to the file structure, with binary and other non-readable files being excluded from the analysis.

3.2 CRITICAL

Our proposed framework, CRITICAL, consists of three major roles:

(1) **Modification Analyst.** Labels and summarizes modifications of each file change.
(2) **Security Analyst.** Identifies potential vulnerabilities and discrepancies based on the file changes and the report of the Modification Analyst.
(3) **Quality Assurance Tester.** Evaluate the analysis and highlight the most critical issues.

Zhang et al. [14] have shown that introducing multi-agent debate combined with continuous reflection can yield the best result in difficult complex tasks. Inspired by human review strategies, our framework allows LLMs to first comprehend the purpose of each change, followed by an in-depth security analysis, and finally a review of the analyses for potential false alarms. By separating these roles, LLMs can concentrate on smaller, more specific tasks, and thereby enhancing efficiency. Due to practical and financial constraints, the LLMs engage in only a single round of debate before producing the final output. Each role also performs its respective analysis just once, which represents an area for potential improvement.

4 RESULTS

In section 4.1, 4.2, and 4.3, we introduce the dataset and metrics we used to evaluate on. We use CodeQL, a static code analysis tool, on unlabeled datasets as our benchmark. We report the performance of our framework in 4.4, compare to baselines in 4.5 and estimated cost to evaluate a commit in 4.6.

4.1 Experimental Setup

We utilized the gpt-4o-2024-05-13 model from OpenAI for our LLM framework. We evaluate our framework on self-build datasets, known npm flaws, and two real-world repositories.

(1) **Web application datasets.** This self-build dataset contains simple web applications such as simple blogs and database operations. Designed to be vulnerable to SQL injection and weak password protection.
(2) **Crypto-related datasets.** This self-build dataset contains crypto-related functions, including AES encryption with weak CBC operation mode, not recommended hash-then-encrypt HMAC.
(3) **Backstabbers-Knife-Collection** [8] provided a collection of known npm flaws, offering insight into prevalent security issues. Since it provides labeled information, we evaluate on a portion of this collection to justify the correctness of our generated report.
(4) **MakeNTU website.** This is a coursework of one of our group members, which has been deployed to serve the largest student-maker event in Taiwan, MakeNTU. This can be seen as a simple real-world scenario.
(5) **XZ Utils.** This open-source package suffered from CVE-2024-3094, which was discovered in late March 2024. The vulnerability occurred because malicious code was deliberately embedded into the software, creating a backdoor in versions 5.6.0 and 5.6.1.

These datasets contain tasks of varying difficulty levels, enabling a step-by-step analysis of our framework's performance across different scenarios. To validate the usability of our experiments, we introduce two real-world examples: one involving a straightforward student project and the other a rigorously maintained open-source project. We aim to uncover subtle traces of backdoors in XZ introduced by highly sophisticated techniques. This comprehensive evaluation allows us to thoroughly investigate and document security vulnerabilities in modern software systems.

4.2 Benchmark

Our evaluation is based on the documented flaws of npm packages or the alert provided by CodeQL defaulted queries.

For the Backstabbers-Knife-Collection, it provides ground truth about the actual flaws of each code snippet. Thus, we can compare it with our generated report to justify the correctness.

CodeQL is an analysis engine that developers use to automate security checks and security researchers to perform variant analysis. Its main purpose is to generate a database representation of a codebase, a CodeQL database. Once the database is ready, we can query it interactively or run a suite of queries to generate a set of results in SARIF format and upload the results to GitHub. We only utilize the default queries, which are the same as the code scanning on GitHub.

These benchmarks provide a solid foundation to evaluate potential vulnerabilities and allow us to compare our approach with tools commonly used in real-world scenarios.

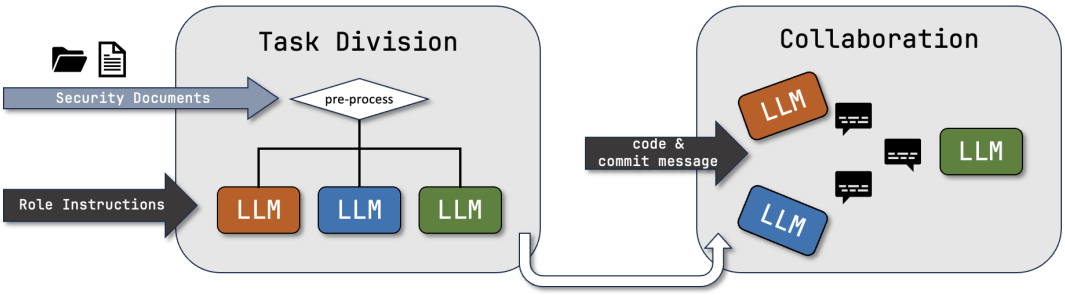


Figure 1: Collaborative Review Involving Thoughtful Intelligent Code Analysis using LLM (CRITICAL).

