Journal of Theoretical and Applied Information Technology
15th January 2025. Vol.103. No.1
© Little Lion Scientific

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

AN APPROACH FOR VULNERABILITY DETECTION IN WEB APPLICATIONS USING GRAPH NEURAL NETWORKS AND TRANSFORMERS

MOHAMMED YAHAYA TANKO [1], ABU BAKAR MD SULTAN [2], MOHD HAFEEZ OSMAN [3], HAZURA ZULZALIL[4]

[*]Department of Software Engineering and Information System, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, 43400 UPM Serdang, Selangor Malaysia

E-mail: [1]tymohammed45@gmail.com, [2]abakar@upm.edu.my, [3]hafeez@upm.edu.my, [4]hazura@upm.edu.my

ABSTRACT

The increasing complexity of software systems and rising security concerns due to open-source package vulnerabilities have made software vulnerability detection a critical priority. Traditional vulnerability detection methods, including static, dynamic, and hybrid approaches, often struggle with high false-positive rates and limited efficiency. Recently, graph-based neural networks (GNNs) and Transformer models have shown potential in improving vulnerability detection accuracy by representing code as graphs that capture syntax and semantics. This paper introduces a hybrid framework combining a Gated Graph Neural Network (GGNN) and Transformer encoder to leverage multiple graph representations: Abstract Syntax Tree (AST), Data Flow Graph (DFG), Control Flow Graph (CFG), and Code Property Graph (CPG). The GGNN extracts graph-level features, while the Transformer enhances sequential context understanding within the graph-encoded data. The model uses these capabilities to detect vulnerabilities in function-level code snippets. Evaluation of our framework on the OWASP WebGoat dataset demonstrates the effectiveness of different graph representations across five major vulnerability types: command injection, weak cryptography, path traversal, SQL injection, and cross-site scripting. Experimental results show that the GGNN+CPG configuration consistently yields high recall for cryptographic weaknesses, while GGNN+CFG excels in detecting control-based vulnerabilities, such as command injections. The integration of GGNN and Transformer models leads to notable enhancements in accuracy, precision, recall, and F1-score across all vulnerability types, with each graph representation contributing unique insights into code structures and vulnerability patterns. These findings highlight the potential of hybrid GNN-Transformer frameworks in enhancing code vulnerability detection for cybersecurity applications.

Keywords: *Vulnerability Detection; Graph Neural Networks; Software Security, Web Application Security, Transformers In Cybersecurity*

1. INTRODUCTION

Software vulnerabilities are weaknesses within software systems that malicious actors can exploit to undermine data integrity, confidentiality, and availability. Manually detecting these vulnerabilities in source code is a labor-intensive and error-prone task due to the complexity of modern software systems and their reliance on diverse code structures and numerous third-party libraries.

As the number of identified software vulnerabilities grows, detecting these flaws has become a critical challenge for the software industry and cybersecurity field [1].

Recent studies have shown that attackers can leverage open-source package managers to spread malware, raising significant security risks for both developers and users, along with substantial financial and social repercussions [1].

A key preventive measure involves scanning source code with vulnerability detection tools before releasing a program. In practice, developers and security professionals mainly rely on code analysis or testing techniques to detect and fix bugs. These methods fall into three categories: static, dynamic, and hybrid. Static techniques, such as rule-based and symbolic analysis, analyze code without executing it. These systems frequently produce a significant number of false positives, leaving engineers with the arduous task of verifying false alarms. Fuzz testing is the most often used dynamic approach right now. While effective at detecting zero-day vulnerabilities, it has limitations in analytical efficiency and code coverage. Hybrid techniques aim to combine static and dynamic analysis to offset their respective drawbacks, yet they remain impractical for widespread use [1], [2].

To overcome the limitations of traditional code analysis methods, significant advancements have been made using machine learning (ML) and deep learning (DL) techniques to identify code vulnerabilities. However, conventional machine learning approaches still struggle to address the full range of vulnerabilities effectively [1].

To address these limitations, numerous deep learning approaches for code vulnerability detection have emerged, aiming to develop robust neural network models that thoroughly capture code semantics and recognize vulnerability patterns [1].

