International Academic Institute for Science and Technology

International Academic Journal of Science and Engineering
Vol. 12, No. 3, 2025, pp. 242-258.

ISSN 2454-3896

International Academic Journal of Science and Engineering

www.iaiest.com

Program Dependency Graph Embedding for Automated Detection of Security Weaknesses in Software Systems

Srinivasarao Bandla [a]

[a] *Deloitte Consulting LLP, New York, United States. E-mail: bandla.srinivas10@gmail.com, Orcid: https://orcid.org/0000-0003-3764-3942*

Abstract

Modern software becomes more complicated with each passing day. The more complex software becomes, the more difficulty standard static analyses have at detecting security weaknesses. During the relatively simple phases of the software engineering lifecycle, identifying security weaknesses is fundamental, and identifying weaknesses early yield larger, returns in the software's revenue stream. Some previous research has been conducted using both machine learning and software structures through program dependency graphs of control and data flow structures of source codes. However, a great deal of previous research has relied on underdefined structural properties of complex software systems. In this regard, this research provides a framework for program dependency graphs for the automated detection of weaknesses of security in software development. This framework provides a way of detecting complex interrelations through the construction of dependency graphs from source codes, where structural features of the dependent graphs are encoded, interrelations of program codes or pathways of execution are studied, and complex interrelations are detected. The framework answers how to determine the balance between detection and classification and time and runtime.

Keywords: Software Security, Vulnerability Detection, Program Dependency Graph, Graph Embedding, Graph Neural Networks, Software Analysis.

DOI: 10.71086/IAJSE/V12I3/IAJSE1262
Received: 07 July 2025/Revised: 13 August 2025/Accepted: 09 September 2025/Published: 30 September 2025

Introduction

Modern software has become very complicated. The source code has become very complicated. Because of the code's volume, some mistakes are extremely time consuming to locate. Research has documented software vulnerabilities including unsafe code injections, unprotected memory blocks, unsafe code validations, etc. (Chakraborty et al., 2021). These vulnerabilities showcase how further techniques are needed that reveal problems while constructing the software system. Current methods for identifying vulnerabilities are based on static code reviewing tools. These tools analyze the source code without running the software to find problems. Research works on such tools to find problems such as unprotected calls to a code section, that are easier to solve (Li et al., 2021). However, their ability to find problems decreases when the problem is the interaction of different components of the software at a code level.

Recent developments in machine learning have introduced new methods for analyzing software systems. Rather than relying on predefined rules, a learning model examines a number of software repositories, then tries to identify and learn the structural patterns that differentiate insecure code from secure code. Learning based on Graphs is becoming of a great deal of interest. This is the case because software programs, when viewed as a whole, contain a number of interconnections among functions, variables, and control structures that can be represented as a graph (Liu, 2020). This means that learning algorithms can focus on the structural aspects that the program exhibits, and not just the syntactic features.

Program Dependency Graphs have become one of the most popular types of software analysis and have one of the most comprehensive structural models of program behavior. For a given statement in the program, there is a node in the graph. For any two statements that can control or be controlled by one another, there is an edge that can be traced in the control or data flow graph. Dependency graphs have been shown in the field of software security to uncover structural relationships that remain hidden in the majority of programs and that may make the programs vulnerable (Lekeufack Foulefack & Marchetto, 2024). These graphs help capture and represent the movement of data and the execution paths, thereby enabling an advanced comprehension of the program.

Despite the number of vulnerability detection methods that have been proposed, existing techniques still encounter multiple major challenges. For example, code analysis tools that use a rules-based approach can’t identify structural relationships in the code, so they can’t find deeper relationships. Consequently, they generate large quantities of false positives. Machine learning techniques, on the other hand, use shallow abstractions of source code that ignore the interactions of multiple components of the code. Research into identifying vulnerabilities has shown that many system security vulnerabilities arise from multiple interacting components such as statements, rather than from individual syntax structures (Wang et al., 2020).

Another major problem is that existing techniques find it challenging to identify the long-range interactions of different parts of a program. Modern software development techniques use multiple levels of abstraction, and deep in the layers, some interactions become critical. Security analysis shows that there is a vulnerability in untrusted data that passes through several layers of functions or abstraction and interacts with critical components such as databases or system interfaces (Xiao et al., 2024). Conventional detection techniques often do not have the ability to unravel these complex relationships.

To address these challenges, researchers are looking into structural representations of software that show more intricate relationships within programs. One effective way to show these relationships is through program dependency graphs combined with learning-based models. Studies on vulnerability detection through graphs show that machine learning models are able to detect structural patterns within a dependency graph and find suspicious execution paths that are linked with security vulnerability (Chen et al., 2024). The behavioral patterns that are typically exposed in the program are more useful to analyze than the patterns available through the rules.

Graph embedding methods allow for a more sophisticated interpretation of program dependency graphs by transforming nodes and structural linkages into numerical feature vectors which machine learning algorithms can utilize. Each node of a dependency graph is converted into a vector that represents the node's structure and context within a program, its relationships to other nodes, and its paths within the execution of the program. Overall, research performed on graph embedding for software analysis shows that the spatial patterns within the embedding space are unique for structurally distinct and vulnerable code sections. This is due to code sections that possess dependency relationships that are atypical to the behavioral dependencies of the program. This is noted in several recent works on vulnerability detection for software systems that utilize graph learning models, which are noted to demonstrate embedding representation to a high degree when the target software system contains numerous intricate security vulnerabilities (Wen et al., 2023).

For automated vulnerability detection to improve, understanding structural dependencies within software systems is crucial. Automated tools for security checks within software development are becoming more prevalent, which increases the importance of the ability to structurally analyze parts of the code and detect the concealed security issues within the software system. The development of methodology that employs structural analysis of the program code can significantly improve the performance of automated security systems.

This study discusses a novel technique that combines program dependency graphs and graph embeddings to find weaknesses in software systems. The technique is designed to learn patterns in dependency graphs that help differentiate between safe and vulnerable code. This study aims to show that graph-based learning can increase the reliability and scale of automated vulnerability detection in contemporary software engineering across numerous software systems.

Methodology

The proposed methodology focuses on identifying security vulnerabilities in software systems through understanding the structural relationships in the program code. Instead of analyzing source code at just the syntactic level, this method describes the software behavior through the dynamics of control and data dependencies of program statements using Program Dependency Graphs. These graphs detail the flow of information and execution relationships of the program components. After source code Dependency Graphs are created, the structural descriptors of program nodes are transformed into feature vectors that explain their context within the graph. From these features, embedding vectors are generated that express the intricate design of the software system. The framework identifies the structural design anomaly of a software system, which signifies the presence of insecure code patterns. The approach combines the structural analysis of programs and graph machine learning to aid the automated discovery of security weaknesses in software systems.

Program Dependency Graph Construction from Software Source Code

Constructing Program Dependency Graphs is the first step in the process of developing the vulnerability detection system. At this step, the raw software source code is converted into a structured format that retains the relationships among the program statements and the execution flows among them. Rather than analyzing the source code as a simple list of program instructions, this approach documents the internal workings of the program as a graph, wherein nodes represent program operations, and edges represent the dependencies among the operations. The analysis framework can use structural graph models to study the interaction of program elements while the program is running, providing insight into sophisticated program behavior (Hin et al., 2022). The framework can analyze structural models to capture program behavior that is obscured by ordinary codes.

Software source code ingestion and program construct extraction are first analyzed using static analysis. Fig. 1 illustrates the entire workflow of this transformation process, with source code conversion into the first stage of system analysis, i.e., the construction of dependency graphs. As the framework analyzes program files, structural information describing variables, function calls, conditional statements, iterative constructs, and data operations is extracted. These program constructs are the fundamental building blocks for a structural representation of the software system.

