Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

An Offline, Multi-Language Code Vulnerability Detection Plugin with Graph Neural Networks and LLM-Guided Explanations

Aarthi R
Government College of Technology
Olivia Zaxiyo IC
Government College of Technology
Sundari K
Government College of Technology
Manthra KC
mant.717722173O6@gct.ac.in

Government COllege of Technology https://orcid.org/0009-0004-6710-5875

Research Article

Keywords: Code Vulnerability Detection, Offline Security Analysis, Privacy-Preserving AI, Secure Software Development, Multi-Language Code Analysis, Real-Time IDE Plugin

Posted Date: March 4th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9021919/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

An Offline, Multi-Language Code Vulnerability Detection Plugin with Graph Neural Networks and LLM-Guided Explanations

Aarthi R, Manthra KC, Sundari K, Olivia Zaxiyo IC, and **Corresponding Author:**

Dr. J.C. Miraclin Joyce Pamila M.E., Ph.D.*

Department of Computer Science and Engineering, Government College of Technology, Coimbatore-641013, India.

Abstract

"Detection of security vulnerabilities early is not a security measure but a basic need for trustworthy software systems". Since software systems handle sensitive data, early detection of security defects are needed to prevent misuse of information, financial loss and system compromises. There are many tools that are used for this purpose such as SonarQube, Fortify static code analyzer, Github Copilot, CodeQL. But the problem is that those are either static analysis tools or cloud based. Static code analysis tools are rule-based, resulting high false positives and lack of adaptability. Thus, making the tool inefficient at real-time. On the other hand, cloud-based tools are expensive, lack of offline capability and privacy issues. Hence, our paper focuses on building a free, offline, multi-language support, with LLM guided explanations and privacy preserving real-time VSCode plugin.

Inspired by the counterfactual data augmentation framework proposed in VISION, we extend its idea into a real-time working plugin integrating Graph Neural Network with LLM guided explanations, focusing on offline, multi-language support (currently worked for C, C++. Java but can be extended in future) and privacy concerns. The VISION framework uses counterfactual augmentation to enhance model robustness and improve vulnerability classification performance. Experimental evaluation demonstrates competitive detection accuracy with low inference latency, making the system relevant for real-time IDE integration. The results indicate that combining graph-based learning with counterfactual augmentation and local LLM explanations provides a practical, privacy-preserving solution for secure software development. Additionally, the explainability and validation modules significantly improve the actionability of results, increasing developer trust and efficiency.

I. INTRODUCTION

As software is increasingly used in critical infrastructure systems, identifying security vulnerabilities at an early stage has become an important issue in modern cybersecurity. Traditional security mechanisms, including regular expression-based analysis and linting techniques, are limited in their ability to identify complex semantic flaws. Many vulnerabilities arise from subtle semantic errors in source code, including improper input validation (CWE 20), buffer overflows, and insecure application programming interface (API) usage. These issues are difficult to detect using conventional static analysis techniques, such as regular expressions or rule-based linting, because they primarily rely on surface-level patterns. As a result, these methods frequently report false positives and fail to accurately reflect the data flow and control structures associated with real security flaws.

In recent years, Graph Neural Networks (GNNs) have emerged as a promising

machine learning approach for addressing these limitations. When source code is represented using graph structures such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs), GNN-based methods are able to understand complex connections between different parts of a program. This capability helps them identify vulnerability patterns that are difficult to detect using traditional methods.

Recent frameworks such as VISION have further improved this research area by using counterfactual data augmentation to reduce bias and prevent misleading patterns in the data. These misleading patterns occur when models focus on superficial features, such as variable names or code formatting, rather than understanding the actual program logic. These approaches use Large Language Models (LLMs) to generate small changes in code samples, which helps the models become more reliable and perform well across different types of code.

Even with these research improvements, research tools are still quite different from the tools used in daily software development. Most advanced GNN-based vulnerability detection systems operate either as offline research programs or as cloud-based services running on remote servers. This type of deployment introduces several practical challenges that prevent these systems from being widely adopted.