The primary objectives of this research are to develop a hybrid model that leverages Gated Graph Convolutional Networks (GGCNs) to process structural relationships within code graphs and Transformers to capture contextual dependencies in source code, enhancing vulnerability detection. This model aims to accurately identify security vulnerabilities, particularly SQL injections, with high precision and recall while minimizing false positives through the integration of graph-based learning and contextual pattern recognition.

2. RELATED WORKS

The adoption of graph-based neural networks for vulnerability detection in code analysis has accelerated over recent years, leveraging various advanced architectures and methodologies to capture syntactic and semantic program information. Wang et al. [3] transform program structures into directed graphs, where statements, identifiers, and values are nodes with relational edges. Graphs are represented in matrices and processed by Graph Neural Networks (GNN) for embedding and prediction, emphasizing node connectivity. Similarly, Lin et al. [4] introduce VulEye, which uses Sub-Dependence Graphs (SDG) and Doc2Vec embeddings to enhance PHP code vulnerability detection, utilizing multilayer perceptron (MLP) for classification.

Several works emphasize deep semantic analysis combined with advanced network architectures. Guo et al. [5] use a gated GNN with multi-head attention to capture semantic and structural code properties, integrating a BILSTM for contextual information and taint analysis. Cao et al. [6] apply Bidirectional GNNs (BGNN) and convolutional neural networks to control flow graphs (CFGs) for feature extraction, while Wang et al. [7] in DeepVulSeeker employ a multi-stage approach that integrates Data Flow Graphs (DFG), Control Flow Graphs (CFG), and a self-attention encoder for vulnerability prediction. Zhou et al. [8] further elaborate this approach with a composite graph embedding layer, gated recurrent layers, and a convolutional module for enhanced representation learning.

Tensor-based and meta-path approaches further augment vulnerability analysis. Fan et al. [9] propose CircleGGNN, a tensor-based GNN extension that combines multiple graphs, while Wen et al. [10] in AMPLE focus on edge-aware graph representations, enhancing node features through kernel-scaled convolutions. Similarly, Li et al. [11] utilize the IVDetect model, which combines graph-based vulnerability detection with interpretative graph-based modules for detailed code vulnerability scores.

Simplifying and refining graph structures has also proven effective. Zhuang et al. [12] introduce a streamlined 3GNN model that classifies source code based on vectorized graph representations, while Cheng et al. [13] in DeepWukong leverage interprocedural control and data-dependence graphs with forward and backward traversal for precise vulnerability identification. Wen et al. [14] propose a meta-path hierarchical attentional graph neural network to efficiently process heterogeneous graph relations.

Machine learning models are also increasingly adapted to leverage pre-trained code embeddings and specialized slicing methods. Suneja et al. [15] utilize code property graphs (CPG) encoded with Word2Vec for bug signature learning. Li et al. [16] in SySeVR employ semantic and syntactic vulnerability components to improve vulnerability vector encoding. Other works such as VulDeePecker [17] and VulSniper [18] use BLSTM and attention models on code property graphs to further address vulnerabilities, illustrating a trend towards integrating linguistic embedding models with graph-based frameworks.

Collectively, these studies highlight the evolution from basic code embeddings to sophisticated multi-layered GNNs, incorporating techniques such as meta-path construction, bidirectional learning, and tensor-based enhancements. The continued advancements in model architecture and preprocessing methods underscore the field's commitment to refining vulnerability detection through increasingly detailed and adaptable graph representations.

3. METHODOLOGY

3.1 Problem Definition

The vulnerability detection task is formulated as a binary classification problem. Given a code snippet represented by a CPG, the model classifies it as either containing a vulnerability (e.g., SQL injection) or being secure.