4.3 Evaluation Metrics

To measure the performance of our framework, we proposed three metrics inspired by Yu et al. [12] and adjusted the classification.

(1) **Completeness.** How the reports produced cover the existing vulnerability and abnormal behavior in the file.
(2) **Significance.** The report should accurately represent the truth and not create false impressions.
(3) **Non-Misleading.** The importance of problems that the reports point out to be most critical.

We manually inspected the content generated by our approach. For each metric, a score ranging from 1 to 5 was assigned. To minimize bias, the mean score was calculated after 4 independent evaluation runs. We reevaluated cases where our evaluations revealed significant discrepancies to ensure uniformity.

4.4 Evaluation

In figure 2a, we evaluate on 3 types of datasets (exclude real-world scenarios). The results show that our tool performs best on the web application dataset, which gets more than 4 points on three metrics and performs worst on the crypto-related dataset, which gets below 3 points on all metrics with high deviation. This might be due to the lack of relevant training data on the scope of cryptography since there is obviously more data about web development and application. On average, our tool performs best in finding vulnerabilities in different cases.

In figure 2b, we compare results on real-world scenarios, xz and MakeNTU website. Our tool has unstable performance on xz. Considering 3 metrics, it has better performance on MakeNTU website.

4.5 Comparison

In this section, we conduct an ablation study on the effect of collaborative LLMs and compare CRITICAL with the traditional static analysis method, CodeQL.

In figure 3, we compare CRITICAL with direct input to GPT-4o with a single run (direct). Our method performs slightly better on average. While the two methods show compatible results on completeness, our method points out more significant vulnerabilities and provides more non-misleading information about the commit. After diving into the reports the three roles produce, we found that the Tester, the last layer, catches most false alarms about security issues, clarifies the severity of each proposed vulnerability, and picks out the most significant one.

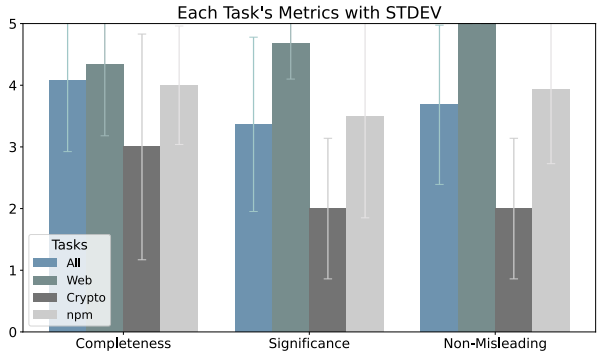
Furthermore, we organized the performance of our method compared with CodeQL into a table. Table 1 considers several prospects such as accuracy, coverage, usability, etc.

(1) **Accuracy.** The accuracy represents whether the problems pointed out exist or not.
(2) **Input form.** The input form describes what type of input the method takes. Our method takes a commit as input, and CodeQL needs to take the whole package as input.
(3) **Coverage.** Coverage describes the number and the categories of vulnerabilities discovered. Our method discovered more vulnerabilities and covered more general issues. CodeQL detects specific vulnerabilities based on the queries.
(4) **Granularity.** Granularity means how coarse and fine the method can point out about the location of the problem. While our solution can only point out a relatively high-hierarchical description to file level, CodeQL can point out which line in the file has the problem. Therefore, CodeQL has finer granularity than our work.
(5) **Usability.** Usability focuses on how user-friendly it is to use the method. For CodeQL, a complicated process for installation and operation is required to generate a code scan report. In contrast, our solution can run easily with only a few steps such as typing the name of the repository in the configuration file, and you are ready to go!
(6) **Running time.** As for running time, our method can run much faster than CodeQL since our method only takes the modification part of the code as input, while CodeQL needs to take the whole package, thus increasing the size of code to be analyzed.
(7) **Comprehensive.** Comprehensive describes whether the method can detect and provide comprehensive information about the whole malicious or vulnerable code.