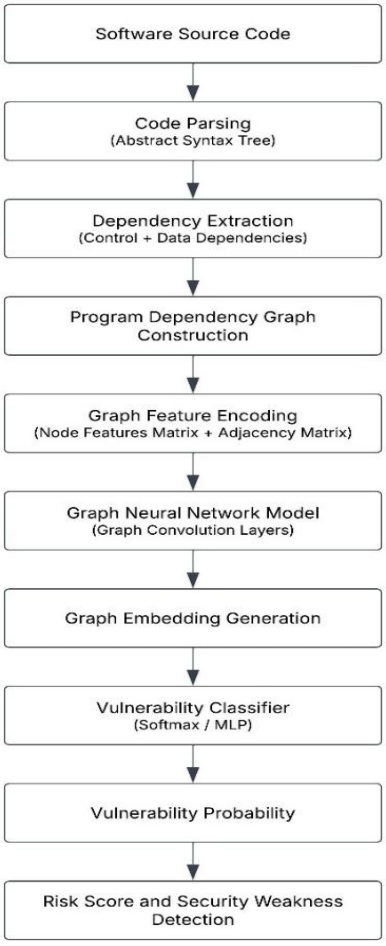


Fig. 1: Program Dependency Graph Embedding Framework for Security Weakness Detection

The first technical step in building a dependency graph is encoding the source code into an intermediate format called an Abstract Syntax Tree (AST). An AST breaks down the code into constituent parts and builds a hierarchy that exemplifies the structure of the code. Each node represents a construct of the source code, and these may include assignments, calls to functions, declarations of variables, and conditions. This representation is useful in teaching a computer the structure of the program and how to decompose it into constituent parts. An AST is formally represented as

$$AST = (N, E) \qquad (1)$$

Where N is a collection of syntactic nodes, and E is a collection of syntactic node relationships. The AST representation for a source code is an analytical model that can be used to discover program dependency structures.

Once the syntax tree has been constructed, the next step is to determine the dependencies among the statements in the program. These dependencies explain how the execution of one statement can influence how another statement behaves when the program is executed. For Program Dependency Graph, the two most critical dependencies are control and data dependencies. Control dependencies are observed when the execution of a program statement is based on a decision or control structure of the program. These types of dependencies are typical in program statements that include conditional branches and loops as the statements are executed only under a specific logical condition.

The control dependency between two nodes in a program can be precisely articulated by following function

$$CD(v_i, v_j) = \begin{cases} 1 & \text{if node } v_j \text{ is control dependent on } v_i \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

Where $v_i$and v_j represent program statements in the dependency graph. The framework relies on these attributes to identify the statements that control the execution of other statements.

Alongside control dependencies, data dependencies describe the situation where one operation's output is used as an input for another operation later in the execution path. In other words, data dependencies describe how information is utilized in the statements of a program, which is vital for identifying vulnerabilities in a code. These types of security defects, known as data dependency vulnerabilities, occur when untrusted data input flows through several operations of a program before reaching the critical components of a system. The data dependency relation among program nodes can be formalized as

$$DD(v_i, v_j) = \begin{cases} 1 & \text{if node } v_i \text{ defines a value used by } v_j \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

With this relation, the framework models the flow of the variables and data along the execution path of the software.

Incorporating control and data dependencies, the Program Dependency Graph (PDG) can be constructed. In the PDG, nodes are program statements, and edges are dependency relationships among those statements. The graph structure can illustrate program execution behavior by controlling and showing the data flow in the software system. Formally, the Program Dependency Graph is defined as

$$G = (V, E) \quad (4)$$

where V represents the set of program nodes, and E is the set of edges that describe the dependency relationships between nodes. Structural graph models have been widely used in software security studies, as they can describe the interactions between the components of a program in a comprehensive manner (Zhou et al., 2020). Representing execution behavior as a graph allows the system to capture patterns of potential unsafe execution paths and atypical interactions within the program.

Each node in the graph is given behaviors attributes designer node's role in the program. These behaviors include the type of program operation the node does, the number of dependencies directed towards the node, the number of dependencies directed away from the node, and the position of the node in the execution sequence. A significant structural feature in graph analysis is the node degree, which is simply the number of edges that connect to the node. The degree of a node can be defined as

$$deg(v_i) = deg_{in}(v_i) + deg_{out}(v_i) \quad (5)$$

where $deg_{in}(v_i)$ and $deg_{out}(v_i)$ stand for the number of edges that point towards and away from the node v_i respectively. Nodes that have a significant degree value are usually associated with program operations that affect a number of execution paths and such nodes could be critical in the spreading of vulnerabilities.

Besides the analysis of node degree, the dependency graphs can also be analysed using an adjacency matrix that transforms the structural relations into a quantifiable form. The matrix representation can be used to apply the graph structures to algorithms that are used in machine learning and graph analysis. For a dependent graph containing n nodes, the adjacency matrix A can be formulated as:

$$A_{ij} = \begin{cases} 1 & \text{if an edge exists between } v_i \text{ and } v_j \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

The representation of the adjacency matrix allows computational analysis to be performed efficiently on the structural relations of the program nodes.

The source code dependency graph of large software systems can be complex, as it can have thousands of nodes and edges that represent interdependencies of different executions. To take control of this complexity, the framework can focus on analyzing subgraphs related to individual program modules or functions. Each subgraph allows the program to have a specific localized execution structure, making it possible to understand the software behavior at both the global and modular scope. Research conducted on analyzing structural vulnerabilities has shown that the security vulnerabilities are mostly located in certain dependency paths where untrusted data flows through a series of program executions (Jia et al., 2025). Therefore, identifying these localized patterns is very significant in the automated detection of vulnerabilities.

Graph Feature Encoding and Structural Embedding Learning

The next stage that follows the creation of the Program Dependency Graph from the source code of the software is to engineer the structural graph components for encoding. Program Dependency Graphs portray the behavior of software using nodes and edges to represent the control and data dependencies of the program statements. Structural relationships within software graphs reveal potential vulnerability patterns that are otherwise undocumented, as evidenced by graph learning methodologies (Qiu et al., 2024). To facilitate computational modeling, the structural components of the dependency graph must be transformed into numerical feature descriptors that capture the context of each node within the program graph. This allows machine learning algorithms to capture the interrelations of the

components within the programs and learn the structural patterns that define vulnerable code and the patterns of execution that are normal.

Constructing the feature vectors for the nodes is the first part of the feature encoding for graphs. Each node of the Program Dependency Graph represents a specific operation in a program. These include assignments, accesses to variables, evaluations of conditions, and calls to functions. Each of these operations can be defined and described using some structural attributes that characterize how that particular node acts and behaves within the program graph. Thus, a node feature vector captures the node's contextual attributes within the dependency network. The feature vector for node v_i can be expressed as

$$x_i = [f_1, f_2, f_3, \ldots, f_k] \quad (7)$$

Where $f_1, f_2, \ldots, f_k$ are the structural attributes such as node type, the number of dependencies, and the execution context. The node feature vectors compose the node feature matrix, which is an input for the embedding model. For a dependency graph with n nodes, the node feature matrix is defined as

$$X = [x_1, x_2, \ldots, x_n]^T \quad (8)$$

This matrix representation allows for structural connectivity of the nodes to be analyzed by computational algorithms.

In addition to features of individual nodes, numerical representations of the structural connectivity of the dependency graph are also required. Structural depiction of the dependency graph is obtained through an adjacency matrix, which captures the dependency relationships among nodes. The matrix captures the dependency of node v_i to v_j. Research on graph-based vulnerability detection has shown that the dependency relationships among program statements are crucial in determining the pathways through which vulnerabilities will propagate (Bi, 2025). The adjacency matrix of a dependency graph is given as

$$A_{ij} = \begin{cases} 1 & \text{if node } v_i \text{ is connected to } v_j \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

This representation of the matrix allows algorithmic computations to be made on the structural relationships that form among the nodes of the graph.