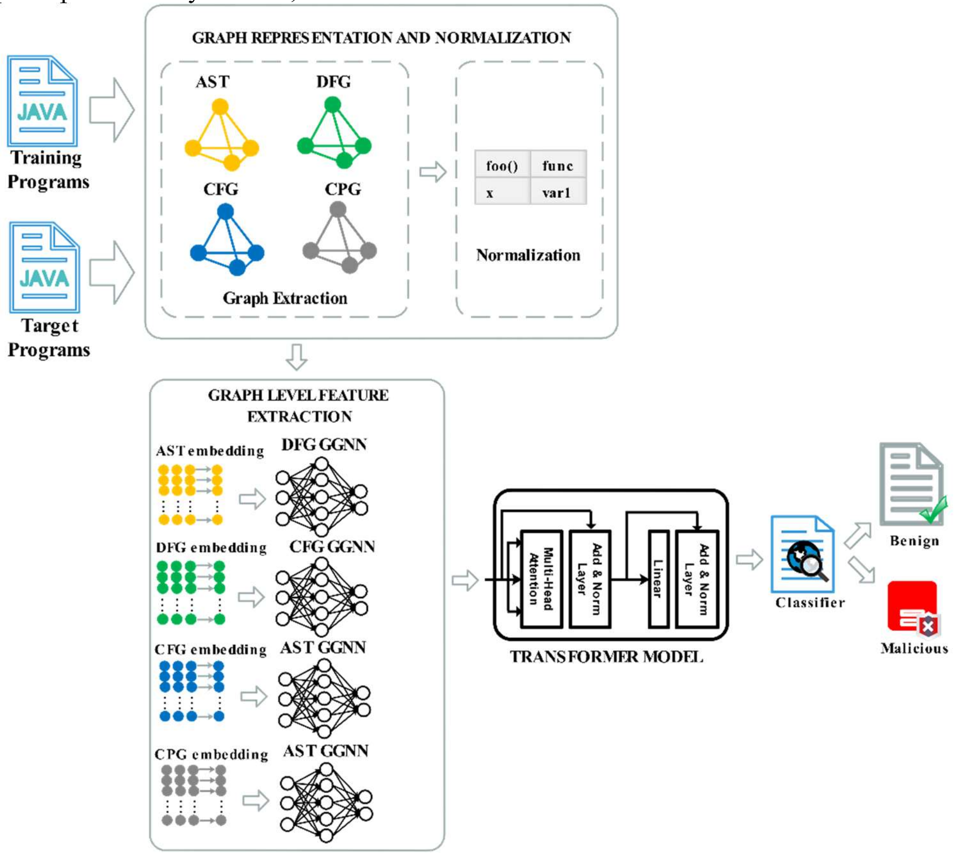

Figure 1: Proposed Framework

3.2 Framework

Figure 1 illustrates our GNN-based approach, which consists of three main components: graph representation and normalization, graph-level feature extraction, and vulnerability prediction. This approach currently supports function-level vulnerability detection. For each program input, we represent the entire class using four types of graphs: Abstract Syntax Tree (AST), Data Flow Graph (DFG), Control Flow Graph (CFG), and Code Property Graph (CPG), which integrates the first three. We use Joern, a powerful code analysis platform to generate these graphs. Afterward, we encode the nodes within the graph, creating the foundation for the learning phase that follows. We then apply a GNN-based learning approach to train a model for vulnerability prediction using this vectorized data.

The proposed model consists of the following components:

(a) Gated Graph Neural Network (GGNN): This module is used to process graph-based representations of program code, where nodes represent program elements, and edges capture relationships between them. The GGNN outputs an enhanced feature representation of each node in the graph, incorporating information from its neighborhood.

(b) Transformer Encoder: A multi-layer Transformer encoder is employed to process sequential data extracted from the graph representation. The encoder uses self-attention mechanisms to model long-range dependencies between nodes in the graph.

(c) **Convolutional Neural Network:** This module combines the graph-based features from the GGNN and the sequential features from the Transformer encoder. Two 1D convolutional layers with batch normalization and max-pooling operations extract hierarchical patterns from the data. Fully connected layers further process the features, with dropout applied to prevent overfitting.

(d) **Final Classification Layer:** A sigmoid activation function is used at the output to predict the presence or absence of a vulnerability in the input.

3.2.1 Graph Representation and Normalization

We examine four graph representations: Abstract Syntax Tree (AST), Data Flow Graph (DFG), Control Flow Graph (CFG), and Code Property Graph (CPG). The AST provides a hierarchical view of the program's syntax, while the DFG tracks variable declarations and usage throughout the code. The CFG captures the program's data and control flows. The CPG integrates AST, DFG, CFG, and Call Graph (CG) elements into one comprehensive graph.