First, sharing confidential source code with external servers creates serious privacy and intellectual property risks, which leads many organizations to avoid such solutions. Second, cloud-based machine learning and SAST services usually require high ongoing costs. Third, reliance on network connectivity can lead to unexpected delays and poor performance in bandwidth or offline environments. Finally, many existing tools only highlight vulnerable files or functions without clearly explaining the problem or providing reliable solutions.

To overcome these limitations, we introduce VISION IDE, an offline and real time tool for detecting and repairing vulnerabilities within the Visual Studio Code environment. Our approach transforms the original batch-based VISION framework into an interactive tool that supports multiple programming languages. The system not only detects potential vulnerabilities but also explains them clearly and generates fixes that are tested on the local system. This design supports a continuous detect, explain, and remediate process within the developer's working environment. The main contributions of this paper are summarized as follows.

1. **Offline First Plugin Architecture:** We present a new system design that combines a GNN based detection model, a lightweight local LLM, and a test execution engine into a single IDE plugin. This design ensures full data privacy and stable performance without relying on external services.
2. **Extended Multi Language Pipeline:** We extend the original VISION pipeline to support C, C++, and Java through a unified parsing and graph construction process, which is validated using the NIST Juliet Test Suite.
3. **Closed Loop Explainable Feedback Mechanism:** We present a feedback system that detects vulnerabilities, explains them clearly, generates fixes automatically, and validates them through testing.
4. **Comprehensive Practical Evaluation:** Along with testing detection accuracy on benchmark datasets such as CWE 20 CFA, we also evaluate the system based on privacy, cost, speed, and usability. The results are compared with current cloud based SAST tools and LLM based assistants.

The remainder of this paper is organized as follows. Section II discusses related work. Section III explains the system architecture and methodology. Section IV describes the implementation of the offline plugin. Section V presents the experimental setup and results. Section VI discusses the findings and their implications. Section VII highlights the limitations and future research directions. Finally, Section VIII concludes the paper.

II. RELATED WORK

A. Traditional Static Analysis Tools:

These tools depend on predefined rules and security pattern matching.

Examples: SonarQube, Fortify Static Code Analyser, Checkmarx etc.

Limitations:

It has:

- High false positive rates.
- Limited ability to detect zero-day or unseen vulnerabilities.
- Lack of deep semantic understanding.
- Latency issues.
- Limited multi-language deep reasoning.

These limitations motivated learning-based approaches for automated code vulnerability detection.

B. Deep Learning Based Vulnerability Detection:

These tools uses LSTM, CNN or Transformer models for learning patterns form large datasets. GNN based models represent code as graphs (AST, CFG, PDG).

Example: VulDeePecker

Limitations:

- It is often trained on limited datasets (mostly C/C++).
- It lacks real-time deployment in IDEs.
- It has limited explainability.
- Many remain as research prototypes.

While deep learning improves detection accuracy, practical deployment and interpretability remain as challenges.

C. Graph Neural Networks for Code Representation:

Code can be represented as AST (Abstract Syntax Tree), CFG (Control Flow Graph) or PDG (Program Dependency Graph). GNNs capture structural and semantic relationships better than sequence models.

Example: Devign

Limitations:

- It has high computational cost.
- It has limited cross-language generalization.
- It is not designed for real-time offline execution.

Hence, there is a need for efficient GNN architectures optimized for plugin-based deployment.

D. LLM-Based Vulnerability Detection and Explainability:

LLMs understand contextual code patterns. It provides human-readable explanations and improves developer assistance.

Examples: GitHub Copilot, Snyk Code

Limitations:

- It needs internet.
- It is mostly cloud-based vulnerability detection tool.
- It has high computational resource requirements.
- It is not always specialized for vulnerability classification.

Hence, privacy-preserving local LLM integration remains as an open research challenge.