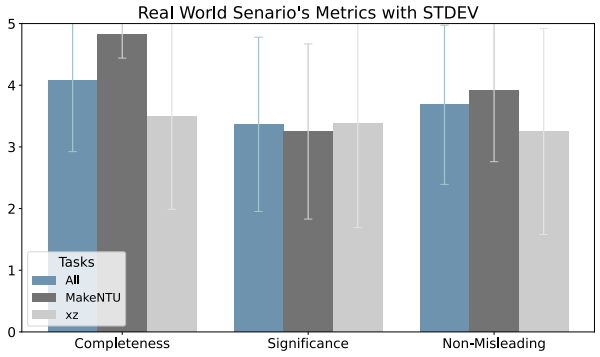
Our method has better coverage, usability, and running time, and CodeQL performs well on accuracy and granularity.

4.6 Cost Estimation

Using gpt-4o-2024-05-13, evaluation costs approximately 0.0467 $USD and approximately 30 seconds per commit.



(a) performance on different tasks



(b) performance on real-world scenarios

Figure 2: Evaluation of CRITICAL on different tasks

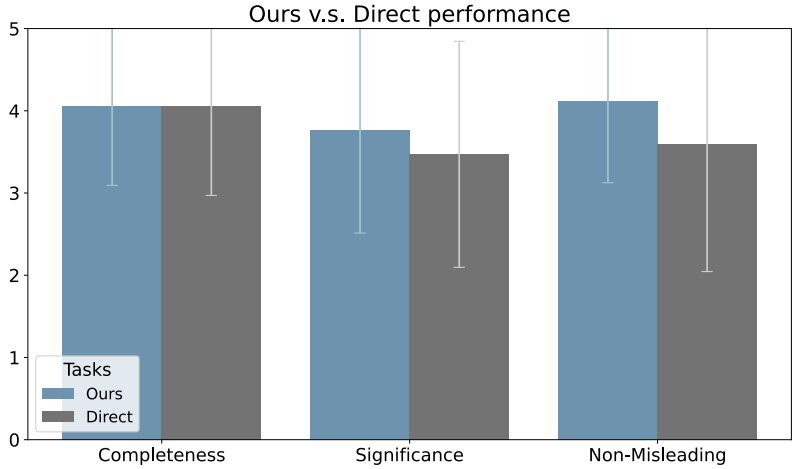


Figure 3: Performance comparison w/ and w/o collaborative discussion

Table 1: Comparison with CodeQL

	CodeQL	Our solution
Accuracy	High	Medium
Input form	whole package	percommit
Coverage	Medium	Large
Granularity	Fine	Coarse
Usability	Complicated	Intuitive
Running time	Longer	Shorter
Comprehensive	Not enough	Not enough

5 RELATED WORK

The related work presents various approaches to detect and mitigate malicious activities in software. The studies [4] and [1] focus both on identifying malicious versions or flows in codebases, utilizing codeQL and their work to achieve low false alarm rates and high accuracy/recall for known vulnerabilities. Their limitations lie in the dependency of existing data sets on possible failures. Furthermore, [6] introduces a rule-based system to automatically detect anomalous commits on GitHub. Furthermore, some studies have explored the capabilities of LLMs to detect software vulnerability. [13] demonstrate the effectiveness of prompt-engineering for vulnerability detection using ChatGPT. Furthermore, [10] [16] [5] show that LLMs have very low performance compared to fine-tuned smaller-scale models [9] but have the ability to recognize common vulnerable code patterns. They also emphasize the impact of various prompts on detection outcomes. Current works are limited by the ability of LLM to detect software vulnerability considering the lack of relevant training data. Some techniques have been used to enhance LLM performance via prompt engineering, including self-reflection and chain-of-thought. However, they may perform poorly on difficult and intricate tasks. [2] employs a self-collaboration framework in which multiple LLMs act as different experts, significantly improving code generation performance compared to direct generation. Another study [15] investigates collaboration among LLM agents from a social psychology perspective,

identifying optimal strategy of several rounds of agent debate and observing human-like social behaviors. These insights provide a direction for us to improve the current disappointing results of the code review with LLM.

6 FUTURE WORK AND CONCLUSION

6.1 Future work

Our proposed system leverages LLM to enhance the efficiency of code reviews, providing real-time warnings of potential vulnerabilities. While the system shows significant promise, there is room for improvement. Here are some feedback from class and some reflection of our limitations that we suggested for future work.