3.2.2 Graph Level Feature Extraction

To support the subsequent model training with the GNN learning approach, an initial encoding of each node in the graphs is required. There are various approaches for such encoding. In this study, we use the Word2Vec [2]. Word2Vec employs the standard word embedding technique to train the model, utilizing node type sequences created by traversing the graphs in a specified order.

3.2.3 Vulnerability Prediction

This study introduces a neural network architecture integrating a **Gated Graph Neural Network (GGNN)** with a **Transformer-based convolutional network** to detect vulnerabilities in web applications. The model leverages graph-based data representations to capture complex dependencies between program elements while utilizing a convolutional architecture to extract meaningful features for classification. To predict the vulnerabilities within a class, we feed the graph encoding of the class into the respective prediction model for each vulnerability.

4. EXPERIMENTS

4.1 Dataset

The model was tested using the OWASP WebGoat dataset [19], a recognized benchmark that includes Java-based applications with labeled vulnerabilities, such as SQL injection flaws. Version 1.1 of this dataset was chosen due to its greater number of test cases compared to version 1.2. Five of the top ten vulnerabilities from the OWASP benchmark were chosen. The dataset was divided into training (70%), validation (15%), and testing (15%) subsets. Table 1 outlines the selected vulnerabilities and the number of test cases associated with each.

Table 1: Description of Dataset

Vulnerability Area	Number of Test Cases	CWE Number
Command Injection	2708	78
Weak Cryptography	1440	327
Path Traversal	2630	22
SQL Injection	3529	89
XSS (Cross-Site Scripting)	3449	79

4.2 Evaluation Metrics

To comprehensively evaluate our model's performance, we use commonly applied metrics: Accuracy, Precision, Recall, and F1-score. These metrics are calculated based on a confusion matrix that contains True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

Accuracy: The overall performance of the model in correctly identifying vulnerable and non-vulnerable code snippets. It is computed as follows:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

Precision: The proportion of true positives among all predicted positives, reflecting the model's capability to minimize false positives. It is computed as follows:

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

Recall: The ratio of true positives to all actual positives, measuring the model's detection power. It is computed as follows:

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

F1-Score: The weighted average of precision and recall, calculated as their harmonic mean. It is determined by the following formula:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (4)$$

4.3 Experimental Setup

Table 2 provides an overview of the hardware and software environments used.

Table 2: Experimental Setup

Hardware	**CPU**	Intel Core i5
	GPU	V100
	Memory	32 GB
	Disk	2 TB
Software	**OS**	Windows 11
	Python	3.9.18
	Pandas	2.1.3
	Pytorch	2.10
	Scikit-learn	1.3.0
	torch_geometric	2.4.0
	joern	2.0.201

In machine learning models, it is optimal to achieve high values for both Precision and Recall. However, these metrics often exhibit a trade-off, where improving one can decrease the other. Furthermore, Accuracy can be affected by an imbalanced sample distribution. To overcome these issues, the F1-measure was introduced [9].

Our goal is to reduce both false positive and false negative rates in the model's detection as much as possible. To accomplish this, we focus on creating a model with high precision and recall. Among the four evaluation metrics, precision, recall, and F1-score are the most crucial for evaluating the model's performance.

4.4 Result and Discussion

The results shown in tables 3, 4 and 5 reveal insights into the model's performance when applied to different vulnerability types using various graph representations (AST, CFG, DFG, and CPG) alongside a Gated Graph Neural Network (GGNN). Each graph representation captures unique aspects of the code's structure and behavior, impacting the

Table 3: Performance of Different Methods on CWE-78 and CWE-327

Method	**Accuracy**		**Precision**		**Recall**		**F1-Measure**	
	CWE-78	**CWE-327**	**CWE-78**	**CWE-327**	**CWE-78**	**CWE-327**	**CWE-78**	**CWE-327**
GGNN+AST	60.80%	50.00%	63.37%	50.00%	51.20%	100.0%	56.64%	66.67%
GGNN+CFG	74.07%	50.00%	79.59%	50.00%	75.58%	100.0%	77.53%	66.67%
GGNN+DFG	69.91%	66.67%	69.39%	66.33%	66.10%	62.50%	67.71%	64.36%
GGNN+CPG	64.80%	50.00%	66.09%	50.00%	60.80%	100.0%	63.33%	66.67%