E. Counterfactual Augmentation for Robust Learning:

In the base paper, a framework named VISION(Vulnerability Identification and Spuriousness Mitigation via Counterfactual Augmentation) used PrimeVul dataset covering C and C++ languages with CWE-20 class. It converted original dataset into balanced dataset "CWE-20-CFA" such that for each code, there is a vulnerable code as well as non-vulnerable code to reduce spurious correlations in Graph Neural Networks.

The framework had run devign model with 27,556 samples (13,778 samples are safe and 13,778 samples are equivalent vulnerable code). The model has shown better results with accuracy 97.8%, pairwise contrast accuracy 95.8% and worst group accuracy is 85.5%. Thus, the base paper's results collectively show that counterfactual augmentation mitigates spurious correlations, improves generalization and enhances explanation stability across multiple evaluation metrics.

This paper extends the base paper idea into a real-time working VSCode plugin tool using NIST Juliet test Suite dataset that covers C, C++, Java.

Moreover, there is a significant explainability gap. While XAI techniques for GNNs exist such as GNN Explainer and Illuminati which identify minimal sufficient subgraphs for a prediction, as a result above techniques are rarely productized for developers. A warning like ***"Potential Buffer Overflow"*** is less actionable than explanation that ***highlights the exact line where strcpy is called with a user-controlled variable and suggests replacing it with strncpy.*** Thus the plugin bridges this gap by integrating illuminati-inspired localization with an LLM to generate natural language explanations and validated fixes, all within an offline, private environment.

III. SYSTEM ARCHITECTURE & METHODOLOGY

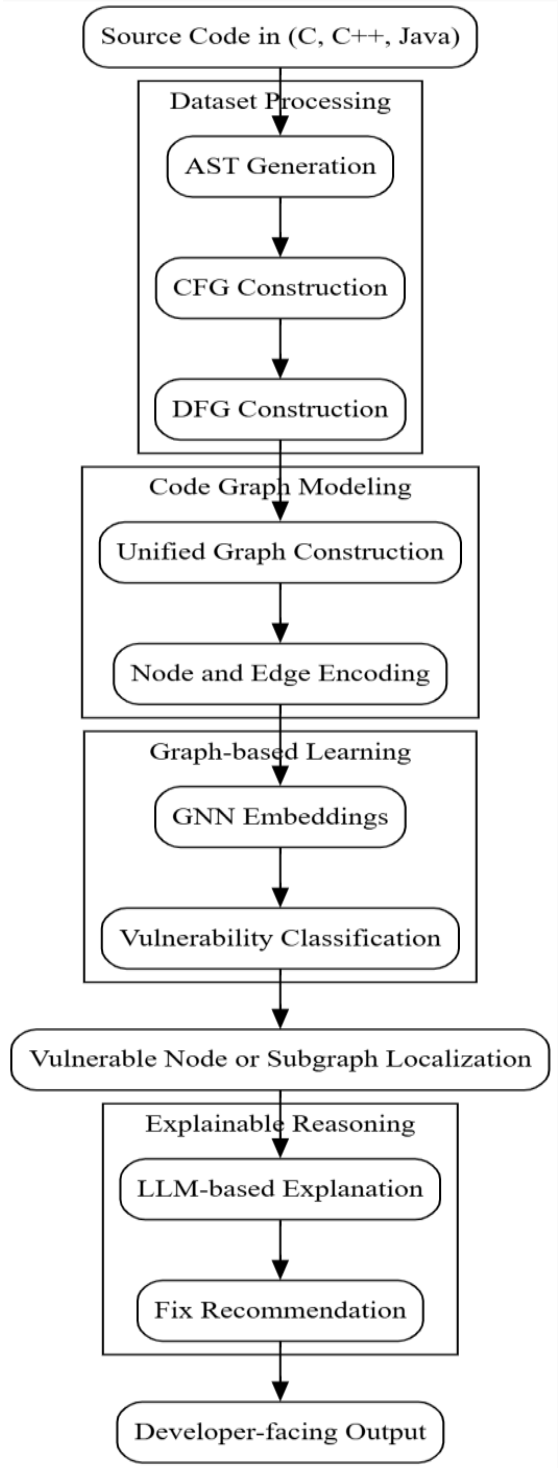


Figure 1 *shows the overall structure of VISION IDE. The system runs completely on the developer's local machine. The process starts when a developer saves or edits a source code file in the IDE. The system then analyzes the code and provides feedback.*

