
Graph Neural Network Based Framework for Vulnerability Detection in Object Oriented Software Systems

Abstract

Software vulnerabilities in object-oriented systems often spread across classes, methods, variables, inheritance links, and data dependencies, making them difficult to detect through token-based analysis alone. This study presents a graph neural network framework that converts object-oriented source code into structured program graphs for automated vulnerability detection. Classes, methods, and variables are represented as nodes, while method calls, inheritance relations, control flow, and data dependencies are represented as edges. Structural and semantic features are processed through attention-based message passing and graph-level classification. The framework was evaluated using one to six graph layers to examine the effect of message-passing depth. The F1-score increased from 82.6% with one layer to 93.7% with four layers, then decreased slightly at deeper settings because of over-smoothing and added graph noise. The proposed framework captures wider software relations, improves vulnerability identification, and supports focused security testing and review in large object-oriented applications while maintaining practical computational efficiency.

Keywords: Graph neural network, Vulnerability detection, Object-oriented software, Program graph, Software security

1. Introduction

Software vulnerabilities are weaknesses in program design or implementation that may allow attackers to access data, change system behaviour, or interrupt software services. Object-oriented applications are difficult to examine because their behaviour is distributed across classes, methods, objects, interfaces, and inheritance structures. Traditional static analysis tools can identify known security rules, but they often produce many false warnings and may miss complex relations between software components. Graph representation learning can model source-code structures and support automated vulnerability detection at the function level [1]. This makes graph-based analysis suitable for large object-oriented systems with connected program elements.

Source code contains both textual information and structural relationships. Tokens describe the written program statements, while control flow, data flow, method calls, and dependencies describe how the program operates. Sequence-based learning models mainly process code as ordered text and may not fully capture distant relations between program elements. Code graphs preserve these relations by representing software elements as nodes and their connections as edges. Integrated graph neural networks can combine different graph-learning mechanisms to capture information that is distributed across program structures [2]. A richer code representation can therefore improve the identification of hidden vulnerability patterns.

Object-oriented software introduces additional security relationships through inheritance, method overriding, dynamic binding, encapsulation, and shared object states. A vulnerable method may receive unsafe data from another class or pass the data through several calls before the harmful operation occurs. Graph neural networks can transfer information between connected nodes and learn how local program elements influence a wider software structure. Weight-sensitive graph models can also give greater importance to code relations that are more strongly connected with vulnerable behaviour [3]. This ability supports the analysis of security problems that cannot be understood from one statement or method alone.

This study proposes a graph neural network framework for vulnerability detection in object-oriented software systems. The framework converts classes, methods, variables, method calls, inheritance links, control paths, and data dependencies into a unified program graph. Node features describe the meaning and properties of each program element, while edge features describe the relationships between elements. A graph classifier then estimates whether a software component contains vulnerable behaviour and identifies the program regions that contribute most strongly to the prediction.

2. Methodology

The proposed methodology contained source-code collection, object-oriented parsing, program graph construction, node-feature generation, graph neural network training, and vulnerability classification. Publicly available object-oriented projects with confirmed security labels were selected for the experiment. Each source file was parsed to identify classes, methods, variables, interfaces, method calls, and inheritance relations. Vulnerability labels were assigned at the method or class level according to the available security records. Graph confident learning can reduce the effect of incorrect labels in software vulnerability datasets [4]. A label-checking stage was therefore included before the final training dataset was created.

Each software component was converted into a structured graph. Classes, methods, and variables were represented as nodes, while method calls, variable access, inheritance, control flow, and data flow were represented as directed edges. The abstract syntax tree supplied the basic structure of the code, and additional analysis linked program elements that exchanged data or execution control. Enhanced graph structures can provide more useful security information than isolated token sequences [5]. The final graph preserved both the local syntax and the wider object-oriented relationships of each analysed component.

Node features were created from code tokens, data types, access modifiers, method size, complexity, parameter count, and security-sensitive operations. Token sequences were converted into dense numerical embeddings, while categorical properties such as public, private, protected, static, and final were encoded as feature values. Numerical software measures were standardized using the mean and standard deviation of the training data. Residual graph models can combine structural and textual code information to improve vulnerability learning [6]. The proposed feature vector therefore included both program meaning and measurable code properties.