Table 4: Performance of Different Methods on CWE-22 and CWE-89

Method	**Accuracy**		**Precision**		**Recall**		**F1-Measure**	
	CWE-22	**CWE-89**	**CWE-22**	**CWE-89**	**CWE-22**	**CWE-89**	**CWE-22**	**CWE-89**
GGNN+AST	68.66%	66.40%	64.81%	50.00%	70.00%	50.40%	67.31%	66.67%
GGNN+CFG	75.28%	53.33%	73.56%	51.85%	75.28%	40.00%	73.56%	50.00%
GGNN+DFG	70.37%	72..22%	73.47%	70.41%	65.45%	69.00%	69.23%	69.70%
GGNN+CPG	67.74%	66.40%	100.0%	63.22%	35.19%	78.40%	52.05%	70.00%

Table 5: Performance of Different Methods on CWE-79

Method	**Accuracy**	**Precision**	**Recall**	**F1-Measure**
	CWE-79	**CWE-79**	**CWE-79**	**CWE-79**
GGNN+AST	76.67%	64.10%	60.00%	69.80%

GGNN+CFG	80.00%	61.32%	75.28%	73.56%
GGNN+DFG	78.42%	60.67%	90.00%	80.22%
GGNN+CPG	82.80%	74.40%	100.0%	85.32%

4.4.1 CWE-78 and CWE-327

For CWE-78 (Command Injection) and CWE-327 (Broken Weak Cryptography), the GGNN with CFG (Control Flow Graph) yielded the highest accuracy for CWE-78 (74.07%) and matched other models for CWE-327 (50.00%) as shown in Table 3 and Figure 2. CFG provided strong results in recall and F1-measure for CWE-78, suggesting that control flow structure is valuable for detecting Command Injections, which often involve sequences of sensitive operations.

- **AST Performance:** The Abstract Syntax Tree (AST) representation, focusing on the syntax of the code, achieved the lowest accuracy and recall for CWE-78. However, it performed notably well in CWE-327, attaining 100% recall, which may indicate that syntactic patterns alone can be highly indicative for certain cryptographic vulnerabilities.
- **CFG and DFG:** CFG's control flow capture appears advantageous for CWE-78, likely due to the sequential and conditional logic associated with command injections. The DFG (Data Flow Graph) scored moderately well across all metrics, achieving balanced performance, particularly for CWE-327, which involves tracking data dependencies that are critical in cryptographic operations.
- **CPG:** The Code Property Graph (CPG), which combines AST, CFG, and DFG elements, displayed a mixed performance, suggesting that while it can capture extensive information, this complexity may not always benefit vulnerability detection directly. Nevertheless, CPG's ability to reach 100% recall in CWE-327 indicates its potential strength in detecting cryptographic issues when high recall is prioritized.

4.4.2 CWE-22 and CWE-89

For CWE-22 (Path Traversal) and CWE-89 (SQL Injection), GGNN with CFG again demonstrated strong performance for CWE-22, achieving 75.28% accuracy and a high F1-measure (73.56%), suggesting that control flow is crucial in path traversal vulnerability detection. In contrast, the DFG representation achieved the highest accuracy for CWE-89 (72.22%), pointing to the importance of data flow in identifying SQL injections, where data dependencies heavily influence vulnerability manifestation.

- **AST**: GGNN combined with AST achieved satisfactory results for CWE-22, with 68.66% accuracy, although it fell short for CWE-89, possibly due to the limitations of syntactic analysis alone in capturing complex interactions of user input and database operations, as required in SQL injection detection.
- **DFG**: GGNN+DFG's high performance for CWE-89 demonstrates that the data flow relationships are pivotal in detecting SQL injection vulnerabilities.
- **CPG**: In CWE-22, CPG attained 100% precision, though its recall was limited. This discrepancy suggests that while CPG is effective in identifying path traversal vulnerabilities precisely, it may not cover all instances exhaustively. In CWE-89, CPG provided relatively balanced performance, achieving a high F1-measure of 70.00%.