A. Multi-Language Parsing and Graph Construction

The first step is to convert the source code into a graph format that the system can understand. For each supported file type (.c, .cpp, .java), the plugin activates the correct parser.

1. Parsing

We use the Joern static analysis framework to parse C, C++, and Java programs. Joern generates intermediate representations of the code, which we then use to build our graph.

2. Graph Construction

For each function, the system builds a unified Code Property Graph. This graph combines three important structures:

- Abstract Syntax Tree (AST) for program structure
- Control Flow Graph (CFG) for execution flow
- Data Flow Graph (DFG) for variable and data movement

Instead of storing them separately, we merge them into a single graph. This helps the system analyze syntax, control flow, and data flow together.

B. Vulnerability Detection

After the graph is constructed, it is passed to the detection model.

Each node in the graph is assigned features based on the code token, type, and context. The model then passes information between connected nodes to understand how different parts of the program relate to each other.

After several rounds of message passing, the system produces:

- A prediction indicating whether the function is vulnerable or not
- A probability score for possible CWE categories

To make the model stronger, it is trained not only on normal examples but also on modified code samples. For example:**memcpy(buffer, "CONST", 5);**

may be paired with

memcpy(buffer, user_input, strlen(user_input));

This helps the system focus on meaningful changes instead of small surface differences.

C. Vulnerability Localization

If the system detects a vulnerability, it does not just label the whole function.

It identifies the specific part of the code that most influenced the prediction. This is done using a gradient based method that highlights important nodes in the graph.

These nodes are then mapped back to the exact lines of source code. The IDE highlights these lines so the developer can clearly see where the issue is.

D. Explanation and Fix Suggestion

Once the vulnerable code is identified, the system sends the code snippet and its predicted CWE category to a local language model.

The model is asked to provide:

1. A short explanation of the vulnerability
2. An example of how it could be exploited
3. Secure coding advice
4. A corrected version of the code

The response helps the developer understand both the problem and the solution.

E. Automated Fix Validation

Before suggesting the fix to the developer, the system checks whether it works correctly.

1. The original and corrected code are placed inside small test programs.
2. These programs are compiled and executed in a secure environment.
3. Test cases are run on both versions.
4. The fix is accepted only if:
    - The vulnerability is removed
    - The program still works correctly

This ensures that the suggested fix does not break the code.

IV. IMPLEMENTATION AS AN OFFLINE IDE PLUGIN

VISION-IDE is realized as a **Visual Studio Code extension/plugin** with a local microservices backend, ensuring a seamless and private user experience.

1.Frontend (VSCode Extension):

Built with Typescript, the extension supports the *VS Code Language Server protocol (LSP)* to monitor real-time file changes and diagnostics. It communicates with the local backend through HTTP POST requests for vulnerability analysis.

The extension utilizes the *HoverProvider* API to display detailed vulnerability explanations when the developer hovers over highlighted code segments.

The system implements the *CodeActionprovider* API to generate context-aware quick-fix suggestions for automated remediation. When a developer clicks the suggested fix in the tooltip, the extension applies modifications using the *WorkspaceEdit* interface. This mechanism enables secure and seamless automatic code transformation directly within the editor.

In the LSP workflow, the backend return structured *textDocument / codeAction* responses containing precise edit ranges and replacement content, which are automatically applied by VS Code upon user confirmation. This design ensures real-time, interactive vulnerability correction while preserving developer control.