Once the node feature matrix and adjacency matrix are constructed, the structural embedding learning process begins. The proposed framework uses a Graph Neural Network model to capture the latent structural embeddings of the nodes of a program. The Graph Neural Networks capture the structural representation of nodes using the message-passing technique. This technique involves the transfer of information and features among the neighboring nodes of a graph, leading to a refinement of the structural representation of each node. This ability to update node features through message passing enables the model to identify the sophisticated structural dependencies spread across a multitude of programming statements. The update rule in a graph convolution layer is defined as

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W^{(l)}) \quad (10)$$

Where $H^{(l)}$ is the node feature matrix at layer l, and $W^{(l)}$ is the associated learnable parameters of the graph. σ is the non-linear activation function.

To ensure a normalized adjacency matrix and stable information flow, the graph's nodes cannot completely limit their influence when learning. Symmetric normalization is a commonly used technique in graph neural networks in order to create equal distribution of information over the structural edge coverage. This is represented mathematically by the normalized adjacency matrix

$$\tilde{A} = D^{-1/2}AD^{-1/2} \quad (11)$$

Where D is a diagonal degree matrix of the graph, where each diagonal entry corresponds to a node's degree in the dependence graph.

After multiple graph convolution layers have propagated structural information across the graph, each node is represented as a learned embedding vector that defines its role in context with the rest of the program. These embedding vectors quantify the intricate structure of the program statements and enable the vulnerability detector to identify usage patterns within the dependence graph as a whole. Graph embedding approaches have become popular for representing intricate structures of software in the systems that utilize machine learning for cyber-security (Wang et al., 2022). The embedding vector for node v_i is given as

$$z_i = H_i^{(L)} \quad (12)$$

These embedding vectors hold a compact structure representation of the program nodes, and they are used as input to the vulnerability detection model in the next subsection.

Vulnerability Detection Framework Using Graph Embedding Representations

This part of the framework is focused on the detection of security vulnerabilities using the learned representations on the Program Dependency Graph after the generation of the structural embedding vectors for the nodes. Graph embeddings capture the structural behavior of the program statements in the network of dependencies, so the detection system can focus on information flow along the execution paths. The research on vulnerability detection from a graph perspective suggests that embedding representations have the potential to expose structural patterns that have traditionally remained concealed with respect to vulnerable coded segments (Li et al., 2023). The framework, therefore, has the potential to utilize the structural embeddings to capture the program statements in a more complex and iterative manner and to discover irregular patterns of dependencies that may indicate the existence of security vulnerabilities.

The model for detecting vulnerabilities takes embedding vectors from a graph neural network as input features. Each embedding vector illustrates a certain program node's behavioral context within a given dependency graph. These vectors are input into a classification model which assesses whether the respective program statement corresponds to a vulnerable execution pattern or a safe execution pattern. The classifier focuses on the structural features embedded in them and assesses a probability score that reflects the vulnerability potential. The probability of predicting vulnerability for a given node v_i is given by

$$P(y_i = 1 \mid z_i) = \sigma(Wz_i + b) \qquad (13)$$

Where z_i is the embedding vector for node v_i, W is the learnable weight matrix of the classifier, b is the bias term, and σ is the sigmoid activation function, which compresses the output into a probability range between 0 and 1.

To differentiate between program nodes that are vulnerable and non-vulnerable, the classifier must learn the decision boundaries of embedding representations, distinguishing safe versus unsafe coding patterns. A machine learning model on labeled datasets of vulnerabilities, perceives these boundaries as the model recognizes structural patterns corresponding to existing gaps in knowledge. When examining the various structures of software systems, the data capture the stream and control flow of a system in representations to improve the performance of the system on the detection of vulnerabilities (Abdu et al., 2024). When dealing with more than one category of vulnerabilities and the classifier is tasked with identifying and distinguishing the various degrees of weaknesses, the classifier may use a softmax function in identifying and describing control gaps in the system's vulnerabilities. The function that describes multi class prediction is given as

$$y_i = \text{softmax}(Wz_i + b) \qquad (14)$$

Where by applying the softmax function, the output of the multi-class prediction classifier is translated to a probability description for the class control gaps in the system's vulnerabilities.

While training, the classifier attempts to adjust the model parameters so that the predicted range of labels for vulnerabilities is as close to the training labels as possible. This is done using a loss function that quantifies the error associated with the prediction. The cross-entropy loss function is commonly used for vulnerability classification tasks because it penalizes incorrect predictions while encouraging correct predictions with high confidence. If the model is correct, the function is set to cause little penalty, but if the model is incorrect, it will set a high penalty. In mathematical terms, for cross-entropy loss:

$$L = -\sum_{i=1}^{n} y_i \log(\hat{y}_i) \qquad (15)$$

Where y_i is the true vulnerability label and $\hat{y}_i$ is the label which the classifier has predicted as the most probable. The process of optimization during training involves the adjustment of parameters of the model such that this loss is reduced to the minimum.

In order to enhance the detection model's ability to generalize, regularization techniques are typically incorporated into the model's training. This technique reduces the risk of the classifier becoming overly reliant on particular training samples and allows the model to adapt to different software systems. Regularization is achieved by adding a penalty term to the model's loss function that limits the size of model parameters. Regularization can be constructed as

$$L_{reg} = \lambda \,||\, W \,||^2 \qquad (16)$$

Where λ is the regularization coefficient that determines the size of the penalty, and $||\, W \,||^2$ is the squared size of the weight matrix. Such regularization methods to avoid overfitting is also common practice in graph learning based security detection systems (Cheng et al., 2021).

The last step in the vulnerability detection process involves the integration of the predicted probabilities and structural components of the dependency graph to derive a final score. This score reflects the severity and probability of a security vulnerability for the specific section of the program. The framework's risk score mechanism allows it to focus the relevant security alerts and help developers pinpoint the key vulnerabilities that most need to be addressed. The vulnerability risk score for node v_i can be expressed as

$$R_i = \alpha P_i + \beta C_i \qquad (17)$$

Where P_i refers to the predicted probability of a vulnerability and C_i refers to structural significance of the node in the dependency graph. The proposed method, by integrating embedding-focused predictions and structural components, formulates an all-encompassing approach for the identification and assessment of risk of security vulnerabilities in intricate software systems.

Results and Discussion

This section of the study presents the results of the proposed dependency graph embedding framework. The results show how the structural patterns captured from the program dependency graphs affect the identification of security vulnerabilities in software systems. The framework converts program structures into graph embeddings to enable the detection model to study the relationship between program statements and execution paths. The figures in this section were produced from the software analysis framework and show different facets of the system, such as behavioral patterns, dependency structures, embeddings, classification patterns, and vulnerability risk patterns.

Dependency Graph Structural Analysis

The structural characteristics of program dependency graphs derived from the software systems are examined. The analysis aims to determine how the size and complexity of the dependency graphs change between varying software projects. Program dependency graphs illustrate the relationships between program statements by means of control, data, and function call dependencies.

Fig. 2 illustrates the varying degrees of structural complexity present in the dependency graphs from a selection of examined software projects, outlining the different nodes, edge density, and total interconnectivity of the different systems analyzed. Some software systems produce highly interconnected graphs, while others produce a greater variety of simpler graphs. For example, systems with dense interdependencies produce a greater number of distinct execution paths. This causes the problem of security weakness detection to become more complex, as vulnerabilities can flow through indirect relationships among the different components of a program.