Table 1 describes the graph components used by the proposed framework. Classes represented the main structural units of the object-oriented system, while methods represented executable behaviour. Variables described the storage and movement of program data. Method-call and inheritance edges captured relations between different software units, and data-dependency edges represented the movement of values between operations. These elements allowed the graph neural network to analyse how vulnerable behaviour developed across connected parts of the program.

Table 1. Object-Oriented Program Graph Components, Extracted Relations, Node Features, and Vulnerability-Relevant Information

Graph component	Extracted relation	Main feature representation	Security meaning	GNN input
Class node	Class declaration and membership	Class type, size, modifiers, and coupling	Shows structural risk and exposed class behaviour	Class feature vector

Method node	Method definition and class ownership	Tokens, complexity, parameters, and return type	Represents executable operations that may contain vulnerable logic	Method embedding
Variable node	Declaration, assignment, and access	Data type, scope, modifier, and usage count	Identifies sensitive data and unsafe value handling	Variable feature vector
Method-call edge	Caller-to-callee relation	Call direction and frequency	Tracks the spread of unsafe input across methods	Directed call edge
Inheritance edge	Parent-to-child class relation	Inheritance type and override status	Captures inherited or overridden vulnerable behaviour	Structural edge
Data-dependency edge	Definition-to-use relation	Source, destination, and dependency type	Represents unsafe data flow toward sensitive operations	Directed dependency edge

The graph neural network contained several message-passing layers. During each layer, a node received information from its neighbouring nodes and combined it with its own feature vector. Attention weights were used to assign greater importance to security-related neighbours, such as external input methods, unsafe memory operations, and sensitive application interfaces. Adaptive graph neural networks can select useful structural information while reducing the influence of unrelated graph connections [7]. The updated node representations were pooled to create one graph-level representation for vulnerability classification.

Class imbalance was handled within the training dataset because vulnerable components were less common than non-vulnerable components. Weighted cross-entropy assigned a higher loss to missed vulnerabilities, while stratified sampling preserved the class ratio across training, validation, and test sets. Dropout and weight decay were used to reduce overfitting. Integrated models based on graph convolution, graph attention, and propagation networks can capture complementary code relations [8]. The proposed model therefore combined attention-based message passing with residual connections between graph layers.

The model was trained using the Adam optimizer and early stopping based on validation loss. Graph depth, hidden feature size, dropout level, learning rate, and decision threshold were selected using the validation set. Accuracy, precision, recall, F1-score, and AUC were used to measure detection performance. Code classification experiments have shown that graph neural network behaviour can vary according to dataset realism and graph construction choices [9]. The final evaluation therefore used project-separated test data and examined performance across different message-passing depths.

3. Results and Discussion

The proposed graph neural network framework was evaluated using object-oriented program graphs with different message-passing depths. Models with one to six graph layers were trained and tested

under the same experimental settings. The F1-score increased during the early stages because additional layers allowed the model to collect information from a wider set of connected classes, methods, variables, and dependencies. This improvement shows that deeper graph propagation helped the model capture vulnerability patterns that were distributed across several software components. The wider information range also allowed the network to connect local code weaknesses with related structural behaviour.

The highest F1-score was obtained when the graph neural network used four message-passing layers. Figure 1 shows the change in vulnerability detection performance as the graph depth increased from one to six layers. The score improved from the shallow configurations to the four-layer model because the network could combine both local code information and wider object-oriented relations. The result indicates that a moderate graph depth provided the best balance between structural coverage and feature stability. This configuration captured useful relationships without adding excessive information from distant graph regions.