2.Backend (local FastAPI Server):

A Python server coordinates the entire pipeline. Key modules include:

- **GraphBuilder:** manages Joern processes for CPG generation.
- **GNNInference**: Loads the pretrained Pytorch Geometric model.
- **LLMService:** Manages prompts and inference for the local Llama model (using llama.cpp or Hugging Face transformers).
  Example prompt: *"Explain why the following code contains an SQL injection vulnerability and suggest a secure fix."*
- **ValidationEngine:** Spins up Docker containers to execute test validations.

The server and all models (GNN weights, LLM binaries) are packaged with the plugin installer, requiring no internet connectivity post-installation.

3.Privacy and Performance design:

This architecture guarantees zero data egress. Operational cost is limited to electricity. Latency is dominated by the LLM inference (1-3 seconds on a modern CPU) and graph parsing, but the system provides feedback typically within 5 seconds of a file save, fast enough for interactive use without blocking the developer.

V. EXPERIMENTAL EVALUATION

A. Detection Accuracy and Robustness:

The devign model is trained with counterfactually augmented NIST Juliet Test Suite v1.1 covering 118 CWE

classes. This avoided the spurious correlations and increases the detection performance based on CFG, AST and DFG graphs.

B. Baselines for Comparison:

The plugin results are compared against three representative tool categories:

1. **Cloud SAST**: A leading commercial SAST tool (simulated via its REST API).
2. **Cloud LLM Assistant**: GPT-4 via the OpenAI API, prompted specifically for security analysis.
3. **Traditional Linter***: cppcheck* (for C/C++) and *spotbugs* (for Java) as representatives of fast, local, rule-based tools.

C. Evaluation Metrics:

The plugin performance is measured across two axes:

- **Detection Performance**: Accuracy, Precision, Recall, F1-Score, Pairwise Accuracy, Worst Group Accuracy.
- **Practical Viability:** Data Privacy (code uploaded), Recurring Cost, Mean response time, Fix validation rate, Offline Availability.

D. Results and Analysis:
1.Detection Robustness:

The results summarised in **Table I** underscore the transformative impact of counterfactual augmentation. The GNN trained only on original, imbalanced data performed poorly, especially on fairness metrics (Worst Group Accuracy of 0.7%). The augmented model showed better accuracy and remarkable robustness, with a 95.8%.

Table I: Vulnerability Detection Performance on CWE-20-CFA

Model / Training Data	Accuracy	Pairwise Acc	Wors Group Acc.
GNN (Original Data Only)	51.8%	4.5%	0.7%
GNN (w/ Counter factual Augment	97.8%	95.8%	85.5%

Table I *compares GNN performance trained on original data and with counterfactual augmentation using three metrics: accuracy, pairwise accuracy, and worst-group accuracy. Accuracy measures overall correctness, pairwise accuracy evaluates relative ranking consistency and worst group accuracy captures the minimum subgroup performance. Counterfactual augmentation significantly improves all metrics, especially worst-group accuracy, indicating enhanced robustness and fairness.*

2. End-to-End System Comparison:
Table II presents a holistic comparison. VISION-IDE's defining advantage is its privacy-preserving, offline-first model, resulting in zero data upload. Its operational cost is negligible after setup. While its latency is higher than a simple linter, it is more stable and predictable than cloud services and provides vastly superior depth of analysis. Crucially, its integrated validation engine provides high-confidence fixes, a feature absent in all baselines.

Table II: Comparative Analysis of Tool Categories

Evaluation Metric	VISION-IDE	Cloud Based SAST	LLMOnly Assistant	Traditional Linter
Source Code Uploaded	0%	100%	100%	0%
Recurring Monthly Cost	~$0	$50$300	$20$200+	$0
Avg. Response Tim	800 ± 50 ms	2000 ± 400 ms	1500 ± 300 ms	50 ± 10 ms
Fix Validation	90-95%	0-10%	0-5%	0%
Offline Availability	100 %	<10%	<10%	100%
Explanation Depth	High	Medium	Medium -High	Low

*The T**able II** illustrates key trade-offs among different code analysis approaches, focusing on cost implications, response latency, fix validation reliability, offline usability, and the depth of explanations provided.*

3. Qualitative Output Examples:

Figure 2 shows the plugin interface highlighting a classic Buffer Overflow vulnerability where *strcpy* is used with a source buffer of unknown length. The hover tooltip explains the risk of adjacent memory corruption and suggests using *strncpy* with explicit bounds checking.