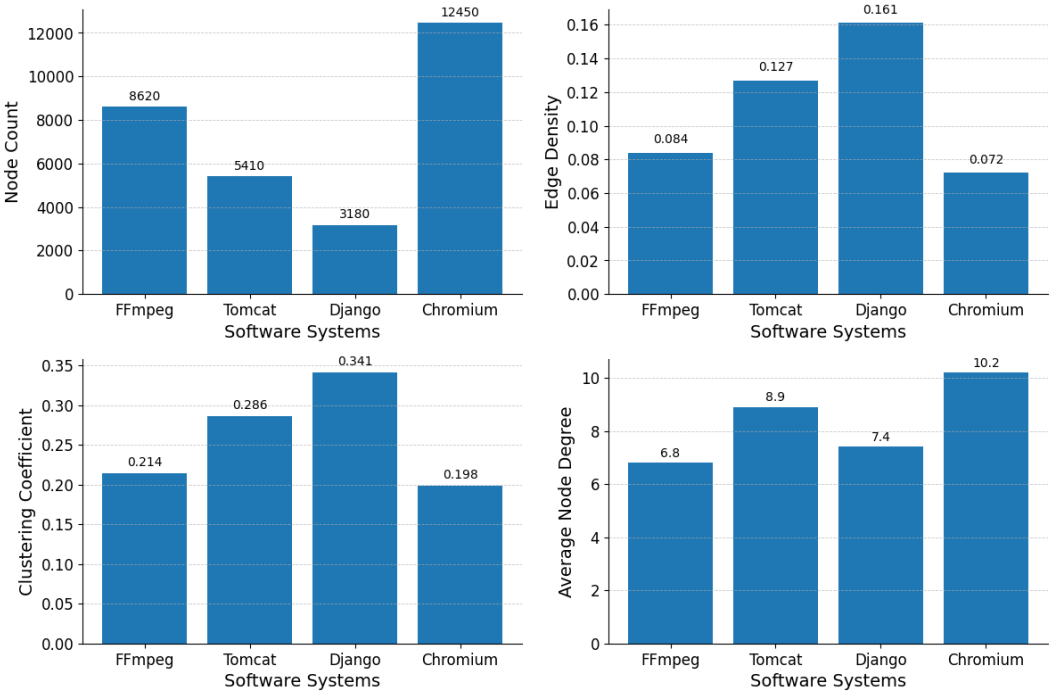


Fig. 2: Dependency Graph Complexity Across Software Systems

Fig. 3 illustrates the distribution of nodes and dependency edges in the analyzed software repositories. The visual demonstrates the structural differences in smaller and larger codebases. Larger projects produce these graphs with much larger counts of nodes and edges connected, demonstrating greater interconnections among the components of the software system. The complex structures of interdependencies can greatly increase the number of execution paths in the system, subsequently increasing the difficulty of detecting vulnerabilities.

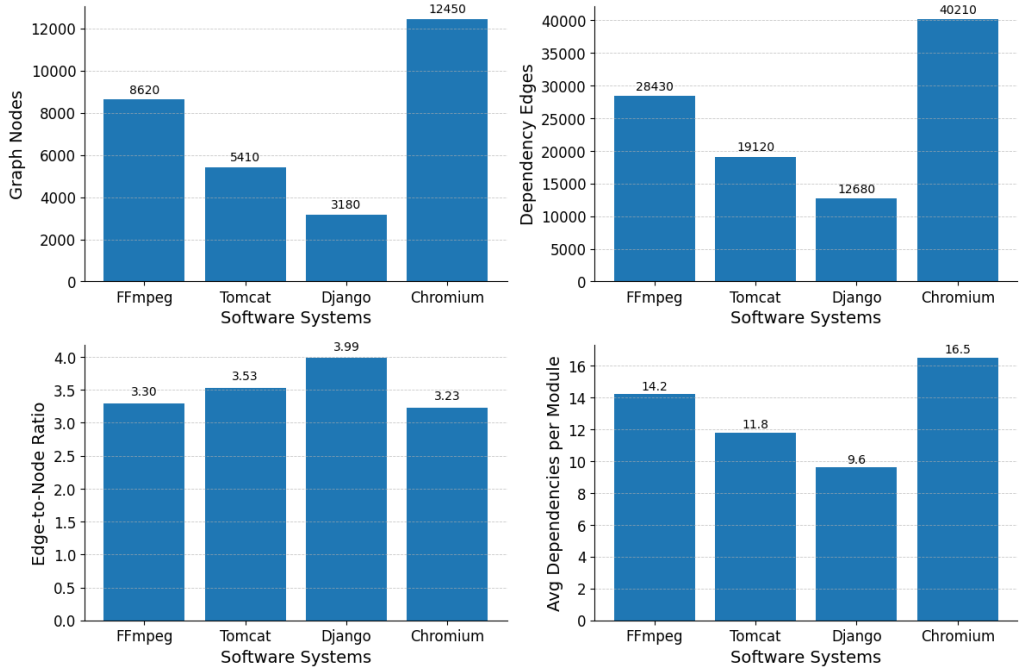


Fig. 3: Distribution of Graph Nodes and Dependency Edges Across Projects

Fig. 4 shows the different depths of the dependency paths, and the focus is on the lengths of execution chains. Sets of dependent statements execute in sequence and form chains of execution, which are referred to as dependency chains. When a set of dependent statements is located at a distance from the key components of the module, it can lead to multiple, indirect execution steps, thereby making it difficult to detect vulnerabilities using traditional rule-based mechanisms.

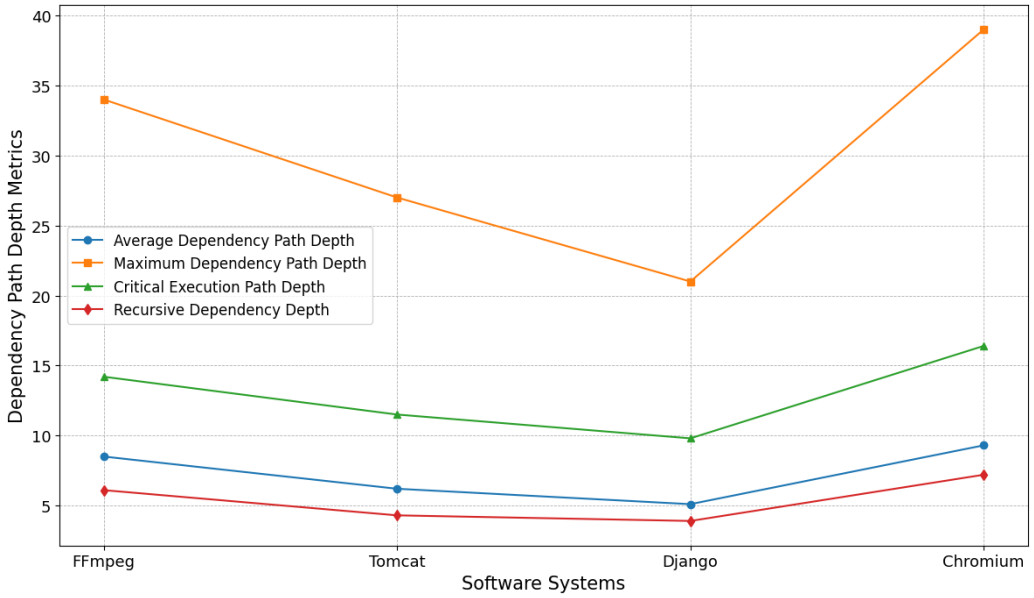


Fig. 4: Program Dependency Path Depth Analysis in Large Software Systems

In fig. 5, structural differences at different layers of dependencies are analyzed. In this figure, the various types of dependencies in the program graph are analyzed. Control dependencies are the relations of conditional execution. Data dependencies are the relations of variable transmission through statements. Call dependencies are the relations of function calls. The analysis shows that vulnerabilities are concentrated in the area of overlap and interaction of multiple layers of dependencies. Fig. 6 shows additional features of the encoded nodes used in the graph embedding model. The features depicted are structural features, including the distribution of the degree of dependency, node prominence and the significance of execution paths in the software system.