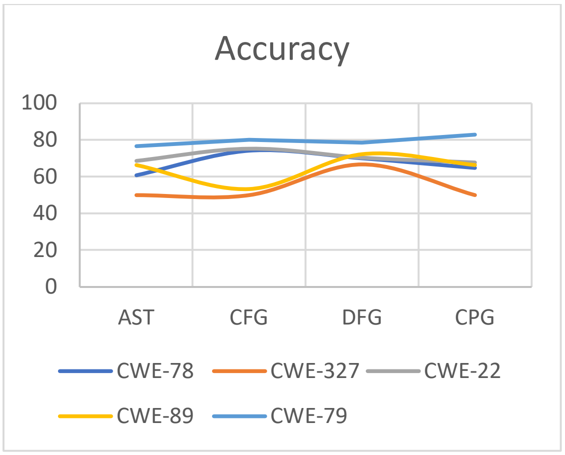

Figure 2: Accuracies

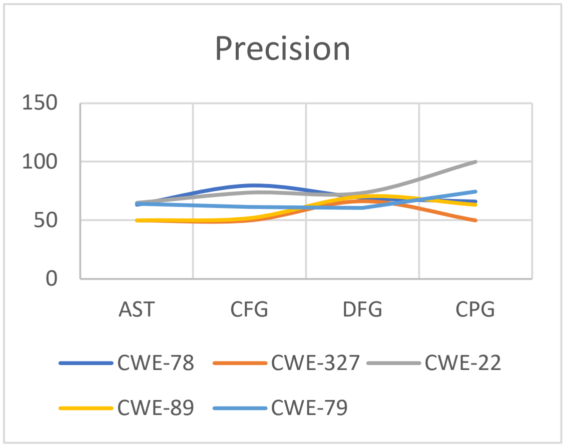

Figure 3: Precision

4.4.3 CWE-79

For CWE-79 (Cross-Site Scripting), the GGNN combined with CPG reached the highest accuracy (82.80%), precision (74.40%), and F1-measure (85.32%). This strong performance likely stems from CPG's comprehensive representation of code elements, including data and control flow, essential for capturing cross-site scripting characteristics.

- **AST:** GGNN+AST performed adequately for CWE-79, though it fell short in precision (64.10%) and recall (60.00%) compared to other methods. AST may miss dynamic relationships in code execution paths, which are often crucial in detecting the injection points of cross-site scripting vulnerabilities.
- **CFG and DFG**: CFG and DFG achieved strong recall and balanced precision, with CFG providing 80.00% accuracy. The data dependencies captured by DFG and the flow sequences captured by CFG help trace how untrusted data might reach client-side scripts, explaining their effectiveness for CWE-79 detection.