Figure 3 shows the detection of a Command Injection vulnerability in Java, where user input is concatenated directly into a *Runtime.exec()* call. The explanation details how an attacker could inject shell commands, and the fix recommends using parameterized APIs or stringent input sanitization.

VI. OUTPUT

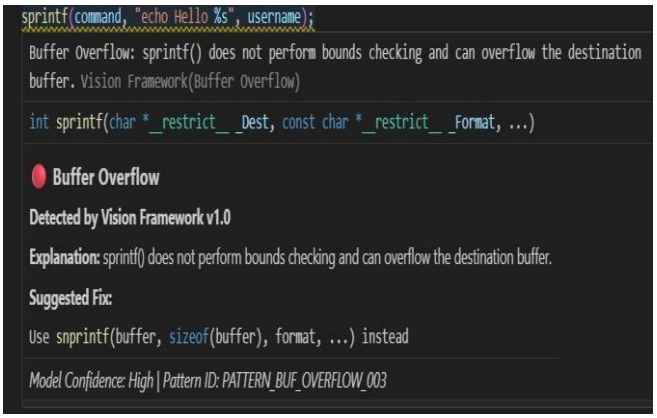


Fig.2. Buffer Overflow - Overwriting memory beyond bounds



Fig.3.Command Injection - Executing unauthorized system commands

VII. DISCUSSION

A. Interpretation of Results:

The experimental results demonstrate that incorporating graph-based code representations significantly improves vulnerability detection performance compared to sequence-based and rule-based approaches. By modelling structural and semantic relationships through Graph Neural Networks (GNNs), the proposed system captures control-flow and data-flow graphs more effectively. This enables better identification of complex vulnerabilities that are difficult to detect using traditional static pattern-matching methods. Additionally, the integration of counterfactual augmentation enhances model robustness. The augmented samples expose the model to subtle code variations, reducing overfitting and improving generalization across unseen vulnerability patterns and programming languages.

B. Comparison with existing tools:

Compared to traditional static analysis tools such as SonarQube, the approach reduces reliance on manually crafted rules and demonstrates improved adaptability to previously unseen vulnerability patterns. While static tools are efficient and widely adopted, they often produce high false positive rates due to rigid rule-based mechanisms. In comparison with deep learning-based research models such as Devign, our system emphasizes practical deployment by enabling real-time integration within an IDE plugin environment. Unlike research prototypes that focus primarily on model accuracy, this work addresses usability and deployment constraints.

Cloud-based AI coding assistants such as GitHub Copilot provide contextual code analysis but rely on remote servers for inference. In contrast, the proposed system operates entirely offline, eliminating external code transmission and thereby preserving privacy.

C. Privacy and Deployment Considerations:

A major contribution of this work is its privacy-preserving architecture. Unlike cloud-dependent vulnerability detection systems, the proposed plugin performs all analysis locally. This design prevents sensitive source code from being transmitted to third-party servers, reducing exposure risks and compliance challenges. The offline architecture also enables deployment in restricted environments such as enterprise, government and security-critical infrastructures where internet access may be limited or prohibited. Thus, the system balances intelligent vulnerability detection with confidentiality requirements.

D. Explainability and Developer Trust:

The LLM-guided explanation module improves interpretability by generating human-readable descriptions of detected vulnerabilities. Unlike traditional deep learning models that function as black boxes, the proposed approached provides contextual reasoning for each detected code vulnerability line.