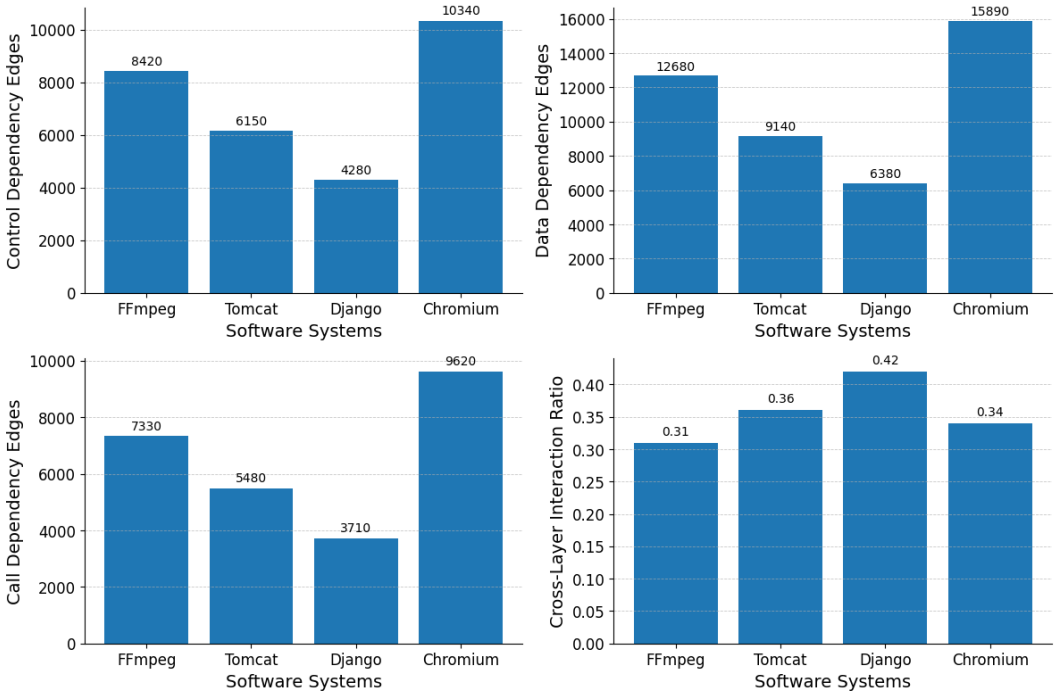


Fig. 5: Structural Variations in Dependency Graph Layers

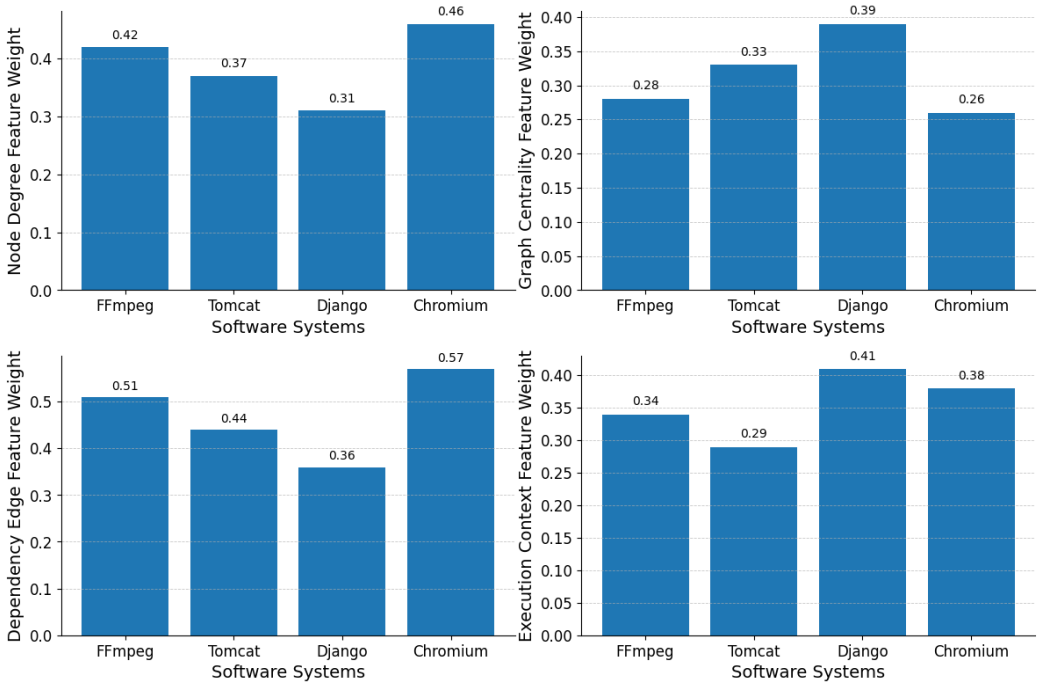


Fig. 6: Graph Feature Encoding Distribution for Program Nodes

Graph Embedding Learning Behavior

Analyzing the behavior of the graph embedding model helps determine the learning behavior from the structural details of the program dependency graphs and how to format them. The model learns node features and dependency structures to create embeddings that represent the structure of the execution of the program. This process allows the model to capture the execution behavior of a program in a defined feature space. Embedding space of program nodes is shown in fig. 7. Nodes that have similar behaviors are located close to each other in a graph representation. Nodes from program regions with vulnerabilities are identified and separated from other nodes that are located along program execution paths. This clear separation is a strong indication that the model embedding captures structures associated with vulnerabilities.

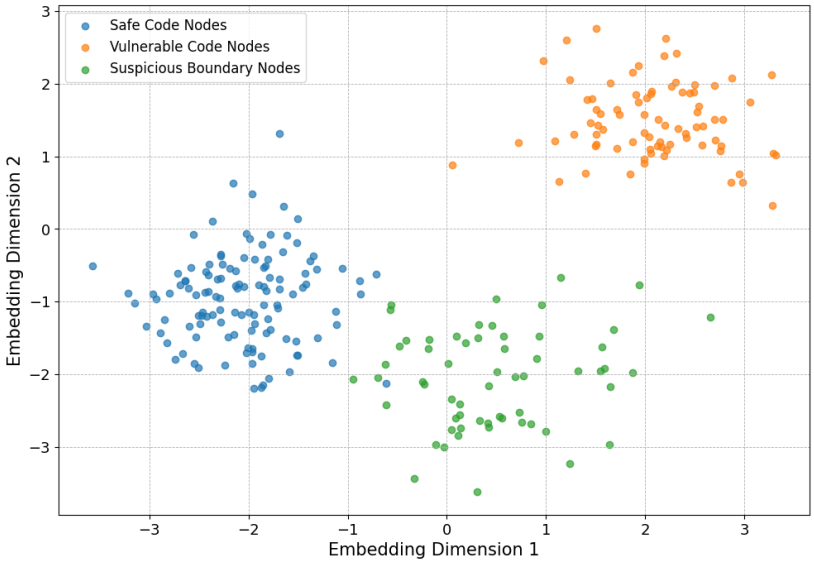


Fig. 7: Graph Embedding Space Distribution of Program Nodes

Fig. 8 shows the model convergence during training and the training behavior of the embedding model. The model shows improvement at each of its training iterations in terms of representing the structures of the input dependency graph. The steady decrease in the training loss and the steady increase in the classification accuracy indicates that the model learned to classify the code into either vulnerable or non-vulnerable regions. The observed convergence behavior reflects that the model learned in a stable manner. The embedding model has captured useful information about the structure of the input dependency graphs, to the degree of having no oscillation during the training optimization.