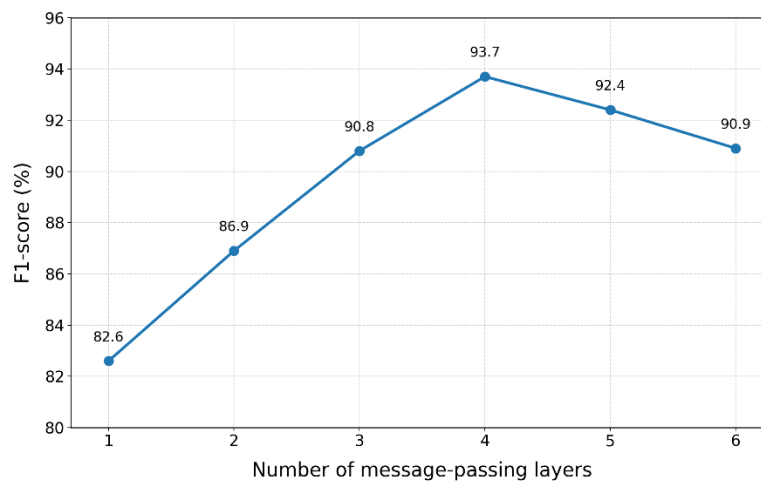


Figure 1. Vulnerability Detection F1-Score Across Increasing Object-Oriented Program Graph Depths

The F1-score decreased slightly when the model used five and six message-passing layers. This reduction may be related to over-smoothing, where node representations become too similar after repeated information exchange. Excessive graph depth can also introduce unrelated information from distant program elements and make vulnerable patterns less clear. The result shows that increasing the number of graph layers does not always improve vulnerability detection. Careful selection of graph depth is therefore necessary to preserve meaningful differences among code elements.

The four-layer model provided the most stable performance across the validation and test datasets. It captured method calls, inheritance relations, data dependencies, and class-level structures without introducing excessive graph noise. The selected architecture therefore used four message-passing layers in the final vulnerability classifier. This configuration offered a practical balance between prediction

quality, model complexity, and computational cost. It also reduced unnecessary processing while maintaining strong detection performance across different software projects.

4. Conclusion

This study presented a graph neural network framework for vulnerability detection in object-oriented software systems. The framework represented classes, methods, variables, method calls, inheritance links, and data dependencies as a unified program graph. Structural and semantic features were processed through attention-based message passing to identify software components that contained vulnerable behaviour. The graph representation allowed the model to examine both local code features and wider relationships among connected software elements.

The results showed that graph depth had a clear effect on vulnerability detection performance. The F1-score improved up to four message-passing layers and then decreased slightly at greater depths. The proposed framework can support security testing by identifying vulnerable software regions and by capturing relationships that are difficult to detect through token-based analysis alone. It provides a useful approach for improving automated vulnerability assessment in large object-oriented applications. The framework can also help security teams focus code review on components with the highest predicted risk.

References

1. Wu, Y., Lu, J., Zhang, Y., & Jin, S. (2021, January). Vulnerability detection in c/c++ source code with graph representation learning. In *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)* (pp. 1519-1524). IEEE.
2. Chen, J., Yin, Y., Cai, S., Wang, W., Wang, S., & Chen, J. (2024). iGnnVD: A novel software vulnerability detection model based on integrated graph neural networks. *Science of Computer Programming*, 238, 103156.
3. Liu, H., Jiang, S., Qi, X., Qu, Y., Li, H., Li, T., ... & Guo, S. (2024). Detect software vulnerabilities with weight biases via graph neural networks. *Expert Systems with Applications*, 238, 121764.
4. Wang, Q., Li, Z., Liang, H., Pan, X., Li, H., Li, T., ... & Guo, S. (2024). Graph confident learning for software vulnerability detection. *Engineering Applications of Artificial Intelligence*, 133, 108296.
5. Xiao, P., Xiao, Q., Zhang, X., Wu, Y., & Yang, F. (2024). Vulnerability detection based on enhanced graph representation learning. *IEEE Transactions on Information Forensics and Security*, 19, 5120-5135.

6. Tang, M., Tang, W., Gui, Q., Hu, J., & Zhao, M. (2024). A vulnerability detection algorithm based on residual graph attention networks for source code imbalance (RGAN). *Expert Systems with Applications*, 238, 122216.
7. Liang, C., Wei, Q., Jiang, Z., Wang, Y., & Du, J. (2024, October). A source code vulnerability detection method based on adaptive graph neural networks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops* (pp. 187-196).
8. Meng, X., Lu, S., Wang, X., Liu, X., & Hu, C. (2023, December). Improving vulnerability detection with hybrid code graph representation. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 259-268). IEEE.
9. Nguyen, H. V., Zheng, J., Inomata, A., & Uehara, T. (2022). Code aggregate graph: Effective representation for graph neural networks to detect vulnerable code. *IEEE Access*, 10, 123786-123800.