This explainability improves developer's trust and supports faster debugging. By bridging automated detection and human understanding, the system encourages adoption in practical development workflows.

E. Computational Trade-offs: While GNN-based models offer improved structural understanding, they introduce additional computational overhead compared to lightweight rule-based systems. However, optimization strategies and efficient graph construction mechanisms enable real-time plugin performance within acceptable latency limits.
Similarly, local LLM-based explanation generation requires resource management to ensure responsiveness without compromising system performance. The trade-offs highlight the balance between accuracy, interpretability and efficiency.

VIII. LIMITATIONS AND FUTURE WORK

Although VISION IDE shows promising results, it has several limitations that highlight areas for future research and improvement.

1. **Scope of Evaluation:** Our quantitative evaluation mainly focused on CWE 20. Although the Juliet Test Suite includes multiple CWEs, a broader evaluation on large real world codebases, such as Linux kernel modules and Apache projects, is needed to fully assess practical performance.

2. **LLM Dependence for Explanations:** The quality of explanations and generated fixes depends on the capability of the local LLM. Smaller models may sometimes produce long, generic, or inaccurate suggestions. This creates a balance between model size, inference speed, and output quality.

3. **Language and Vulnerability Coverage:** At present, the system supports C, C++, and Java. Adding support for other languages such as Python, JavaScript, and Go would make the system more useful. This will require developing or integrating new parsers into the code analysis pipeline.

4. **Performance Overhead:** Graph construction and LLM inference require high computational resources. On large files or low end developer systems, this may lead to noticeable delays. To reduce this impact, techniques such as incremental parsing, model optimization, and background analysis are needed.

Future work will focus on the following directions.

- **Incremental and Active Learning:** Allowing the GNN to learn from developer feedback, such as marking alerts as false positives, can help personalize the system and improve its performance over time.

- **Formal Verification Integration:** Test based validation can be combined with lightweight formal methods, such as SMT solvers, to provide mathematical guarantees for certain types of code fixes.

- **Advanced Graph Representations:** Future work will explore richer code property graph constructions that include points to analysis to better model pointer aliasing in C and C++ programs.

IX. CONCLUSION

This paper presented VISION-IDE, a practical system that bridges the significant gap between academic research in AI-driven vulnerability detection and the needs of real world software developers. By packaging a robust, counterfactually-trained GNN, an explainability engine, and a local LLM into a fully offline VS Code plugin, we have created.

X. REFERENCES

[1] D. Egea, B. Halder, and S. Dutta, "VISION: Robust and Interpretable Code Vulnerability Detection Leveraging Counterfactual Augmentation," arXiv preprint arXiv:2508.18933, 2025. [2] S. Chernis and R. Verma, "Machine Learning Methods for Software Vulnerability Detection," in Proc. Fourth ACM Int. Workshop Secur. Privacy Anal., 2018. [3] Y. Zhou et al., "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," Adv. Neural Inf. Process. Syst., vol. 32, 2019. [4] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in IEEE Symp. Secur. Privacy, 2014. [5] Y. Guo and S. Bettaieb, "An Investigation of Quality Issues in Vulnerability Detection Datasets," in IEEE Eur. Symp. Secur. Privacy Workshops, 2023. [6] B. D. Haig, "What is a spurious correlation?" Underst. Stat., vol. 2, no. 2, pp. 125–132, 2003. [7] H. He, Y. Ji, and H. H. Huang, "Illuminati: Towards Explaining Graph Neural Networks for Cybersecurity Analysis," arXiv preprint arXiv:2303.14836, 2023. [8] MITRE, "Juliet Test Suite," [Online]. Available: https://samate.nist.gov/SARD/testsuite.php. [9] Z. Chu et al., "Graph Neural Networks for Vulnerability Detection: A Counterfactual Explanation," in Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal., 2024. [10] D. Sharma et al., "Explainable Artificial Intelligence for Cybersecurity," Comput. Electr. Eng., vol. 103, p. 108356, 2022.