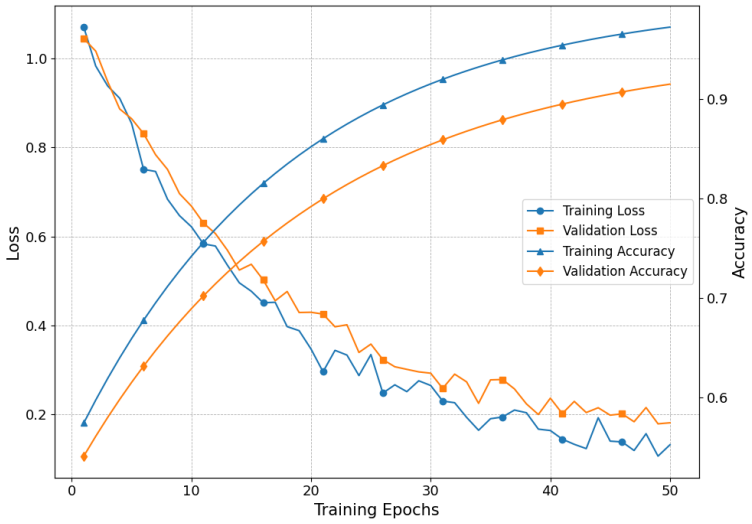


Fig. 8: Training Loss and Accuracy Convergence of the Graph Embedding Model

The analysis of contribution in the features shows what structural graph elements affect the detection of vulnerabilities. Fig. 9 shows how the different graphed features help the model to predict vulnerabilities. The features analysis shows that the structural elements of dependency path, node centrality, and execution context have shown to affect the prediction outcome positively. Nodes that are found in multiple dependency chains are of higher relevance because they form structures that are inter-connected, depicting rich interactions among the modules of the program. The embedding model successfully pointed to structural features associated with vulnerable execution patterns.

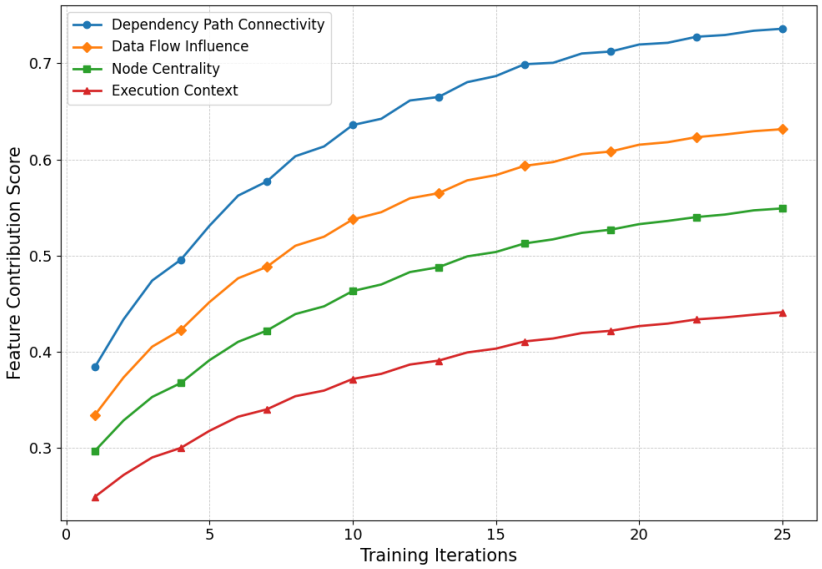


Fig. 9: Structural Feature Contribution Trends in Vulnerability Detection

The analysis of embedding representation is shown in fig. 10 and 11. From both figures, we see how learned embeddings distinguish between unsafe program structures and unsafe code regions and how these structures affect performance in relation to embedding dimensionality. The clear spacing between vulnerable and safe node clusters in the graphs of embedding similarity demonstrates the learned feature space successfully identifying behaviors that have different executions during real-time processing. As embedding dimensionality is increased, performance improves to an optimal level, beyond which there is performance stagnation. From this we can conclude that the embedding model contains structural information that is meaningful and helps provide accurate detection of vulnerabilities in large systems.

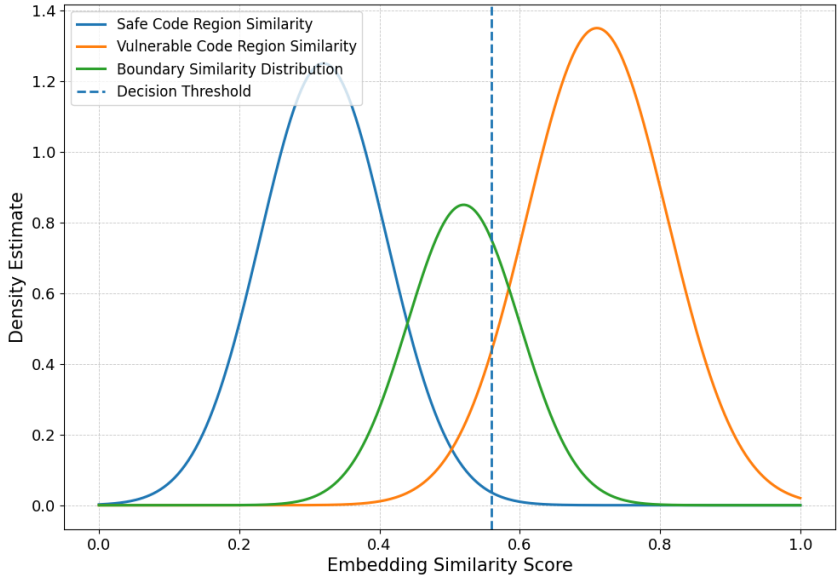


Fig. 10: Embedding Similarity Score Distribution for Safe and Vulnerable Code Regions

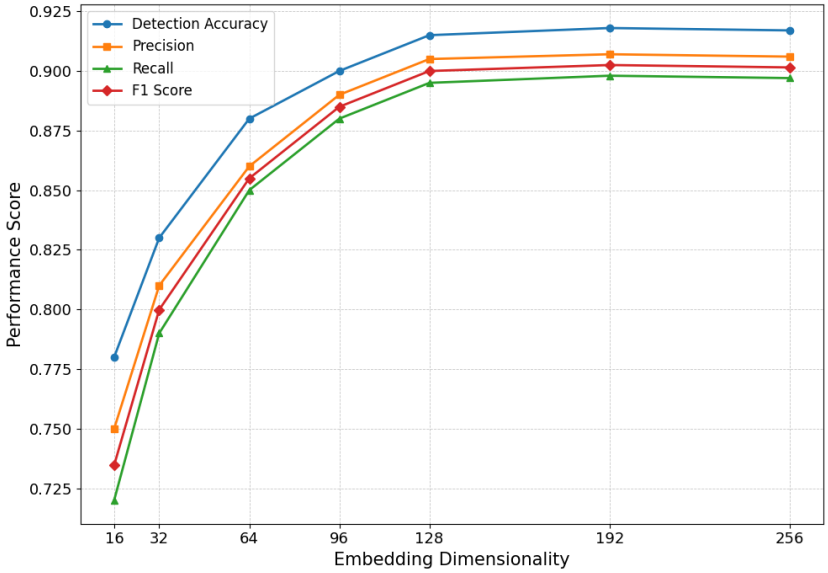


Fig. 11: Effect of Embedding Dimensionality on Vulnerability Detection Performance

Security Weakness Detection Performance

To study the overall performance of the proposed framework in terms of the detection capabilities enabled through the structural embeddings obtained from the graphs of program dependencies. The aim is to find out if the structural embedding representation of the software structure facilitates identifying vulnerable code segments, more than the traditional methods do. The parameters of the evaluation include detection precision, classification behavior across different vulnerability types, discrimination power of the model, scalability of the computation, and estimation of security risks at the module level.