6.1.1 Use of Labeled Data for Testing. One of the critical issues in our current approach is the use of labeled data for testing the LLM. This introduces a potential bias, as the LLM might have previously encountered this data during its training phase. If the LLM has seen this data before, it could artificially inflate its performance metrics, giving a false impression of its effectiveness in real-world scenarios. To mitigate this, we should consider using completely novel datasets for testing, ensuring that the LLM has no prior exposure to the data.

6.1.2 Role-Based Improvements and Team Formation. The addition of specific roles—Modification Analyst, Security Analyst, and Quality Assurance Tester—has shown potential in improving the results of our LLM-based code review system. Each role focuses on distinct aspects of the code review process, enhancing the thoroughness and accuracy of the overall assessment. To capitalize on this improvement, we could try forming team for each role, i.e., setting more discussion within the same role, and see if it can perform better.

6.2 Conclusion

In our work, we have developed an LLM-based framework for code review aimed at enhancing the security of open-source projects. The system leverages the capabilities of multiple LLM agents, each playing a specific role in the code review process, to identify anomalies and risky code segments. By providing real-time warnings of potential vulnerabilities, our approach significantly improves the efficiency and accuracy of code reviews.

Our experiments demonstrated that the system can effectively identify security issues, offering higher accuracy and broader coverage compared to traditional tools like CodeQL. The role-based framework, where Modification Analysts, Security Analysts, and Quality Assurance Testers collaborate, has proven to be a robust method for ensuring thorough code analysis and review.

Despite the challenges associated with using labeled data and the need for more complex case analysis, our system shows great promise in automating and enhancing the code review process. The results indicate that integrating LLMs into the review workflow can reduce the burden on human reviewers, allowing them to focus on more critical tasks while maintaining high security standards.

Overall, our LLM-based code review system represents a significant step forward in securing open-source projects, offering a scalable and efficient solution to the challenges posed by manual code reviews.

REFERENCES

[1] Yiu Wai Chow, Max Schäfer, and Michael Pradel. 2023. Beware of the Unexpected: Bimodal Taint Analysis. arXiv:2301.10545 [cs.SE]

[2] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration Code Generation via ChatGPT. arXiv:2304.07590 [cs.SE]

[3] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139* (2020).

[4] Fabian Froh, Matías Gobbi, and Johannes Kinder. 2023. Differential Static Analysis for Detecting Malicious Updates to Open Source Packages. 41–49. https://doi.org/10.1145/3605770.3625211

[5] Michael Fu, Chakkrit Tantithamthavorn, Van Nguyen, and Trung Le. 2023. ChatGPT for Vulnerability Detection, Classification, and Repair: How Far Are We? arXiv:2310.09810 [cs.SE]

[6] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. arXiv:2103.03846 [cs.SE]

[7] Jeferson Martínez and Javier M Durán. 2021. Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. *International Journal of Safety and Security Engineering* 11, 5 (2021), 537–545.

[8] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.

[9] Cong Pan, Minyan Lu, and Biao Xu. 2021. An Empirical Study on Software Defect Prediction Using CodeBERT Model. *Applied Sciences* 11 (05 2021), 4793. https://doi.org/10.3390/app11114793

[10] M. Purba, A. Ghosh, B. J. Radford, and B. Chu. 2023. Software Vulnerability Detection using Large Language Models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE Computer Society, 112–119.

[11] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards using source code repositories to identify software supply chain attacks. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2093–2095.

[12] Jiaxin Yu, Peng Liang, Yujia Fu, Amjed Tahir, Mojtaba Shahin, Chong Wang, and Yangxiao Cai. 2024. Security Code Review by LLMs: A Deep Dive into Responses. arXiv:2401.16310 [cs.SE]

[13] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. 2024. Prompt-Enhanced Software Vulnerability Detection Using ChatGPT. arXiv:2308.12697 [cs.SE]

[14] Jintian Zhang, Xin Xu, Ningyu Zhang, Ruibo Liu, Bryan Hooi, and Shumin Deng. 2023. Exploring Collaboration Mechanisms for LLM Agents: A Social Psychology View. *CoRR* abs/2310.02124 (2023). https://doi.org/10.48550/ARXIV.2310.02124

[15] Jintian Zhang, Xin Xu, Ningyu Zhang, Ruibo Liu, Bryan Hooi, and Shumin Deng. 2024. Exploring Collaboration Mechanisms for LLM Agents: A Social Psychology View. arXiv:2310.02124 [cs.CL]

[16] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. arXiv:2401.15468 [cs.SE]