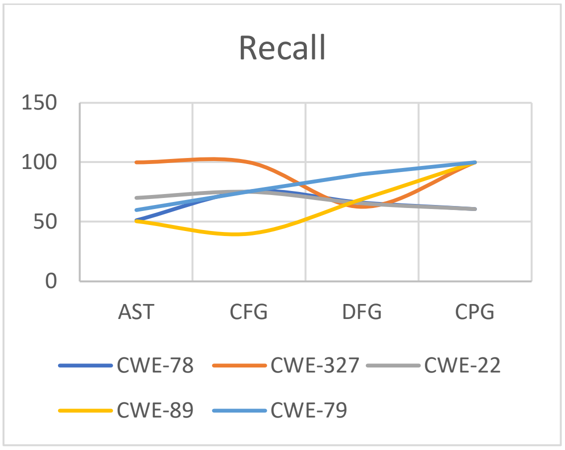

Figure 4: Recall

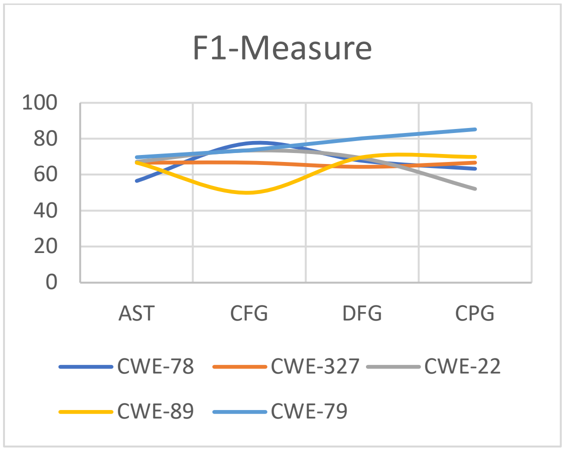

Figure 5: F-1 Measure

Figures 2, 3, 4, and 5 represent the charts for the GGNN with AST, CFG, DFG, and CPG for each metric, showing their relative strengths and weaknesses across different vulnerabilities. The accuracy of the model shows fluctuations across various across vulnerability types as shown in figure 2, with **CFG** generally yielding high values for CWE-78, CWE-22, and CWE-79, while **DFG** shows a peak for CWE-89. **CPG** has a prominent accuracy for CWE-79, whereas **AST** generally has moderate accuracy across most vulnerabilities, except CWE-327 where it performs lower. Figure 2 shows the chart for precision values of the model which vary, with **CPG** achieving a high value in CWE-22 due to perfect identification, but with relatively low recall. For CWE-79, **CPG** and **DFG** tend to be high. **CFG** performs well in CWE-78 and CWE-22, while **AST** shows moderate precision across all vulnerability areas except CWE-327.

Figures 4 and 5 show the chats for the recall and F1-measure of the model respectively. The recall chart highlights **CPG's** strength in CWE-327, reaching 100% recall, while **DFG** and **CFG** show generally balanced recall across other vulnerabilities. **AST** tends to show lower recall for CWE-78 and CWE-22 but performs adequately for CWE-327. For CWE-89, **DFG** achieves high recall due to effective data flow tracking. F1-measure lines are similar to recall, with **CPG** reaching high F1 scores for CWE-79 and CWE-327. **CFG** and **DFG** both achieve high F1 scores for CWE-78 and CWE-89 respectively, reflecting their strengths in control flow and data flow dependencies, while **AST** has moderate F1-scores across most vulnerabilities but drops for CWE-89 and CWE-79.

5. CONCLUSION AND FUTURE RESEARCH

This research presents a Gated Graph Neural Network (GGNN) framework that utilizes multi-graph representations (AST, DFG, CFG, and CPG) for enhanced vulnerability detection in software code, applied to five major vulnerabilities in the OWASP WebGoat dataset. Our results demonstrate that different graph structures capture unique aspects of code vulnerabilities, allowing the GGNN model to achieve notable performance improvements. Specifically, CFG and DFG representations are effective for identifying vulnerabilities that rely on control and data flow patterns, while CPG's composite structure supports high recall for cryptographic weaknesses and cross-site scripting.

Despite these advancements, the study has limitations. The model's performance is partially constrained by the quality and quantity of available vulnerability-labeled datasets, which impacts generalization to new or less common vulnerabilities. Additionally, certain vulnerabilities—such as those requiring contextual understanding beyond code structure (e.g., business logic flaws)—may not be fully captured by the graph representations used. The reliance on manual graph normalization and node encoding introduces potential inefficiencies and may limit scalability across large codebases.

Future research can address these limitations by exploring automated graph representation and encoding techniques to reduce preprocessing time and enhance model scalability. Additionally, incorporating contextual information from software documentation and runtime behaviors could enrich vulnerability detection, particularly for logic-based or dynamic vulnerabilities. Lastly, testing this framework on more diverse datasets, including non-Java applications, could further validate and refine its effectiveness across different programming environments.

ACKNOWLEDGMENT

This research was supported by the Faculty of Information Technology and Computer Science, Universiti Putra Malaysia.

REFERENCES:

[1] B. Wu and F. Zou, "Code Vulnerability Detection Based on Deep Sequence and Graph Models: A Survey," Secur. Commun. Networks, vol. 2022, 2022, doi: 10.1155/2022/1176898.

[2] G. Lin, S. Wen, Q. L. Han, J. Zhang, and Y. Xiang, "Software Vulnerability Detection Using Deep Neural Networks: A Survey," Proc. IEEE, vol. 108, no. 10, pp. 1825–1848, 2020, doi: 10.1109/JPROC.2020.2993293.