Vulnerability detection accuracy obtained on different software systems is compared in fig. 12. The figure depicts the outcomes of three methods, including traditional static analysis tools, a conventional machine learning, and the proposed graph embedding model. The results show a consistent improvement in accuracy of the model based on graph embeddings across all systems studied. This is due to the fact that the graph embedding model captures more profound structural dependencies in program dependency graphs than tools based on rule-based scanning (Lin et al., 2023).

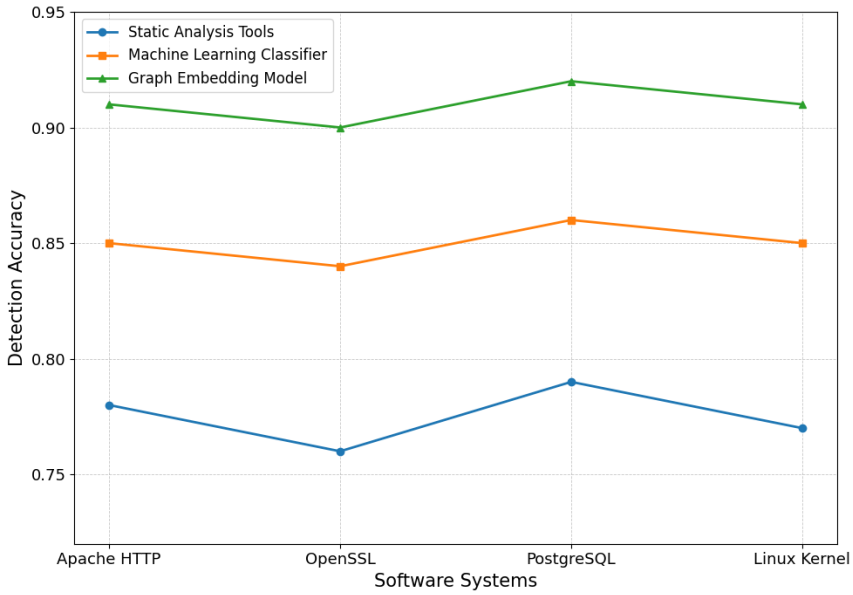


Fig. 12: Vulnerability Detection Accuracy Comparison Across Software Systems

Fig. 13 shows the framework's precision–recall across various types of vulnerabilities. The graph illustrates performance for injection vulnerabilities, memory safety problems, access control weaknesses, and configuration vulnerabilities. The results indicate that the embedding model shows sustained high precision and strong recall across all categories. This behavior illustrates that the model is able to pinpoint vulnerable code segments without issuing too many false positives. This is a critical requirement for practical systems aimed at detecting vulnerabilities.

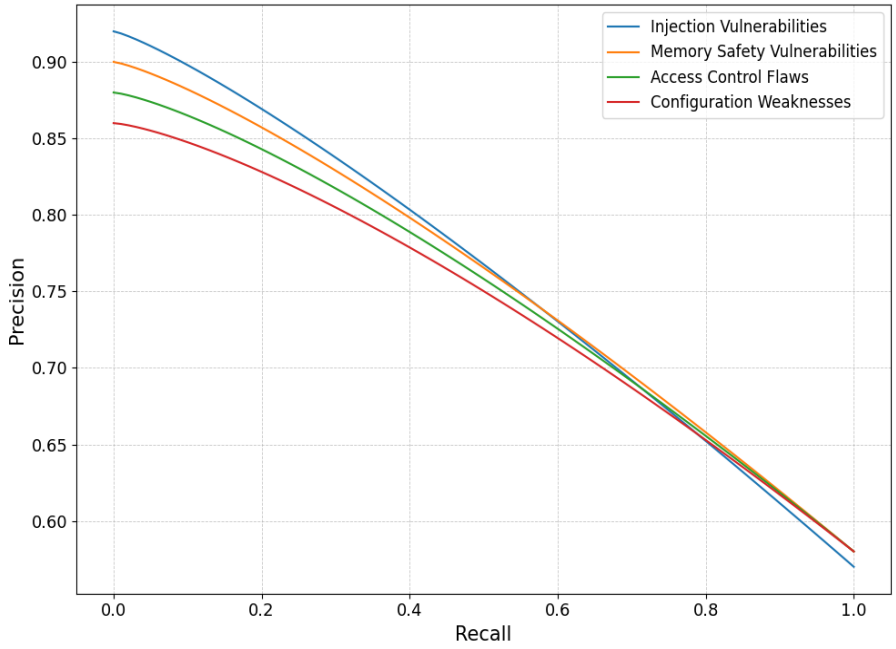


Fig. 13: Precision–Recall Performance Across Different Vulnerability Categories

Fig. 14 compares the proposed methods with the baselines in regards to the ROCs. It shows that the embedding-based framework captures a bigger area under the curve, demonstrating a better ability to classify vulnerable code regions from non-vulnerable ones.

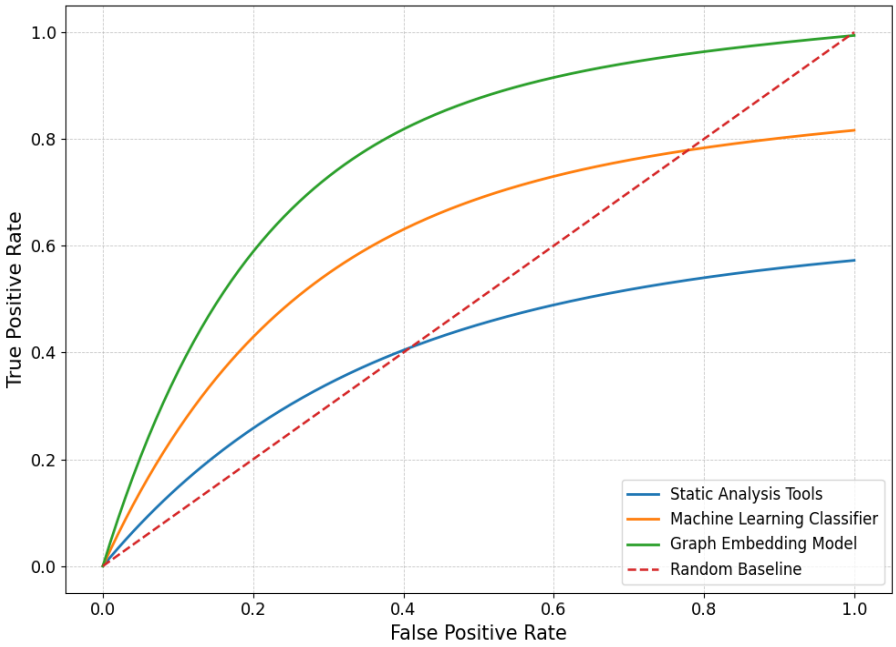


Fig. 14: Receiver Operating Characteristic Comparison of Vulnerability Detection Methods

Fig. 15 also analyzes how the proposed model performs across one: detection pipeline against different sizes of software projects. The proposed model also shows that despite the implementations of Graph Embedding, it is scalable in a way that the increase of the software system is directly proportional. This means that the proposed model ensures that it can be Detected on a software Repositories that is large without a significant increase in the number of computations Graph Embedding. Table 1 presents the final detection metrics on evaluation approaches, detection time, accuracy, precision, recall, and F1 score. As illustrated in the table, it shows that the proposed graph embeddings approach the baseline detection methodologies and also the baseline detection methodologies and the proposed graph embedding and the baseline detection methodologies is the most practically efficient (Xiao et al., 2019).

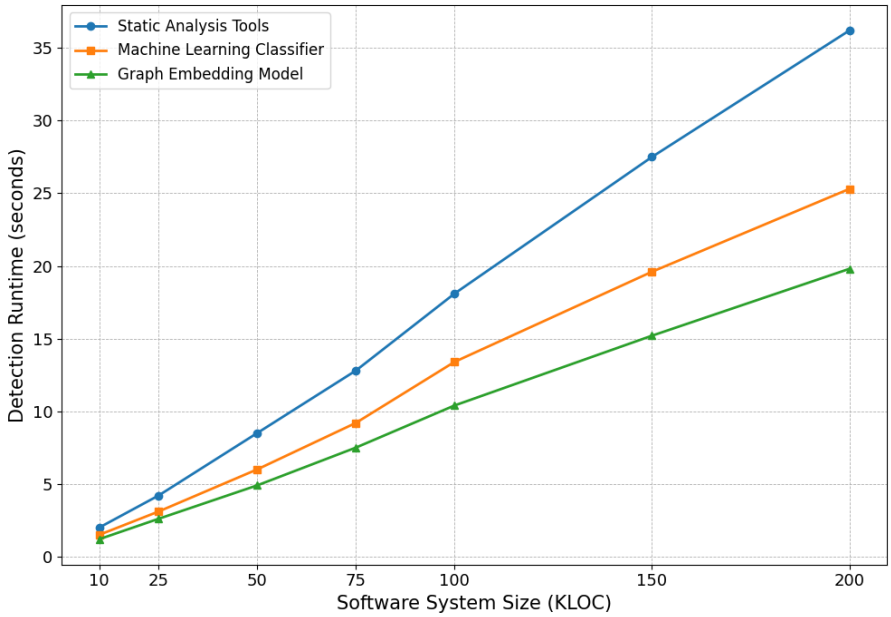


Fig. 15: Runtime Scalability of Vulnerability Detection Framework Across Software System Sizes

Table 1: Performance Comparison of Security Weakness Detection Methods

Method	Accuracy	Precision	Recall	F1 Score	Detection Time (seconds)
Static Analysis Tools	0.78	0.74	0.71	0.72	36.2
Machine Learning Classifier	0.86	0.84	0.82	0.83	25.3
Proposed Graph Embedding Model	0.92	0.90	0.91	0.90	19.8

Conclusion

The study proposed a framework that employs a graph embedding approach to detect security vulnerabilities of software systems. A significant challenge in modern software systems is the presence of intricate structural interconnections among program statements. This makes the traditional rule-based tools for vulnerability detection ineffective. This approach uses program dependency graphs to represent software behavior to learn the structure of the graphs through machine learning techniques. The framework, which encompasses control dependence and data flow, shows, for the first time, the detection of vulnerability patterns that are undetectable through traditional static analysis.

The experiment results in this study show the importance of structural graph embeddings to vulnerability detection in software systems. The software systems in the study varied widely, suggesting the methods used in the study-maintained consistency across the systems. This consistency can be observed in the improved detection rates compared to traditional, static analytical systems and standard machine learning classifiers. The model studied also demonstrated strong recall and precision across the spectrum of vulnerabilities. This can be observed in the receiver operating characteristic analysis, which illustrates the embedding-based model demonstrated improved classification capabilities when distinguishing between vulnerable and non-vulnerable segments of code.

The results display the pragmatic scalability of the framework. The model realizes structural analysis and embedding learning through program dependency graphs. However, the runtime evaluation shows that the size of the software system keeps the detection pipeline computationally reasonable. This means that the framework can be applied to real-world large-scale software repositories while maintaining reasonable computational demands.

In summary, the proposed graph embedding framework shows a significant advancement in the automated software vulnerability detection problem. The framework combines structural code analysis and graph representation learning to better understand the program's behavior and vulnerability propagation patterns. The future work can be extended to the integration of dynamic program analysis, other software repositories, the use of other sophisticated graph learning architectures, and large-scale software security analysis.

References

Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2021). Deep learning-based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, *48*(9), 3280-3296. https://doi.org/10.1109/TSE.2021.3087402

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., & Chen, Z. (2021). Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, *19*(4), 2244-2258. https://doi.org/10.1109/TDSC.2021.3051525

Liu, S. (2020, December). A unified framework to learn program semantics with graph neural networks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1364-1366). https://doi.org/10.1145/3324884.3418924

Lekeufack Foulefack, R. Z., & Marchetto, A. (2024, September). A rapid review on graph-based learning vulnerability detection. In *International Conference on the Quality of Information and Communications Technology* (pp. 355-372). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-70245-7_25

Wang, H., Ye, G., Tang, Z., Tan, S. H., Huang, S., Fang, D., ... & Wang, Z. (2020). Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, *16*, 1943-1958. https://doi.org/10.1109/TIFS.2020.3044773

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. https://doi.org/10.1109/TIFS.2024.3392536

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. https://doi.org/10.1016/j.scico.2024.103156

Xiao, H., Xing, Z., Li, X., & Guo, H. (2019, December). Embedding and predicting software security entity relationships: A knowledge graph-based approach. In *International Conference on Neural Information Processing* (pp. 50-63). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-36718-3_5

Wen, X. C., Chen, Y., Gao, C., Zhang, H., Zhang, J. M., & Liao, Q. (2023, May). Vulnerability detection with graph simplification and enhanced graph representation learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (pp. 2275-2286). IEEE. https://doi.org/10.1109/ICSE48619.2023.00191

Hin, D., Kan, A., Chen, H., & Babar, M. A. (2022, May). Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th international conference on mining software repositories* (pp. 596-607). https://doi.org/10.1145/3524842.3527949

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., ... & Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI open*, *1*, 57-81. https://doi.org/10.1016/j.aiopen.2021.01.001

Jia, X., Du, J., Fang, M., Liu, H., Li, Y., & Jiang, F. (2025). Vulnerability detection with graph enhancement and global dependency representation learning. *Automated Software Engineering*, *32*(1), 14. https://doi.org/10.1007/s10515-024-00484-3

Qiu, F., Liu, Z., Hu, X., Xia, X., Chen, G., & Wang, X. (2024). Vulnerability detection via multiple-graph-based code representation. *IEEE Transactions on Software Engineering*, *50*(8), 2178-2199. https://doi.org/10.1109/TSE.2024.3427815

Bi, R. (2025). An adaptive semantic retrieval framework for digital libraries integrating graph neural networks, ontology, and user behavior. *Scientific Reports*, *15*(1), 40528. https://doi.org/10.1038/s41598-025-24276-1

Lin, C., Xu, Y., Fang, Y., & Liu, Z. (2023). VulEye: A novel graph neural network vulnerability detection approach for PHP application. *Applied Sciences*, *13*(2), 825. https://doi.org/10.3390/app13020825

Wang, Y., Hou, X., Ma, X., & Lv, Q. (2022, November). A software security entity relationships prediction framework based on knowledge graph embedding using sentence-bert. In *International Conference on Wireless Algorithms, Systems, and Applications* (pp. 501-513). Cham: Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-19214-2_42

Li, X., Xin, Y., Zhu, H., Yang, Y., & Chen, Y. (2023). Cross-domain vulnerability detection using graph embedding and domain adaptation. *Computers & Security*, *125*, 103017. https://doi.org/10.1016/j.cose.2022.103017

Abdu, A., Zhai, Z., Abdo, H. A., & Algabri, R. (2024). Software defect prediction based on deep representation learning of source code from contextual syntax and semantic graph. *IEEE Transactions on Reliability*, *73*(2), 820-834. https://doi.org/10.1109/TR.2024.3354965

Cheng, X., Wang, H., Hua, J., Xu, G., & Sui, Y. (2021). Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(3), 1-33. https://doi.org/10.1145/3436877