[3] H. Wang et al., "Combining Graph-Based Learning with Automated Data Collection for Code Vulnerability Detection," IEEE Trans. Inf. Forensics Secur., vol. 16, pp. 1943–1958, 2021, doi: 10.1109/TIFS.2020.3044773.

[4] C. Lin, Y. Xu, Y. Fang, and Z. Liu, "VulEye: A Novel Graph Neural Network Vulnerability Detection Approach for PHP Application," Appl. Sci., vol. 13, no. 2, 2023, doi: 10.3390/app13020825.

[5] W. Guo, Y. Fang, C. Huang, H. Ou, C. Lin, and Y. Guo, "HyVulDect: A hybrid semantic vulnerability mining system based on graph neural network," Comput. Secur., vol. 121, p. 102823, 2022, doi: 10.1016/j.cose.2022.102823.

[6] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "BGNN4VD: Constructing Bidirectional Graph Neural-Network for Vulnerability Detection," Inf. Softw. Technol., vol. 136, no. October 2020, p. 106576, 2021, doi: 10.1016/j.infsof.2021.106576.

[7] J. Wang, H. Xiao, S. Zhong, and Y. Xiao, "DeepVulSeeker: A novel vulnerability identification framework via code graph structure and pre-training mechanism," Futur. Gener. Comput. Syst., vol. 148, pp. 15–26, 2023, doi: 10.1016/j.future.2023.05.016.

[8] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," Adv. Neural Inf. Process. Syst., vol. 32, no. NeurIPS, pp. 1–11, 2019.

[9] Y. Fan, C. Wan, C. Fu, L. Han, and H. Xu, "VDoTR: Vulnerability detection based on tensor representation of comprehensive code graphs," Comput. Secur., vol. 130, p. 103247, 2023, doi: 10.1016/j.cose.2023.103247.

[10] X.-C. Wen, Y. Chen, C. Gao, H. Zhang, J. M. Zhang, and Q. Liao, "Vulnerability Detection with Graph Simplification and Enhanced Graph Representation Learning," pp. 2275–2286, 2023, doi: 10.1109/icse48619.2023.00191.

[11] Y. Li, S. Wang, and T. N. Nguyen, "Vulnerability detection with fine-grained interpretations," ESEC/FSE 2021 - Proc. 29th ACM Jt. Meet. Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng., pp. 292–303, 2021, doi: 10.1145/3468264.3468597.

[12] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software Vulnerability Detection via Deep Learning over Disaggregated Code Graph Representation," 2021, [Online]. Available: http://arxiv.org/abs/2109.03341.

[13] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network," ACM Trans. Softw. Eng. Methodol., vol. 30, no. 3, 2021, doi: 10.1145/3436877.

[14] X.-C. Wen, C. Gao, J. Ye, Z. Tian, Y. Jia, and X. Wang, "Meta-Path Based Attentional Graph Learning Model for Vulnerability Detection," pp. 1–13, 2022, [Online]. Available: http://arxiv.org/abs/2212.14274.

[15] S. Suneja, Y. Zheng, Y. Zhuang, J. Laredo, and A. Morari, "Learning to map source code to software vulnerability using code-as-a-graph," pp. 1–8, 2020, [Online]. Available: http://arxiv.org/abs/2006.08614.

[16] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," IEEE Trans. Dependable Secur. Comput., vol. 19, no. 4, pp. 2244–2258, 2022, doi: 10.1109/TDSC.2021.3051525.

[17] Z. Li et al., "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection," vol. 18, no. 5, pp. 2224–2236, 2018, doi: 10.14722/ndss.2018.23158.

[18] X. Duan et al., "Vulsniper: Focus your attention to shoot fine-grained vulnerabilities," IJCAI Int. Jt. Conf. Artif. Intell., vol. 2019-Augus, pp. 4665–4671, 2019, doi: 10.24963/ijcai.2019/648.

[19] "OWASP Benchmark Project." https://owasp.org/www-project-benchmark/ (accessed Jan. 23, 2022).